

Computer Vision: Algorithms and Applications

2nd Edition

Richard Szeliski

Final draft, September 30, 2021

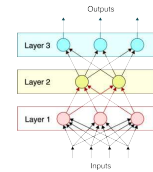
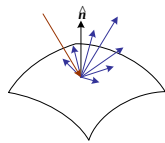
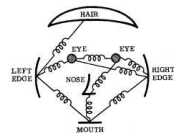
© 2022 Springer

This electronic draft was downloaded Decfi 27,fi 2022 for the personal use of
???

jacky870810@icloud.com
and may not be posted or re-distributed in any form.

Please refer interested readers to the book's Web site at
<https://szeliski.org/Book>, where you can also provide feedback.

*This book is dedicated to my parents,
Zdzisław and Jadwiga,
and my family,
Lyn, Anne, and Stephen.*



1 Introduction 1

What is computer vision? • A brief history •
Book overview • Sample syllabus • Notation

2 Image formation 33

Geometric primitives and transformations •
Photometric image formation • The digital camera

3 Image processing 107

Point operators • Linear filtering •
Non-linear filtering • Fourier transforms •
Pyramids and wavelets • Geometric transformations

4 Model fitting and optimization 191

Scattered data interpolation •
Variational methods and regularization •
Markov random fields

5 Deep learning 235

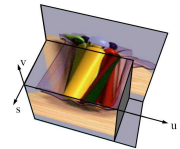
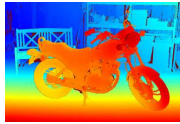
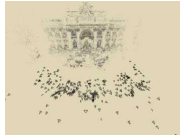
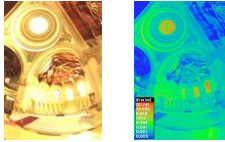
Supervised learning • Unsupervised learning •
Deep neural networks • Convolutional networks •
More complex models

6 Recognition 343

Instance recognition • Image classification •
Object detection • Semantic segmentation •
Video understanding • Vision and language

7 Feature detection and matching 417

Points and patches • Edges and contours •
Contour tracking • Lines and vanishing points •
Segmentation



8 Image alignment and stitching 501

- Pairwise alignment • Image stitching •
- Global alignment • Compositing

9 Motion estimation 555

- Translational alignment • Parametric motion •
- Optical flow • Layered motion

10 Computational photography 607

- Photometric calibration • High dynamic range imaging •
- Super-resolution, denoising, and blur removal •
- Image matting and compositing •
- Texture analysis and synthesis

11 Structure from motion and SLAM 681

- Geometric intrinsic calibration • Pose estimation •
- Two-frame structure from motion •
- Multi-frame structure from motion •
- Simultaneous localization and mapping (SLAM)

12 Depth estimation 749

- Epipolar geometry • Sparse correspondence •
- Dense correspondence • Local methods •
- Global optimization • Deep neural networks •
- Multi-view stereo • Monocular depth estimation

13 3D reconstruction 805

- Shape from X • 3D scanning •
- Surface representations • Point-based representations •
- Volumetric representations • Model-based reconstruction •
- Recovering texture maps and albedos

14 Image-based rendering 861

- View interpolation • Layered depth images •
- Light fields and Lumigraphs • Environment mattes •
- Video-based rendering • Neural rendering

Preface

The seeds for this book were first planted in 2001 when Steve Seitz at the University of Washington invited me to co-teach a course called “Computer Vision for Computer Graphics”. At that time, computer vision techniques were increasingly being used in computer graphics to create image-based models of real-world objects, to create visual effects, and to merge real-world imagery using computational photography techniques. Our decision to focus on the applications of computer vision to fun problems such as image stitching and photo-based 3D modeling from personal photos seemed to resonate well with our students.

That initial course evolved into a more complete computer vision syllabus and project-oriented course structure that I used to co-teach general computer vision courses both at the University of Washington and at Stanford. (The latter was a course I co-taught with David Fleet in 2003.) Similar curricula were then adopted at a number of other universities and also incorporated into more specialized courses on computational photography. (For ideas on how to use this book in your own course, please see Table 1.1 in Section 1.4.)

This book also reflects my 40 years’ experience doing computer vision research in corporate research labs, mostly at Digital Equipment Corporation’s Cambridge Research Lab, Microsoft Research, and Facebook. In pursuing my work, I have mostly focused on problems and solution techniques (algorithms) that have practical real-world applications and that work well in practice. Thus, this book has more emphasis on basic techniques that work under real-world conditions and less on more esoteric mathematics that has intrinsic elegance but less practical applicability.

This book is suitable for teaching a senior-level undergraduate course in computer vision to students in both computer science and electrical engineering. I prefer students to have either an image processing or a computer graphics course as a prerequisite, so that they can spend less time learning general background mathematics and more time studying computer vision techniques. The book is also suitable for teaching graduate-level courses in computer vision, e.g., by delving into more specialized topics, and as a general reference to fundamental

techniques and the recent research literature. To this end, I have attempted wherever possible to at least cite the newest research in each sub-field, even if the technical details are too complex to cover in the book itself.

In teaching our courses, we have found it useful for the students to attempt a number of small implementation projects, which often build on one another, in order to get them used to working with real-world images and the challenges that these present. The students are then asked to choose an individual topic for each of their small-group, final projects. (Sometimes these projects even turn into conference papers!) The exercises at the end of each chapter contain numerous suggestions for smaller mid-term projects, as well as more open-ended problems whose solutions are still active research topics. Wherever possible, I encourage students to try their algorithms on their own personal photographs, since this better motivates them, often leads to creative variants on the problems, and better acquaints them with the variety and complexity of real-world imagery.

In formulating and solving computer vision problems, I have often found it useful to draw inspiration from four high-level approaches:

- **Scientific:** build detailed models of the image formation process and develop mathematical techniques to invert these in order to recover the quantities of interest (where necessary, making simplifying assumptions to make the mathematics more tractable).
- **Statistical:** use probabilistic models to quantify the prior likelihood of your unknowns and the noisy measurement processes that produce the input images, then infer the best possible estimates of your desired quantities and analyze their resulting uncertainties. The inference algorithms used are often closely related to the optimization techniques used to invert the (scientific) image formation processes.
- **Engineering:** develop techniques that are simple to describe and implement but that are also known to work well in practice. Test these techniques to understand their limitation and failure modes, as well as their expected computational costs (run-time performance).
- **Data-driven:** collect a representative set of test data (ideally, with labels or ground-truth answers) and use these data to either tune or learn your model parameters, or at least to validate and quantify its performance.

These four approaches build on each other and are used throughout the book.

My personal research and development philosophy (and hence the exercises in the book) have a strong emphasis on *testing* algorithms. It's too easy in computer vision to develop an

algorithm that does something *plausible* on a few images rather than something *correct*. The best way to validate your algorithms is to use a three-part strategy.

First, test your algorithm on clean synthetic data, for which the exact results are known. Second, add noise to the data and evaluate how the performance degrades as a function of noise level. Finally, test the algorithm on real-world data, preferably drawn from a wide variety of sources, such as photos found on the web. Only then can you truly know if your algorithm can deal with real-world complexity, i.e., images that do not fit some simplified model or assumptions.

In order to help students in this process, Appendix C includes pointers to commonly used datasets and software libraries that contain implementations of a wide variety of computer vision algorithms, which can enable you to tackle more ambitious projects (with your instructor's consent).

Notes on the Second Edition

The last decade has seen a truly dramatic explosion in the performance and applicability of computer vision algorithms, much of it engendered by the application of machine learning algorithms to large amounts of visual training data (Su and Crandall 2021).

Deep neural networks now play an essential role in so many vision algorithms that the new edition of this book introduces them early on as a fundamental technique that gets used extensively in subsequent chapters.

The most notable changes in the second edition include:

- Machine learning, deep learning, and deep neural networks are introduced early on in Chapter 5, as they play just as fundamental a role in vision algorithms as more classical techniques, such as image processing, graphical/probabilistic models, and energy minimization, which are introduced in the preceding two chapters.
- The recognition chapter has been moved earlier in the book to Chapter 6, since end-to-end deep learning systems no longer require the development of building blocks such as feature detection, matching, and segmentation. Many of the students taking vision classes are primarily interested in visual recognition, so presenting this material earlier in the course makes it easier for students to base their final project on these topics. This chapter also includes sections on semantic segmentation, video understanding, and vision and language.
- The application of neural networks and deep learning to myriad computer vision algorithms and applications, including flow and stereo, 3D shape modeling, and newly

emerging fields such as neural rendering.

- New technologies such as SLAM (simultaneous localization and mapping) and VIO (visual inertial odometry) that now run reliably and are used in real-time applications such as augmented reality and autonomous navigation.

In addition to these larger changes, the book has been updated to reflect the latest state-of-the-art techniques such as internet-scale image search and phone-based computational photography. The new edition includes over 1500 new citations (papers) and has over 200 new figures.

Acknowledgements

I would like to gratefully acknowledge all of the people whose passion for research and inquiry as well as encouragement have helped me write this book.

Steve Zucker at McGill University first introduced me to computer vision, taught all of his students to question and debate research results and techniques, and encouraged me to pursue a graduate career in this area.

Takeo Kanade and Geoff Hinton, my PhD thesis advisors at Carnegie Mellon University, taught me the fundamentals of good research, writing, and presentation and mentored several generations of outstanding students and researchers. They fired up my interest in visual processing, 3D modeling, and statistical methods, while Larry Matthies introduced me to Kalman filtering and stereo matching. Geoff continues to inspire so many of us with this undiminished passion for trying to figure out “what makes the brain work”. It’s been a delight to see his pursuit of connectionist ideas bear so much fruit in this past decade.

Demetri Terzopoulos was my mentor at my first industrial research job and taught me the ropes of successful publishing. Yvan Leclerc and Pascal Fua, colleagues from my brief interlude at SRI International, gave me new perspectives on alternative approaches to computer vision.

During my six years of research at Digital Equipment Corporation’s Cambridge Research Lab, I was fortunate to work with a great set of colleagues, including Ingrid Carlbom, Gudrun Klinker, Keith Waters, William Hsu, Richard Weiss, Stéphane Lavallée, and Sing Bing Kang, as well as to supervise the first of a long string of outstanding summer interns, including David Tonnesen, Sing Bing Kang, James Coughlan, and Harry Shum. This is also where I began my long-term collaboration with Daniel Scharstein.

At Microsoft Research, I had the outstanding fortune to work with some of the world’s best researchers in computer vision and computer graphics, including Michael Cohen, Matt

Uyttendaele, Sing Bing Kang, Harry Shum, Larry Zitnick, Sudipta Sinha, Drew Steedly, Simon Baker, Johannes Kopf, Neel Joshi, Krishnan Ramnath, Anandan, Phil Torr, Antonio Criminisi, Simon Winder, Matthew Brown, Michael Goesele, Richard Hartley, Hugues Hoppe, Stephen Gortler, Steve Shafer, Matthew Turk, Georg Petschnigg, Kentaro Toyama, Ramin Zabih, Shai Avidan, Patrice Simard, Chris Pal, Nebojsa Jojic, Patrick Baudisch, Dani Lischinski, Raanan Fattal, Eric Stollnitz, David Nistér, Blaise Aguera y Arcas, Andrew Fitzgibbon, Jamie Shotton, Wolf Kienzle, Piotr Dollar, and Ross Girshick. I was also lucky to have as interns such great students as Polina Golland, Simon Baker, Mei Han, Arno Schödl, Ron Dror, Ashley Eden, Jonathan Shade, Jinxiang Chai, Rahul Swaminathan, Yanghai Tsin, Sam Hasi-noff, Anat Levin, Matthew Brown, Eric Bennett, Vaibhav Vaish, Jan-Michael Frahm, James Diebel, Ce Liu, Josef Sivic, Grant Schindler, Colin Zheng, Neel Joshi, Sudipta Sinha, Zeev Farbman, Rahul Garg, Tim Cho, Yekeun Jeong, Richard Roberts, Varsha Hedau, Dilip Krishnan, Adarsh Kowdle, Edward Hsiao, Yong Seok Heo, Fabian Langguth, Andrew Owens, and Tianfan Xue. Working with such outstanding students also gave me the opportunity to collaborate with some of their amazing advisors, including Bill Freeman, Irfan Essa, Marc Pollefeys, Michael Black, Marc Levoy, and Andrew Zisserman.

Since moving to Facebook, I've had the pleasure to continue my collaborations with Michael Cohen, Matt Uyttendaele, Johannes Kopf, Wolf Kienzle, and Krishnan Ramnath, and also new colleagues including Kevin Matzen, Bryce Evans, Suhib Alsisan, Changil Kim, David Geraghty, Jan Herling, Nils Plath, Jan-Michael Frahm, True Price, Richard Newcombe, Thomas Whelan, Michael Goesele, Steven Lovegrove, Julian Straub, Simon Green, Brian Cabral, Michael Toksvig, Albert Para Pozzo, Laura Sevilla-Lara, Georgia Gkioxari, Justin Johnson, Chris Sweeney, and Vassileios Balntas. I've also had the pleasure to collaborate with some outstanding summer interns, including Tianfan Xue, Scott Wehrwein, Peter Hedman, Joel Janai, Aleksander Hołystyński, Xuan Luo, Rui Wang, Olivia Wiles, and Yulun Tian. I'd like to thank in particular Michael Cohen, my mentor, colleague, and friend for the last 25 years for his unwavering support of my sprint to complete this second edition.

While working at Microsoft and Facebook, I've also had the opportunity to collaborate with wonderful colleagues at the University of Washington, where I hold an Affiliate Professor appointment. I'm indebted to Tony DeRose and David Salesin, who first encouraged me to get involved with the research going on at UW, my long-time collaborators Brian Curless, Steve Seitz, Maneesh Agrawala, Sameer Agarwal, and Yasu Furukawa, as well as the students I have had the privilege to supervise and interact with, including Frédéric Pighin, Yung-Yu Chuang, Doug Zongker, Colin Zheng, Aseem Agarwala, Dan Goldman, Noah Snavely, Ian Simon, Rahul Garg, Ryan Kaminsky, Juliet Fiss, Aleksander Hołystyński, and Yifan Wang. As I mentioned at the beginning of this preface, this book owes its inception to the vision course

that Steve Seitz invited me to co-teach, as well as to Steve's encouragement, course notes, and editorial input.

I'm also grateful to the many other computer vision researchers who have given me so many constructive suggestions about the book, including Sing Bing Kang, who was my informal book editor, Vladimir Kolmogorov, Daniel Scharstein, Richard Hartley, Simon Baker, Noah Snavely, Bill Freeman, Svetlana Lazebnik, Matthew Turk, Jitendra Malik, Alyosha Efros, Michael Black, Brian Curless, Sameer Agarwal, Li Zhang, Deva Ramanan, Olga Veksler, Yuri Boykov, Carsten Rother, Phil Torr, Bill Triggs, Bruce Maxwell, Rico Malvar, Jana Kosecká, Eero Simoncelli, Aaron Hertzmann, Antonio Torralba, Tomaso Poggio, Theo Pavlidis, Baba Vemuri, Nando de Freitas, Chuck Dyer, Song Yi, Falk Schubert, Roman Plugfelder, Marshall Tappen, James Coughlan, Sammy Rogmans, Klaus Strobel, Shanmuganathan, Andreas Siebert, Yongjun Wu, Fred Pighin, Juan Cockburn, Ronald Mallet, Tim Soper, Georgios Evangelidis, Dwight Fowler, Itzik Bayaz, Daniel O'Connor, Srikrishna Bhat, and Toru Tamaki, who wrote the Japanese translation and provided many useful errata.

For the second edition, I received significant help and advice from three key contributors. Daniel Scharstein helped me update the chapter on stereo, Matt Deitke contributed descriptions of the newest papers in deep learning, including the sections on transformers, variational autoencoders, and text-to-image synthesis, along with the exercises in Chapters 5 and 6 and some illustrations. Sing Bing Kang reviewed multiple drafts and provided useful suggestions. I'd also like to thank Andrew Glassner, whose book (Glassner 2018) and figures were a tremendous help, Justin Johnson, Sean Bell, Ishan Misra, David Fouhey, Michael Brown, Abdelrahman Abdelhamed, Frank Dellaert, Xinlei Chen, Ross Girshick, Andreas Geiger, Dmytro Mishkin, Aleksander Hołyński, Joel Janai, Christoph Feichtenhofer, Yuandong Tian, Alyosha Efros, Pascal Fua, Torsten Sattler, Laura Leal-Taixé, Aljosa Osep, Qunjie Zhou, Jirí Matas, Eddy Ilg, Yann LeCun, Larry Jackel, Vasileios Balntas, Daniel DeTone, Zachary Teed, Junhwa Hur, Jun-Yan Zhu, Filip Radenović, Michael Zollhöfer, Matthias Nießner, Andrew Owens, Hervé Jégou, Luowei Zhou, Ricardo Martin Brualla, Pratul Srinivasan, Matteo Poggi, Fabio Tosi, Ahmed Osman, Dave Howell, Holger Heidrich, Howard Yen, Anton Papst, Syamprasad K. Rajagopalan, Abhishek Nagar, Vladimir Kuznetsov, Raphaël Fouque, Marian Ciobanu, Darko Simonovic, and Guilherme Schlinker.

In preparing the second edition, I taught some of the new material in two courses that I helped co-teach in 2020 at Facebook and UW. I'd like to thank my co-instructors Jan-Michael Frahm, Michael Goesele, Georgia Gkioxari, Ross Girshick, Jakob Julian Engel, Daniel Scharstein, Fernando de la Torre, Steve Seitz, and Harpreet Sawhney, from whom I learned a lot about the latest techniques that are included in the new edition. I'd also like to thank the TAs, including David Geraghty, True Price, Kevin Matzen, Akash Bapat, Alek-

sander HoŹyński, Keunhong Park, and Svetoslav Kolev, for the wonderful job they did in creating and grading the assignments. I'd like to give a special thanks to Justin Johnson, whose excellent class slides (Johnson 2020), based on earlier slides from Stanford (Li, Johnson, and Yeung 2019), taught me the fundamentals of deep learning and which I used extensively in my own class and in preparing the new chapter on deep learning.

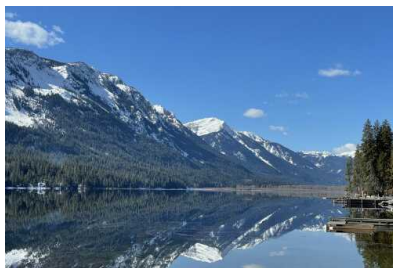
Shena Deuchers and Ian Kingston did a fantastic job copy-editing the first and second editions, respectively and suggesting many useful improvements, and Wayne Wheeler and Simon Rees at Springer were most helpful throughout the whole book publishing process. Keith Price's Annotated Computer Vision Bibliography was invaluable in tracking down references and related work.

If you have any suggestions for improving the book, please send me an e-mail, as I would like to keep the book as accurate, informative, and timely as possible.

The last year of writing this second edition took place during the worldwide COVID-19 pandemic. I would like to thank all of the first responders, medical and front-line workers, and everyone else who helped get us through these difficult and challenging times and to acknowledge the impact that this and other recent tragedies have had on all of us.

Lastly, this book would not have been possible or worthwhile without the incredible support and encouragement of my family. I dedicate this book to my parents, Zdzisław and Jadwiga, whose love, generosity, and accomplishments always inspired me; to my sister Basia for her lifelong friendship; and especially to Lyn, Anne, and Stephen, whose love and support in all matters (including my book projects) makes it all worthwhile.

*Lake Wenatchee
May 2021*



Contents

Preface	vii
Contents	xv
1 Introduction	1
1.1 What is computer vision?	3
1.2 A brief history	10
1.3 Book overview	22
1.4 Sample syllabus	30
1.5 A note on notation	31
1.6 Additional reading	31
2 Image formation	33
2.1 Geometric primitives and transformations	36
2.1.1 2D transformations	40
2.1.2 3D transformations	43
2.1.3 3D rotations	45
2.1.4 3D to 2D projections	51
2.1.5 Lens distortions	63
2.2 Photometric image formation	66
2.2.1 Lighting	66
2.2.2 Reflectance and shading	67
2.2.3 Optics	74
2.3 The digital camera	79
2.3.1 Sampling and aliasing	84
2.3.2 Color	87
2.3.3 Compression	98

2.4	Additional reading	101
2.5	Exercises	102
3	Image processing	107
3.1	Point operators	109
3.1.1	Pixel transforms	111
3.1.2	Color transforms	112
3.1.3	Compositing and matting	113
3.1.4	Histogram equalization	115
3.1.5	<i>Application: Tonal adjustment</i>	119
3.2	Linear filtering	119
3.2.1	Separable filtering	124
3.2.2	Examples of linear filtering	125
3.2.3	Band-pass and steerable filters	127
3.3	More neighborhood operators	131
3.3.1	Non-linear filtering	132
3.3.2	Bilateral filtering	133
3.3.3	Binary image processing	138
3.4	Fourier transforms	142
3.4.1	Two-dimensional Fourier transforms	146
3.4.2	<i>Application: Sharpening, blur, and noise removal</i>	148
3.5	Pyramids and wavelets	149
3.5.1	Interpolation	150
3.5.2	Decimation	153
3.5.3	Multi-resolution representations	154
3.5.4	Wavelets	159
3.5.5	<i>Application: Image blending</i>	165
3.6	Geometric transformations	168
3.6.1	Parametric transformations	168
3.6.2	Mesh-based warping	175
3.6.3	<i>Application: Feature-based morphing</i>	177
3.7	Additional reading	178
3.8	Exercises	180
4	Model fitting and optimization	191
4.1	Scattered data interpolation	194
4.1.1	Radial basis functions	196

4.1.2	Overfitting and underfitting	199
4.1.3	Robust data fitting	202
4.2	Variational methods and regularization	204
4.2.1	Discrete energy minimization	206
4.2.2	Total variation	210
4.2.3	Bilateral solver	210
4.2.4	<i>Application: Interactive colorization</i>	211
4.3	Markov random fields	212
4.3.1	Conditional random fields	222
4.3.2	<i>Application: Interactive segmentation</i>	227
4.4	Additional reading	230
4.5	Exercises	232
5	Deep Learning	235
5.1	Supervised learning	239
5.1.1	Nearest neighbors	241
5.1.2	Bayesian classification	243
5.1.3	Logistic regression	248
5.1.4	Support vector machines	250
5.1.5	Decision trees and forests	254
5.2	Unsupervised learning	257
5.2.1	Clustering	257
5.2.2	K-means and Gaussians mixture models	259
5.2.3	Principal component analysis	262
5.2.4	Manifold learning	265
5.2.5	Semi-supervised learning	266
5.3	Deep neural networks	268
5.3.1	Weights and layers	270
5.3.2	Activation functions	272
5.3.3	Regularization and normalization	274
5.3.4	Loss functions	280
5.3.5	Backpropagation	284
5.3.6	Training and optimization	287
5.4	Convolutional neural networks	291
5.4.1	Pooling and unpooling	295
5.4.2	<i>Application: Digit classification</i>	298

5.4.3	Network architectures	299
5.4.4	Model zoos	304
5.4.5	Visualizing weights and activations	307
5.4.6	Adversarial examples	311
5.4.7	Self-supervised learning	312
5.5	More complex models	317
5.5.1	Three-dimensional CNNs	317
5.5.2	Recurrent neural networks	321
5.5.3	Transformers	322
5.5.4	Generative models	328
5.6	Additional reading	336
5.7	Exercises	337
6	Recognition	343
6.1	Instance recognition	346
6.2	Image classification	349
6.2.1	Feature-based methods	350
6.2.2	Deep networks	358
6.2.3	<i>Application: Visual similarity search</i>	360
6.2.4	Face recognition	363
6.3	Object detection	370
6.3.1	Face detection	371
6.3.2	Pedestrian detection	376
6.3.3	General object detection	379
6.4	Semantic segmentation	387
6.4.1	<i>Application: Medical image segmentation</i>	390
6.4.2	Instance segmentation	391
6.4.3	Panoptic segmentation	392
6.4.4	<i>Application: Intelligent photo editing</i>	394
6.4.5	Pose estimation	395
6.5	Video understanding	396
6.6	Vision and language	400
6.7	Additional reading	409
6.8	Exercises	413

7	Feature detection and matching	417
7.1	Points and patches	419
7.1.1	Feature detectors	422
7.1.2	Feature descriptors	434
7.1.3	Feature matching	441
7.1.4	Large-scale matching and retrieval	448
7.1.5	Feature tracking	452
7.1.6	<i>Application: Performance-driven animation</i>	454
7.2	Edges and contours	455
7.2.1	Edge detection	456
7.2.2	Contour detection	461
7.2.3	<i>Application: Edge editing and enhancement</i>	465
7.3	Contour tracking	466
7.3.1	Snakes and scissors	467
7.3.2	Level Sets	474
7.3.3	<i>Application: Contour tracking and rotoscoping</i>	476
7.4	Lines and vanishing points	477
7.4.1	Successive approximation	477
7.4.2	Hough transforms	477
7.4.3	Vanishing points	481
7.5	Segmentation	483
7.5.1	Graph-based segmentation	486
7.5.2	Mean shift	487
7.5.3	Normalized cuts	489
7.6	Additional reading	491
7.7	Exercises	495
8	Image alignment and stitching	501
8.1	Pairwise alignment	503
8.1.1	2D alignment using least squares	504
8.1.2	<i>Application: Panography</i>	506
8.1.3	Iterative algorithms	507
8.1.4	Robust least squares and RANSAC	510
8.1.5	3D alignment	513
8.2	Image stitching	514
8.2.1	Parametric motion models	516

8.2.2	<i>Application: Whiteboard and document scanning</i>	517
8.2.3	Rotational panoramas	519
8.2.4	Gap closing	520
8.2.5	<i>Application: Video summarization and compression</i>	522
8.2.6	Cylindrical and spherical coordinates	523
8.3	Global alignment	526
8.3.1	Bundle adjustment	527
8.3.2	Parallax removal	531
8.3.3	Recognizing panoramas	533
8.4	Compositing	536
8.4.1	Choosing a compositing surface	536
8.4.2	Pixel selection and weighting (deghosting)	538
8.4.3	<i>Application: Photomontage</i>	544
8.4.4	Blending	544
8.5	Additional reading	547
8.6	Exercises	549
9	Motion estimation	555
9.1	Translational alignment	558
9.1.1	Hierarchical motion estimation	562
9.1.2	Fourier-based alignment	563
9.1.3	Incremental refinement	566
9.2	Parametric motion	570
9.2.1	<i>Application: Video stabilization</i>	573
9.2.2	Spline-based motion	575
9.2.3	<i>Application: Medical image registration</i>	577
9.3	Optical flow	578
9.3.1	Deep learning approaches	584
9.3.2	<i>Application: Rolling shutter wobble removal</i>	587
9.3.3	Multi-frame motion estimation	587
9.3.4	<i>Application: Video denoising</i>	589
9.4	Layered motion	589
9.4.1	<i>Application: Frame interpolation</i>	593
9.4.2	Transparent layers and reflections	594
9.4.3	Video object segmentation	597
9.4.4	Video object tracking	598

9.5	Additional reading	600
9.6	Exercises	602
10	Computational photography	607
10.1	Photometric calibration	610
10.1.1	Radiometric response function	611
10.1.2	Noise level estimation	614
10.1.3	Vignetting	615
10.1.4	Optical blur (spatial response) estimation	616
10.2	High dynamic range imaging	620
10.2.1	Tone mapping	627
10.2.2	<i>Application: Flash photography</i>	634
10.3	Super-resolution, denoising, and blur removal	637
10.3.1	Color image demosaicing	646
10.3.2	Lens blur (bokeh)	648
10.4	Image matting and compositing	650
10.4.1	Blue screen matting	651
10.4.2	Natural image matting	653
10.4.3	Optimization-based matting	656
10.4.4	Smoke, shadow, and lash matting	661
10.4.5	Video matting	662
10.5	Texture analysis and synthesis	663
10.5.1	<i>Application: Hole filling and inpainting</i>	665
10.5.2	<i>Application: Non-photorealistic rendering</i>	667
10.5.3	Neural style transfer and semantic image synthesis	669
10.6	Additional reading	671
10.7	Exercises	674
11	Structure from motion and SLAM	681
11.1	Geometric intrinsic calibration	685
11.1.1	Vanishing points	687
11.1.2	<i>Application: Single view metrology</i>	688
11.1.3	Rotational motion	689
11.1.4	Radial distortion	691
11.2	Pose estimation	693
11.2.1	Linear algorithms	693
11.2.2	Iterative non-linear algorithms	695

11.2.3	<i>Application: Location recognition</i>	698
11.2.4	Triangulation	701
11.3	Two-frame structure from motion	703
11.3.1	Eight, seven, and five-point algorithms	703
11.3.2	Special motions and structures	708
11.3.3	Projective (uncalibrated) reconstruction	710
11.3.4	Self-calibration	712
11.3.5	<i>Application: View morphing</i>	714
11.4	Multi-frame structure from motion	715
11.4.1	Factorization	715
11.4.2	Bundle adjustment	717
11.4.3	Exploiting sparsity	719
11.4.4	<i>Application: Match move</i>	723
11.4.5	Uncertainty and ambiguities	723
11.4.6	<i>Application: Reconstruction from internet photos</i>	725
11.4.7	Global structure from motion	728
11.4.8	Constrained structure and motion	731
11.5	Simultaneous localization and mapping (SLAM)	734
11.5.1	<i>Application: Autonomous navigation</i>	737
11.5.2	<i>Application: Smartphone augmented reality</i>	739
11.6	Additional reading	740
11.7	Exercises	743
12	Depth estimation	749
12.1	Epipolar geometry	753
12.1.1	Rectification	755
12.1.2	Plane sweep	757
12.2	Sparse correspondence	760
12.2.1	3D curves and profiles	760
12.3	Dense correspondence	762
12.3.1	Similarity measures	764
12.4	Local methods	766
12.4.1	Sub-pixel estimation and uncertainty	768
12.4.2	<i>Application: Stereo-based head tracking</i>	769
12.5	Global optimization	771
12.5.1	Dynamic programming	774

12.5.2	Segmentation-based techniques	775
12.5.3	<i>Application: Z-keying and background replacement</i>	777
12.6	Deep neural networks	778
12.7	Multi-view stereo	781
12.7.1	Scene flow	785
12.7.2	Volumetric and 3D surface reconstruction	786
12.7.3	Shape from silhouettes	794
12.8	Monocular depth estimation	796
12.9	Additional reading	799
12.10	Exercises	800
13	3D reconstruction	805
13.1	Shape from X	809
13.1.1	Shape from shading and photometric stereo	809
13.1.2	Shape from texture	814
13.1.3	Shape from focus	814
13.2	3D scanning	816
13.2.1	Range data merging	820
13.2.2	<i>Application: Digital heritage</i>	824
13.3	Surface representations	825
13.3.1	Surface interpolation	826
13.3.2	Surface simplification	827
13.3.3	Geometry images	828
13.4	Point-based representations	829
13.5	Volumetric representations	830
13.5.1	Implicit surfaces and level sets	831
13.6	Model-based reconstruction	833
13.6.1	Architecture	833
13.6.2	Facial modeling and tracking	838
13.6.3	<i>Application: Facial animation</i>	839
13.6.4	Human body modeling and tracking	843
13.7	Recovering texture maps and albedos	850
13.7.1	Estimating BRDFs	852
13.7.2	<i>Application: 3D model capture</i>	854
13.8	Additional reading	855
13.9	Exercises	857

14 Image-based rendering	861
14.1 View interpolation	863
14.1.1 View-dependent texture maps	865
14.1.2 <i>Application: Photo Tourism</i>	867
14.2 Layered depth images	868
14.2.1 Impostors, sprites, and layers	869
14.2.2 <i>Application: 3D photography</i>	872
14.3 Light fields and Lumigraphs	875
14.3.1 Unstructured Lumigraph	879
14.3.2 Surface light fields	880
14.3.3 <i>Application: Concentric mosaics</i>	882
14.3.4 <i>Application: Synthetic re-focusing</i>	883
14.4 Environment mattes	883
14.4.1 Higher-dimensional light fields	885
14.4.2 The modeling to rendering continuum	886
14.5 Video-based rendering	887
14.5.1 Video-based animation	888
14.5.2 Video textures	889
14.5.3 <i>Application: Animating pictures</i>	892
14.5.4 3D and free-viewpoint Video	893
14.5.5 <i>Application: Video-based walkthroughs</i>	896
14.6 Neural rendering	899
14.7 Additional reading	908
14.8 Exercises	910
15 Conclusion	915
A Linear algebra and numerical techniques	919
A.1 Matrix decompositions	920
A.1.1 Singular value decomposition	921
A.1.2 Eigenvalue decomposition	922
A.1.3 QR factorization	925
A.1.4 Cholesky factorization	925
A.2 Linear least squares	927
A.2.1 Total least squares	929
A.3 Non-linear least squares	930
A.4 Direct sparse matrix techniques	932

Preface	xxv
A.4.1 Variable reordering	932
A.5 Iterative techniques	934
A.5.1 Conjugate gradient	934
A.5.2 Preconditioning	936
A.5.3 Multigrid	937
B Bayesian modeling and inference	939
B.1 Estimation theory	941
B.2 Maximum likelihood estimation and least squares	943
B.3 Robust statistics	945
B.4 Prior models and Bayesian inference	948
B.5 Markov random fields	949
B.6 Uncertainty estimation (error analysis)	952
C Supplementary material	953
C.1 Datasets and benchmarks	954
C.2 Software	961
C.3 Slides and lectures	970
References	973
Index	1179

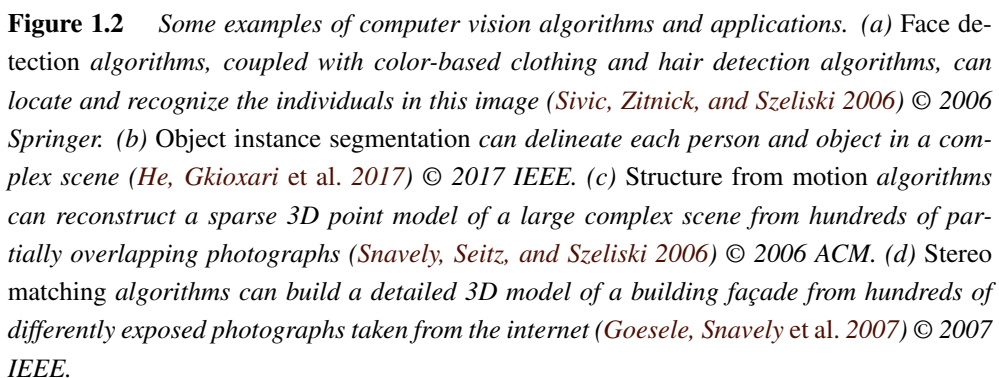
Chapter 1

Introduction

1.1	What is computer vision?	3
1.2	A brief history	10
1.3	Book overview	22
1.4	Sample syllabus	30
1.5	A note on notation	31
1.6	Additional reading	31



Figure 1.1 *The human visual system has no problem interpreting the subtle variations in translucency and shading in this photograph and correctly segmenting the object from its background.*



1.1 What is computer vision?

As humans, we perceive the three-dimensional structure of the world around us with apparent ease. Think of how vivid the three-dimensional percept is when you look at a vase of flowers sitting on the table next to you. You can tell the shape and translucency of each petal through the subtle patterns of light and shading that play across its surface and effortlessly segment each flower from the background of the scene (Figure 1.1). Looking at a framed group portrait, you can easily count and name all of the people in the picture and even guess at their emotions from their facial expressions (Figure 1.2a). Perceptual psychologists have spent decades trying to understand how the visual system works and, even though they can devise optical illusions¹ to tease apart some of its principles (Figure 1.3), a complete solution to this puzzle remains elusive (Marr 1982; Wandell 1995; Palmer 1999; Livingstone 2008; Frisby and Stone 2010).

Researchers in computer vision have been developing, in parallel, mathematical techniques for recovering the three-dimensional shape and appearance of objects in imagery. Here, the progress in the last two decades has been rapid. We now have reliable techniques for accurately computing a 3D model of an environment from thousands of partially overlapping photographs (Figure 1.2c). Given a large enough set of views of a particular object or façade, we can create accurate dense 3D surface models using stereo matching (Figure 1.2d). We can even, with moderate success, delineate most of the people and objects in a photograph (Figure 1.2a). However, despite all of these advances, the dream of having a computer explain an image at the same level of detail and causality as a two-year old remains elusive.

Why is vision so difficult? In part, it is because it is an *inverse problem*, in which we seek to recover some unknowns given insufficient information to fully specify the solution. We must therefore resort to physics-based and probabilistic *models*, or machine learning from large sets of examples, to disambiguate between potential solutions. However, modeling the visual world in all of its rich complexity is far more difficult than, say, modeling the vocal tract that produces spoken sounds.

The *forward* models that we use in computer vision are usually developed in physics (radiometry, optics, and sensor design) and in computer graphics. Both of these fields model how objects move and animate, how light reflects off their surfaces, is scattered by the atmosphere, refracted through camera lenses (or human eyes), and finally projected onto a flat (or curved) image plane. While computer graphics are not yet perfect, in many domains, such as rendering a still scene composed of everyday objects or animating extinct creatures such

¹Some fun pages with striking illusions include <https://michaelbach.de/ot>, <https://www.illusionsindex.org>, and <http://www.ritsumei.ac.jp/~akitaoka/index-e.html>.

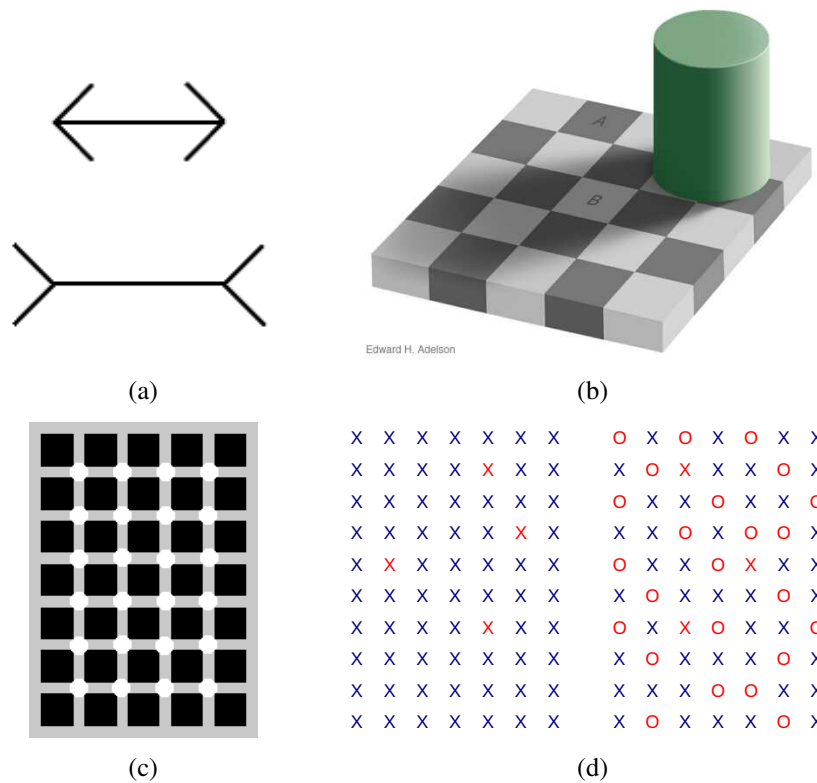


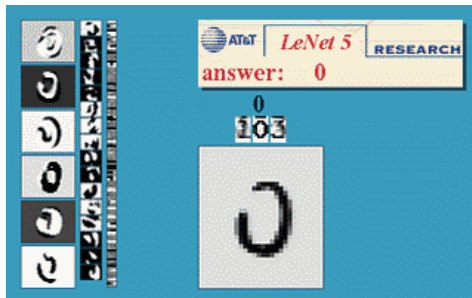
Figure 1.3 Some common optical illusions and what they might tell us about the visual system: (a) The classic Müller-Lyer illusion, where the lengths of the two horizontal lines appear different, probably due to the imagined perspective effects. (b) The “white” square B in the shadow and the “black” square A in the light actually have the same absolute intensity value. The percept is due to brightness constancy, the visual system’s attempt to discount illumination when interpreting colors. Image courtesy of Ted Adelson, <http://persci.mit.edu/gallery/checkershadow>. (c) A variation of the Hermann grid illusion, courtesy of Hany Farid. As you move your eyes over the figure, gray spots appear at the intersections. (d) Count the red Xs in the left half of the figure. Now count them in the right half. Is it significantly harder? The explanation has to do with a pop-out effect (Treisman 1985), which tells us about the operations of parallel perception and integration pathways in the brain.

as dinosaurs, the illusion of reality is essentially there.

In computer vision, we are trying to do the inverse, i.e., to describe the world that we see in one or more images and to reconstruct its properties, such as shape, illumination, and color distributions. It is amazing that humans and animals do this so effortlessly, while computer vision algorithms are so error prone. People who have not worked in the field often underestimate the difficulty of the problem. This misperception that vision should be easy dates back to the early days of artificial intelligence (see Section 1.2), when it was initially believed that the *cognitive* (logic proving and planning) parts of intelligence were intrinsically more difficult than the *perceptual* components (Boden 2006).

The good news is that computer vision *is* being used today in a wide variety of real-world applications, which include:

- **Optical character recognition (OCR):** reading handwritten postal codes on letters (Figure 1.4a) and automatic number plate recognition (ANPR);
- **Machine inspection:** rapid parts inspection for quality assurance using stereo vision with specialized illumination to measure tolerances on aircraft wings or auto body parts (Figure 1.4b) or looking for defects in steel castings using X-ray vision;
- **Retail:** object recognition for automated checkout lanes and fully automated stores (Wingfield 2019);
- **Warehouse logistics:** autonomous package delivery and pallet-carrying “drives” (Guizzo 2008; O’Brian 2019) and parts picking by robotic manipulators (Figure 1.4c; Ackerman 2020);
- **Medical imaging:** registering pre-operative and intra-operative imagery (Figure 1.4d) or performing long-term studies of people’s brain morphology as they age;
- **Self-driving vehicles:** capable of driving point-to-point between cities (Figure 1.4e; Montemerlo, Becker *et al.* 2008; Urmson, Anhalt *et al.* 2008; Janai, Güney *et al.* 2020) as well as autonomous flight (Kaufmann, Gehrig *et al.* 2019);
- **3D model building (photogrammetry):** fully automated construction of 3D models from aerial and drone photographs (Figure 1.4f);
- **Match move:** merging computer-generated imagery (CGI) with live action footage by tracking feature points in the source video to estimate the 3D camera motion and shape of the environment. Such techniques are widely used in Hollywood, e.g., in movies such as Jurassic Park (Roble 1999; Roble and Zafar 2009); they also require the use of



(a)



(b)



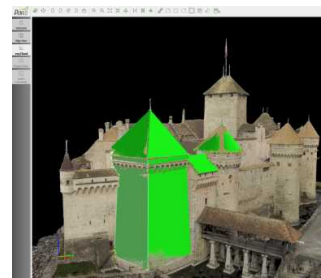
(c)



(d)



(e)



(f)

Figure 1.4 Some industrial applications of computer vision: (a) optical character recognition (OCR), <http://yann.lecun.com/exdb/lenet>; (b) mechanical inspection, <http://www.cognitens.com>; (c) warehouse picking, <https://covariant.ai>; (d) medical imaging, <http://www.clarontech.com>; (e) self-driving cars, (Montemerlo, Becker et al. 2008) © 2008 Wiley; (f) drone-based photogrammetry, <https://www.pix4d.com/blog/mapping-chillon-castle-with-drone>.

precise *matting* to insert new elements between foreground and background elements (Chuang, Agarwala *et al.* 2002).

- **Motion capture (mocap):** using retro-reflective markers viewed from multiple cameras or other vision-based techniques to capture actors for computer animation;
- **Surveillance:** monitoring for intruders, analyzing highway traffic and monitoring pools for drowning victims (e.g., <https://swimeye.com>);
- **Fingerprint recognition and biometrics:** for automatic access authentication as well as forensic applications.

David Lowe’s website of industrial vision applications (<http://www.cs.ubc.ca/spider/lowe/vision.html>) lists many other interesting industrial applications of computer vision. While the above applications are all extremely important, they mostly pertain to fairly specialized kinds of imagery and narrow domains.

In addition to all of these industrial applications, there exist myriad *consumer-level* applications, such as things you can do with your own personal photographs and video. These include:

- **Stitching:** turning overlapping photos into a single seamlessly stitched panorama (Figure 1.5a), as described in Section 8.2;
- **Exposure bracketing:** merging multiple exposures taken under challenging lighting conditions (strong sunlight and shadows) into a single perfectly exposed image (Figure 1.5b), as described in Section 10.2;
- **Morphing:** turning a picture of one of your friends into another, using a seamless *morph* transition (Figure 1.5c);
- **3D modeling:** converting one or more snapshots into a 3D model of the object or person you are photographing (Figure 1.5d), as described in Section 13.6;
- **Video match move and stabilization:** inserting 2D pictures or 3D models into your videos by automatically tracking nearby reference points (see Section 11.4.4)² or using motion estimates to remove shake from your videos (see Section 9.2.1);
- **Photo-based walkthroughs:** navigating a large collection of photographs, such as the interior of your house, by flying between different photos in 3D (see Sections 14.1.2 and 14.5.5);

²For a fun student project on this topic, see the “PhotoBook” project at <http://www.cc.gatech.edu/dvfx/videos/dvfx2005.html>.

- **Face detection:** for improved camera focusing as well as more relevant image searching (see Section 6.3.1);
- **Visual authentication:** automatically logging family members onto your home computer as they sit down in front of the webcam (see Section 6.2.4).

The great thing about these applications is that they are already familiar to most students; they are, at least, technologies that students can immediately appreciate and use with their own personal media. Since computer vision is a challenging topic, given the wide range of mathematics being covered³ and the intrinsically difficult nature of the problems being solved, having fun and relevant problems to work on can be highly motivating and inspiring.

The other major reason why this book has a strong focus on applications is that they can be used to *formulate* and *constrain* the potentially open-ended problems endemic in vision. Thus, it is better to think back from the problem at hand to suitable techniques, rather than to grab the first technique that you may have heard of. This kind of working back from problems to solutions is typical of an **engineering** approach to the study of vision and reflects my own background in the field.

First, I come up with a detailed problem definition and decide on the constraints and specifications for the problem. Then, I try to find out which techniques are known to work, implement a few of these, evaluate their performance, and finally make a selection. In order for this process to work, it is important to have realistic **test data**, both synthetic, which can be used to verify correctness and analyze noise sensitivity, and real-world data typical of the way the system will finally be used. If machine learning is being used, it is even more important to have representative unbiased **training data** in sufficient quantity to obtain good results on real-world inputs.

However, this book is not just an engineering text (a source of recipes). It also takes a **scientific** approach to basic vision problems. Here, I try to come up with the best possible models of the physics of the system at hand: how the scene is created, how light interacts with the scene and atmospheric effects, and how the sensors work, including sources of noise and uncertainty. The task is then to try to invert the acquisition process to come up with the best possible description of the scene.

The book often uses a **statistical** approach to formulating and solving computer vision problems. Where appropriate, probability distributions are used to model the scene and the noisy image acquisition process. The association of prior distributions with unknowns is often called *Bayesian modeling* (Appendix B). It is possible to associate a risk or loss function with

³These techniques include physics, Euclidean and projective geometry, statistics, and optimization. They make computer vision a fascinating field to study and a great way to learn techniques widely applicable in other fields.

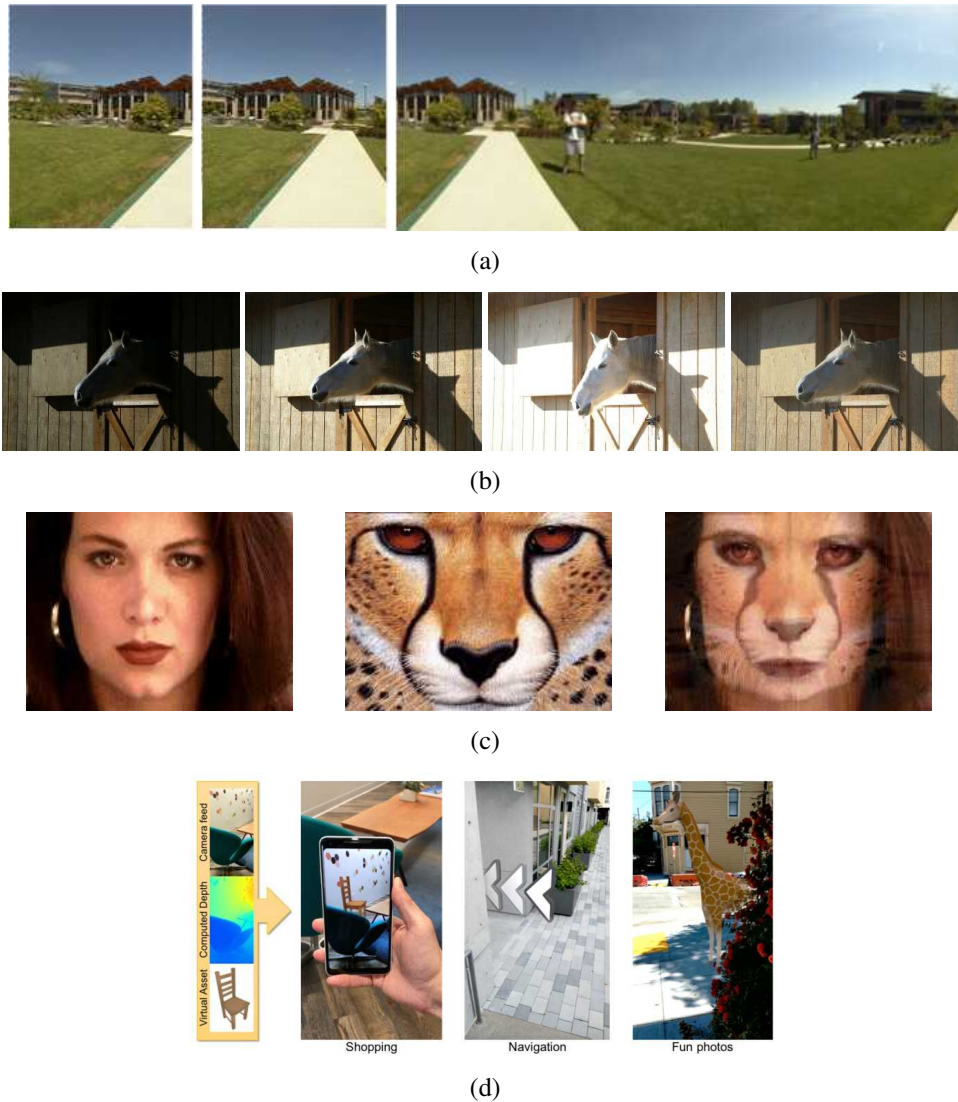


Figure 1.5 Some consumer applications of computer vision: (a) image stitching: merging different views (Szeliski and Shum 1997) © 1997 ACM; (b) exposure bracketing: merging different exposures; (c) morphing: blending between two photographs (Gomes, Darsa et al. 1999) © 1999 Morgan Kaufmann; (d) smartphone augmented reality showing real-time depth occlusion effects (Valentin, Kowdle et al. 2018) © 2018 ACM.

misestimating the answer (Section B.2) and to set up your inference algorithm to minimize the expected risk. (Consider a robot trying to estimate the distance to an obstacle: it is usually safer to underestimate than to overestimate.) With statistical techniques, it often helps to gather lots of training data from which to learn probabilistic models. Finally, statistical approaches enable you to use proven inference techniques to estimate the best answer (or distribution of answers) and to quantify the uncertainty in the resulting estimates.

Because so much of computer vision involves the solution of inverse problems or the estimation of unknown quantities, my book also has a heavy emphasis on **algorithms**, especially those that are known to work well in practice. For many vision problems, it is all too easy to come up with a mathematical description of the problem that either does not match realistic real-world conditions or does not lend itself to the stable estimation of the unknowns. What we need are algorithms that are both **robust** to noise and deviation from our models and reasonably **efficient** in terms of run-time resources and space. In this book, I go into these issues in detail, using Bayesian techniques, where applicable, to ensure robustness, and efficient search, minimization, and linear system solving algorithms to ensure efficiency.⁴ Most of the algorithms described in this book are at a high level, being mostly a list of steps that have to be filled in by students or by reading more detailed descriptions elsewhere. In fact, many of the algorithms are sketched out in the exercises.

Now that I’ve described the goals of this book and the frameworks that I use, I devote the rest of this chapter to two additional topics. Section 1.2 is a brief synopsis of the history of computer vision. It can easily be skipped by those who want to get to “the meat” of the new material in this book and do not care as much about who invented what when.

The second is an overview of the book’s contents, Section 1.3, which is useful reading for everyone who intends to make a study of this topic (or to jump in partway, since it describes chapter interdependencies). This outline is also useful for instructors looking to structure one or more courses around this topic, as it provides sample curricula based on the book’s contents.

1.2 A brief history

In this section, I provide a brief personal synopsis of the main developments in computer vision over the last fifty years (Figure 1.6) with a focus on advances I find personally interesting and that have stood the test of time. Readers not interested in the provenance of various ideas and the evolution of this field should skip ahead to the book overview in Section 1.3.

⁴In some cases, deep neural networks have also been shown to be an effective way to speed up algorithms that previously relied on iteration (Chen, Xu, and Koltun 2017).

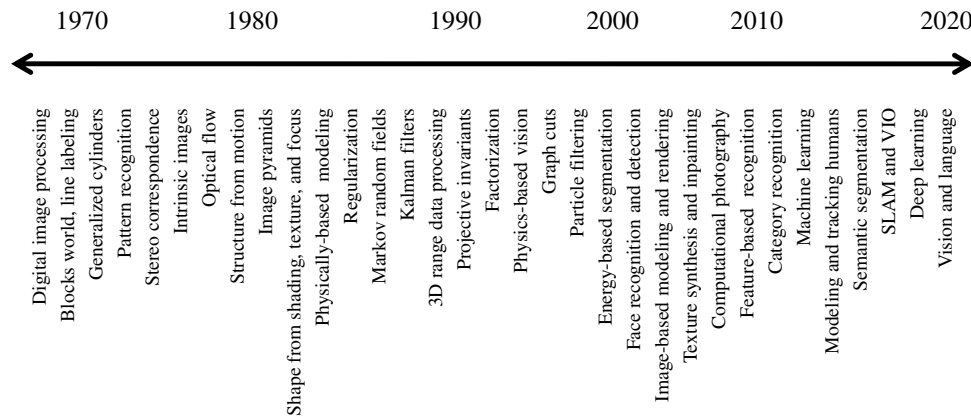


Figure 1.6 A rough timeline of some of the most active topics of research in computer vision.

1970s. When computer vision first started out in the early 1970s, it was viewed as the visual perception component of an ambitious agenda to mimic human intelligence and to endow robots with intelligent behavior. At the time, it was believed by some of the early pioneers of artificial intelligence and robotics (at places such as MIT, Stanford, and CMU) that solving the “visual input” problem would be an easy step along the path to solving more difficult problems such as higher-level reasoning and planning. According to one well-known story, in 1966, Marvin Minsky at MIT asked his undergraduate student Gerald Jay Sussman to “spend the summer linking a camera to a computer and getting the computer to describe what it saw” (Boden 2006, p. 781).⁵ We now know that the problem is slightly more difficult than that.⁶

What distinguished computer vision from the already existing field of digital image processing (Rosenfeld and Pfaltz 1966; Rosenfeld and Kak 1976) was a desire to recover the three-dimensional structure of the world from images and to use this as a stepping stone towards full scene understanding. Winston (1975) and Hanson and Riseman (1978) provide two nice collections of classic papers from this early period.

Early attempts at scene understanding involved extracting edges and then inferring the

⁵Boden (2006) cites (Crevier 1993) as the original source. The actual Vision Memo was authored by Seymour Papert (1966) and involved a whole cohort of students.

⁶To see how far robotic vision has come in the last six decades, have a look at some of the videos on the Boston Dynamics <https://www.bostondynamics.com>, Skydio <https://www.skydio.com>, and Covariant <https://covariant.ai> websites.

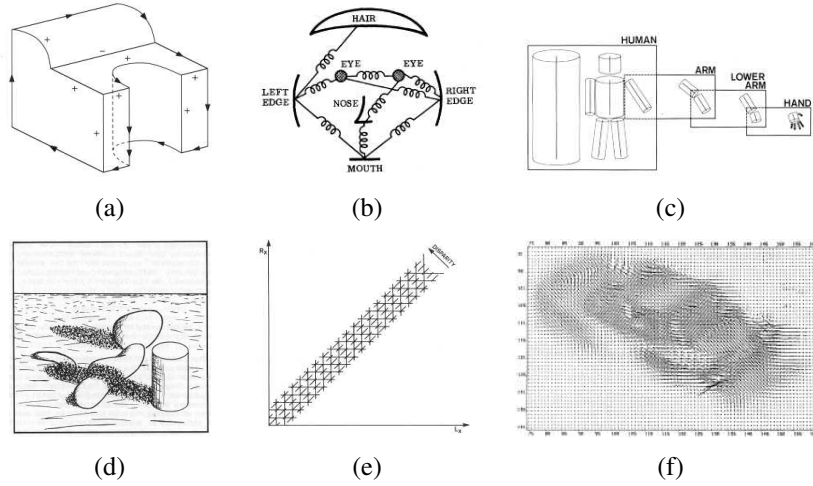


Figure 1.7 Some early (1970s) examples of computer vision algorithms: (a) line labeling (Nalwa 1993) © 1993 Addison-Wesley, (b) pictorial structures (Fischler and Elschlager 1973) © 1973 IEEE, (c) articulated body model (Marr 1982) © 1982 David Marr, (d) intrinsic images (Barrow and Tenenbaum 1981) © 1973 IEEE, (e) stereo correspondence (Marr 1982) © 1982 David Marr, (f) optical flow (Nagel and Enkelmann 1986) © 1986 IEEE.

3D structure of an object or a “blocks world” from the topological structure of the 2D lines (Roberts 1965). Several *line labeling* algorithms (Figure 1.7a) were developed at that time (Huffman 1971; Clowes 1971; Waltz 1975; Rosenfeld, Hummel, and Zucker 1976; Kanade 1980). Nalwa (1993) gives a nice review of this area. The topic of edge detection was also an active area of research; a nice survey of contemporaneous work can be found in (Davis 1975).

Three-dimensional modeling of non-polyhedral objects was also being studied (Baumgart 1974; Baker 1977). One popular approach used *generalized cylinders*, i.e., solids of revolution and swept closed curves (Agin and Binford 1976; Nevatia and Binford 1977), often arranged into parts relationships⁷ (Hinton 1977; Marr 1982) (Figure 1.7c). Fischler and Elschlager (1973) called such *elastic* arrangements of parts *pictorial structures* (Figure 1.7b).

A qualitative approach to understanding intensities and shading variations and explaining them by the effects of image formation phenomena, such as surface orientation and shadows, was championed by Barrow and Tenenbaum (1981) in their paper on *intrinsic images* (Figure 1.7d), along with the related $2\frac{1}{2}$ -D *sketch* ideas of Marr (1982). This approach has seen

⁷In robotics and computer animation, these linked-part graphs are often called *kinematic chains*.

periodic revivals, e.g., in the work of Tappen, Freeman, and Adelson (2005) and Barron and Malik (2012).

More quantitative approaches to computer vision were also developed at the time, including the first of many feature-based stereo correspondence algorithms (Figure 1.7e) (Dev 1974; Marr and Poggio 1976, 1979; Barnard and Fischler 1982; Ohta and Kanade 1985; Grimson 1985; Pollard, Mayhew, and Frisby 1985) and intensity-based optical flow algorithms (Figure 1.7f) (Horn and Schunck 1981; Huang 1981; Lucas and Kanade 1981; Nagel 1986). The early work in simultaneously recovering 3D structure and camera motion (see Chapter 11) also began around this time (Ullman 1979; Longuet-Higgins 1981).

A lot of the philosophy of how vision was believed to work at the time is summarized in David Marr's (1982) book.⁸ In particular, Marr introduced his notion of the three levels of description of a (visual) information processing system. These three levels, very loosely paraphrased according to my own interpretation, are:

- **Computational theory:** What is the goal of the computation (task) and what are the constraints that are known or can be brought to bear on the problem?
- **Representations and algorithms:** How are the input, output, and intermediate information represented and which algorithms are used to calculate the desired result?
- **Hardware implementation:** How are the representations and algorithms mapped onto actual hardware, e.g., a biological vision system or a specialized piece of silicon? Conversely, how can hardware constraints be used to guide the choice of representation and algorithm? With the prevalent use of graphics chips (GPUs) and many-core architectures for computer vision, this question is again quite relevant.

As I mentioned earlier in this introduction, it is my conviction that a careful analysis of the problem specification and known constraints from image formation and priors (the scientific and statistical approaches) must be married with efficient and robust algorithms (the engineering approach) to design successful vision algorithms. Thus, it seems that Marr's philosophy is as good a guide to framing and solving problems in our field today as it was 25 years ago.

1980s. In the 1980s, a lot of attention was focused on more sophisticated mathematical techniques for performing quantitative image and scene analysis.

Image pyramids (see Section 3.5) started being widely used to perform tasks such as image blending (Figure 1.8a) and coarse-to-fine correspondence search (Rosenfeld 1980; Burt

⁸More recent developments in visual perception theory are covered in (Wandell 1995; Palmer 1999; Livingstone 2008; Frisby and Stone 2010).

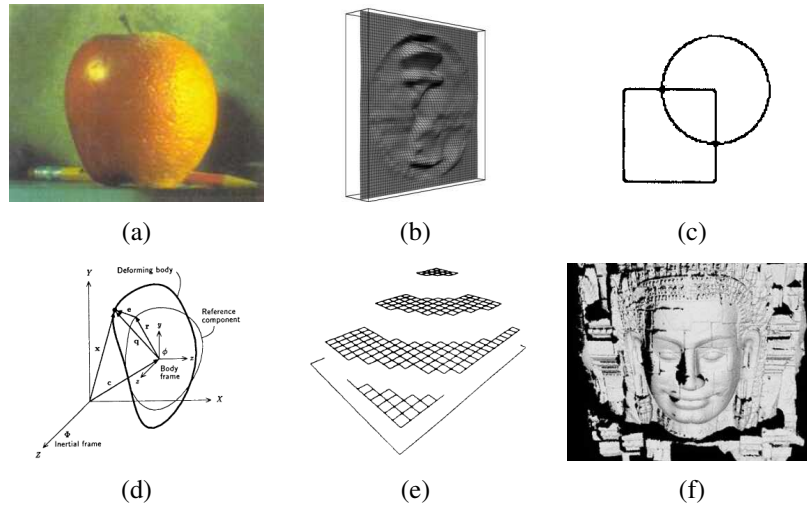


Figure 1.8 Examples of computer vision algorithms from the 1980s: (a) pyramid blending (Burt and Adelson 1983b) © 1983 ACM, (b) shape from shading (Freeman and Adelson 1991) © 1991 IEEE, (c) edge detection (Freeman and Adelson 1991) © 1991 IEEE, (d) physically based models (Terzopoulos and Witkin 1988) © 1988 IEEE, (e) regularization-based surface reconstruction (Terzopoulos 1988) © 1988 IEEE, (f) range data acquisition and merging (Banno, Masuda et al. 2008) © 2008 Springer.

and Adelson 1983b; Rosenfeld 1984; Quam 1984; Anandan 1989). Continuous versions of pyramids using the concept of *scale-space* processing were also developed (Witkin 1983; Witkin, Terzopoulos, and Kass 1986; Lindeberg 1990). In the late 1980s, wavelets (see Section 3.5.4) started displacing or augmenting regular image pyramids in some applications (Mallat 1989; Simoncelli and Adelson 1990a; Simoncelli, Freeman *et al.* 1992).

The use of stereo as a quantitative shape cue was extended by a wide variety of *shape-from-X* techniques, including shape from shading (Figure 1.8b) (see Section 13.1.1 and Horn 1975; Pentland 1984; Blake, Zisserman, and Knowles 1985; Horn and Brooks 1986, 1989), photometric stereo (see Section 13.1.1 and Woodham 1981), shape from texture (see Section 13.1.2 and Witkin 1981; Pentland 1984; Malik and Rosenholtz 1997), and shape from focus (see Section 13.1.3 and Nayar, Watanabe, and Noguchi 1995). Horn (1986) has a nice discussion of most of these techniques.

Research into better edge and contour detection (Figure 1.8c) (see Section 7.2) was also active during this period (Canny 1986; Nalwa and Binford 1986), including the introduction of dynamically evolving contour trackers (Section 7.3.1) such as *snakes* (Kass, Witkin,

and Terzopoulos 1988), as well as three-dimensional *physically based models* (Figure 1.8d) (Terzopoulos, Witkin, and Kass 1987; Kass, Witkin, and Terzopoulos 1988; Terzopoulos and Fleischer 1988).

Researchers noticed that a lot of the stereo, *low*, shape-from-X, and edge detection algorithms could be unified, or at least described, using the same mathematical framework if they were posed as variational optimization problems and made more robust (well-posed) using regularization (Figure 1.8e) (see Section 4.2 and Terzopoulos 1983; Poggio, Torre, and Koch 1985; Terzopoulos 1986b; Blake and Zisserman 1987; Bertero, Poggio, and Torre 1988; Terzopoulos 1988). Around the same time, Geman and Geman (1984) pointed out that such problems could equally well be formulated using discrete *Markov random field* (MRF) models (see Section 4.3), which enabled the use of better (global) search and optimization algorithms, such as simulated annealing.

Online variants of MRF algorithms that modeled and updated uncertainties using the Kalman filter were introduced a little later (Dickmanns and Graefe 1988; Matthies, Kanade, and Szeliski 1989; Szeliski 1989). Attempts were also made to map both regularized and MRF algorithms onto parallel hardware (Poggio and Koch 1985; Poggio, Little *et al.* 1988; Fischler, Firschein *et al.* 1989). The book by Fischler and Firschein (1987) contains a nice collection of articles focusing on all of these topics (stereo, *low*, regularization, MRFs, and even higher-level vision).

Three-dimensional range data processing (acquisition, merging, modeling, and recognition; see Figure 1.8f) continued being actively explored during this decade (Agin and Binford 1976; Besl and Jain 1985; Faugeras and Hebert 1987; Curless and Levoy 1996). The compilation by Kanade (1987) contains a lot of the interesting papers in this area.

1990s. While a lot of the previously mentioned topics continued to be explored, a few of them became significantly more active.

A burst of activity in using projective invariants for recognition (Mundy and Zisserman 1992) evolved into a concerted effort to solve the structure from motion problem (see Chapter 11). A lot of the initial activity was directed at *projective reconstructions*, which did not require knowledge of camera calibration (Faugeras 1992; Hartley, Gupta, and Chang 1992; Hartley 1994a; Faugeras and Luong 2001; Hartley and Zisserman 2004). Simultaneously, *factorization* techniques (Section 11.4.1) were developed to solve efficiently problems for which orthographic camera approximations were applicable (Figure 1.9a) (Tomasi and Kanade 1992; Poelman and Kanade 1997; Anandan and Irani 2002) and then later extended to the perspective case (Christy and Horaud 1996; Triggs 1996). Eventually, the field started using full global optimization (see Section 11.4.2 and Taylor, Kriegman, and Anandan 1991;

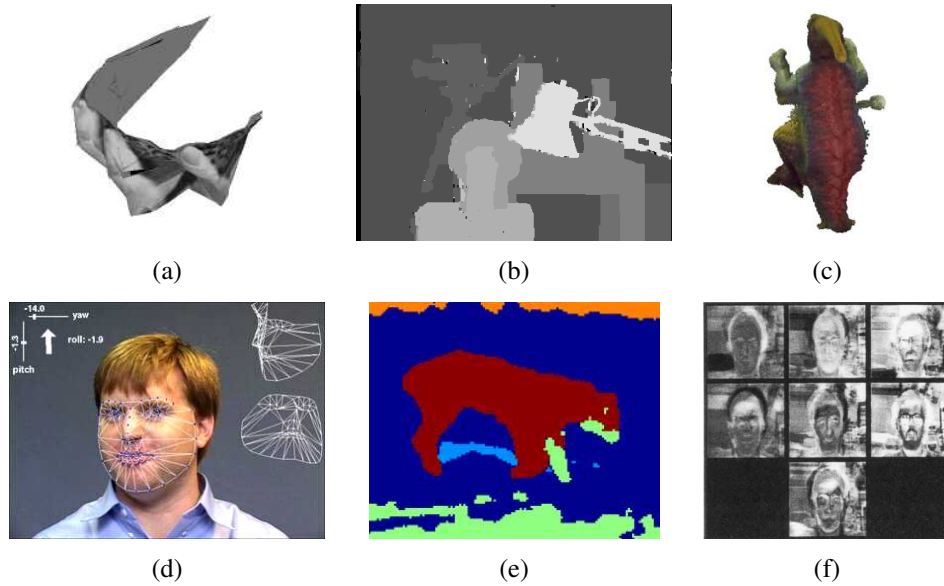


Figure 1.9 Examples of computer vision algorithms from the 1990s: (a) factorization-based structure from motion (Tomasi and Kanade 1992) © 1992 Springer, (b) dense stereo matching (Boykov, Veksler, and Zabih 2001), (c) multi-view reconstruction (Seitz and Dyer 1999) © 1999 Springer, (d) face tracking (Matthews, Xiao, and Baker 2007), (e) image segmentation (Belongie, Fowlkes et al. 2002) © 2002 Springer, (f) face recognition (Turk and Pentland 1991).

Szeliski and Kang 1994; Azarbayejani and Pentland 1995), which was later recognized as being the same as the *bundle adjustment* techniques traditionally used in photogrammetry (Triggs, McLauchlan *et al.* 1999). Fully automated 3D modeling systems were built using such techniques (Beardsley, Torr, and Zisserman 1996; Schaffalitzky and Zisserman 2002; Snavely, Seitz, and Szeliski 2006; Agarwal, Furukawa *et al.* 2011; Frahm, Fite-Georgel *et al.* 2010).

Work begun in the 1980s on using detailed measurements of color and intensity combined with accurate physical models of radiance transport and color image formation created its own subfield known as *physics-based vision*. A good survey of the field can be found in the three-volume collection on this topic (Wolff, Shafer, and Healey 1992a; Healey and Shafer 1992; Shafer, Healey, and Wolff 1992).

Optical flow methods (see Chapter 9) continued to be improved (Nagel and Enkelmann 1986; Bolles, Baker, and Marimont 1987; Horn and Weldon Jr. 1988; Anandan 1989; Bergen,

Anandan *et al.* 1992; Black and Anandan 1996; Bruhn, Weickert, and Schnörr 2005; Papenberg, Bruhn *et al.* 2006), with (Nagel 1986; Barron, Fleet, and Beauchemin 1994; Baker, Scharstein *et al.* 2011) being good surveys. Similarly, a lot of progress was made on dense stereo correspondence algorithms (see Chapter 12, Okutomi and Kanade (1993, 1994); Boykov, Veksler, and Zabih (1998); Birchfield and Tomasi (1999); Boykov, Veksler, and Zabih (2001), and the survey and comparison in Scharstein and Szeliski (2002)), with the biggest breakthrough being perhaps global optimization using *graph cut* techniques (Figure 1.9b) (Boykov, Veksler, and Zabih 2001).

Multi-view stereo algorithms (Figure 1.9c) that produce complete 3D surfaces (see Section 12.7) were also an active topic of research (Seitz and Dyer 1999; Kutulakos and Seitz 2000) that continues to be active today (Seitz, Curless *et al.* 2006; Schöps, Schönberger *et al.* 2017; Knapitsch, Park *et al.* 2017). Techniques for producing 3D volumetric descriptions from binary silhouettes (see Section 12.7.3) continued to be developed (Potmesil 1987; Srinivasan, Liang, and Hackwood 1990; Szeliski 1993; Laurentini 1994), along with techniques based on tracking and reconstructing smooth occluding contours (see Section 12.2.1 and Cipolla and Blake 1992; Vaillant and Faugeras 1992; Zheng 1994; Boyer and Berger 1997; Szeliski and Weiss 1998; Cipolla and Giblin 2000).

Tracking algorithms also improved a lot, including contour tracking using *active contours* (see Section 7.3), such as *snakes* (Kass, Witkin, and Terzopoulos 1988), *particle filters* (Blake and Isard 1998), and *level sets* (Malladi, Sethian, and Vemuri 1995), as well as intensity-based (*direct*) techniques (Lucas and Kanade 1981; Shi and Tomasi 1994; Rehg and Kanade 1994), often applied to tracking faces (Figure 1.9d) (Lanitis, Taylor, and Cootes 1997; Matthews and Baker 2004; Matthews, Xiao, and Baker 2007) and whole bodies (Sidenbladh, Black, and Fleet 2000; Hilton, Fua, and Ronfard 2006; Moeslund, Hilton, and Krüger 2006).

Image segmentation (see Section 7.5) (Figure 1.9e), a topic which has been active since the earliest days of computer vision (Brice and Fennema 1970; Horowitz and Pavlidis 1976; Riseman and Arbib 1977; Rosenfeld and Davis 1979; Haralick and Shapiro 1985; Pavlidis and Liow 1990), was also an active topic of research, producing techniques based on minimum energy (Mumford and Shah 1989) and minimum description length (Leclerc 1989), *normalized cuts* (Shi and Malik 2000), and *mean shift* (Comaniciu and Meer 2002).

Statistical learning techniques started appearing, first in the application of principal component *eigenface* analysis to face recognition (Figure 1.9f) (see Section 5.2.3 and Turk and Pentland 1991) and linear dynamical systems for curve tracking (see Section 7.3.1 and Blake and Isard 1998).

Perhaps the most notable development in computer vision during this decade was the increased interaction with computer graphics (Seitz and Szeliski 1999), especially in the

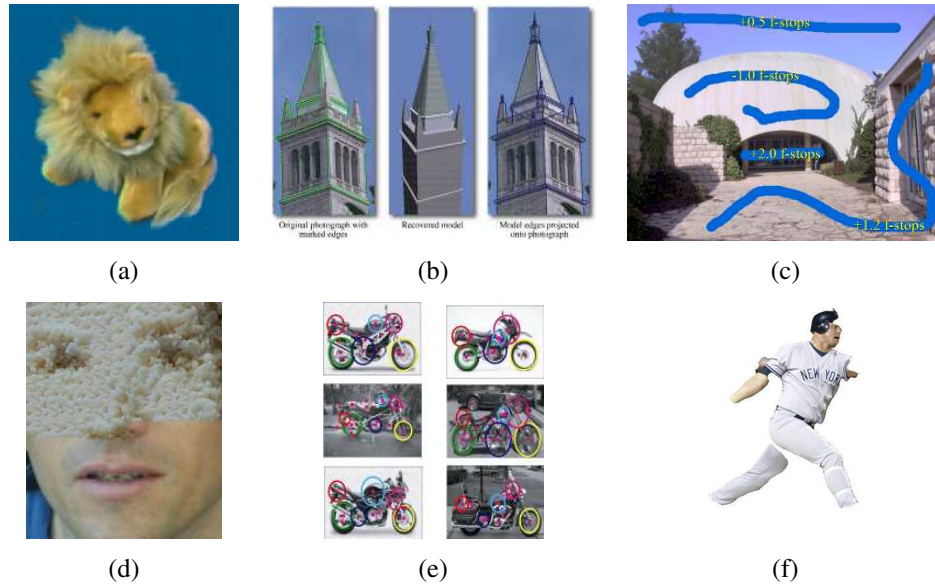


Figure 1.10 Examples of computer vision algorithms from the 2000s: (a) image-based rendering (Gortler, Grzeszczuk et al. 1996), (b) image-based modeling (Debevec, Taylor, and Malik 1996) © 1996 ACM, (c) interactive tone mapping (Lischinski, Farbman et al. 2006) (d) texture synthesis (Efros and Freeman 2001), (e) feature-based recognition (Fergus, Perona, and Zisserman 2007), (f) region-based recognition (Mori, Ren et al. 2004) © 2004 IEEE.

cross-disciplinary area of *image-based modeling and rendering* (see Chapter 14). The idea of manipulating real-world imagery directly to create new animations first came to prominence with *image morphing* techniques (Figure 1.5c) (see Section 3.6.3 and Beier and Neely 1992) and was later applied to *view interpolation* (Chen and Williams 1993; Seitz and Dyer 1996), panoramic image stitching (Figure 1.5a) (see Section 8.2 and Mann and Picard 1994; Chen 1995; Szeliski 1996; Szeliski and Shum 1997; Szeliski 2006a), and full light-field rendering (Figure 1.10a) (see Section 14.3 and Gortler, Grzeszczuk et al. 1996; Levoy and Hanrahan 1996; Shade, Gortler et al. 1998). At the same time, image-based modeling techniques (Figure 1.10b) for automatically creating realistic 3D models from collections of images were also being introduced (Beardsley, Torr, and Zisserman 1996; Debevec, Taylor, and Malik 1996; Taylor, Debevec, and Malik 1996).

2000s. This decade continued to deepen the interplay between the vision and graphics fields, but more importantly embraced data-driven and learning approaches as core compo-

nents of vision. Many of the topics introduced under the rubric of image-based rendering, such as image stitching (see Section 8.2), light-field capture and rendering (see Section 14.3), and *high dynamic range* (HDR) image capture through exposure bracketing (Figure 1.5b) (see Section 10.2 and Mann and Picard 1995; Debevec and Malik 1997), were re-christened as *computational photography* (see Chapter 10) to acknowledge the increased use of such techniques in everyday digital photography. For example, the rapid adoption of exposure bracketing to create high dynamic range images necessitated the development of *tone mapping* algorithms (Figure 1.10c) (see Section 10.2.1) to convert such images back to displayable results (Fattal, Lischinski, and Werman 2002; Durand and Dorsey 2002; Reinhard, Stark *et al.* 2002; Lischinski, Farbman *et al.* 2006). In addition to merging multiple exposures, techniques were developed to merge *flash* images with non-*flash* counterparts (Eisemann and Durand 2004; Petschnigg, Agrawala *et al.* 2004) and to interactively or automatically select different regions from overlapping images (Agarwala, Dontcheva *et al.* 2004).

Texture synthesis (Figure 1.10d) (see Section 10.5), quilting (Efros and Leung 1999; Efros and Freeman 2001; Kwatra, Schödl *et al.* 2003), and inpainting (Bertalmio, Sapiro *et al.* 2000; Bertalmio, Vese *et al.* 2003; Criminisi, Pérez, and Toyama 2004) are additional topics that can be classified as computational photography techniques, since they re-combine input image samples to produce new photographs.

A second notable trend during this decade was the emergence of feature-based techniques (combined with learning) for object recognition (see Section 6.1 and Ponce, Hebert *et al.* 2006). Some of the notable papers in this area include the *constellation model* of Fergus, Perona, and Zisserman (2007) (Figure 1.10e) and the *pictorial structures* of Felzenszwalb and Huttenlocher (2005). Feature-based techniques also dominate other recognition tasks, such as scene recognition (Zhang, Marszalek *et al.* 2007) and panorama and location recognition (Brown and Lowe 2007; Schindler, Brown, and Szeliski 2007). And while *interest point* (patch-based) features tend to dominate current research, some groups are pursuing recognition based on contours (Belongie, Malik, and Puzicha 2002) and region segmentation (Figure 1.10f) (Mori, Ren *et al.* 2004).

Another significant trend from this decade was the development of more efficient algorithms for complex global optimization problems (see Chapter 4 and Appendix B.5 and Szeliski, Zabih *et al.* 2008; Blake, Kohli, and Rother 2011). While this trend began with work on graph cuts (Boykov, Veksler, and Zabih 2001; Kohli and Torr 2007), a lot of progress has also been made in message passing algorithms, such as *loopy belief propagation* (LBP) (Yedidia, Freeman, and Weiss 2001; Kumar and Torr 2006).

The most notable trend from this decade, which has by now completely taken over visual recognition and most other aspects of computer vision, was the application of sophisticated

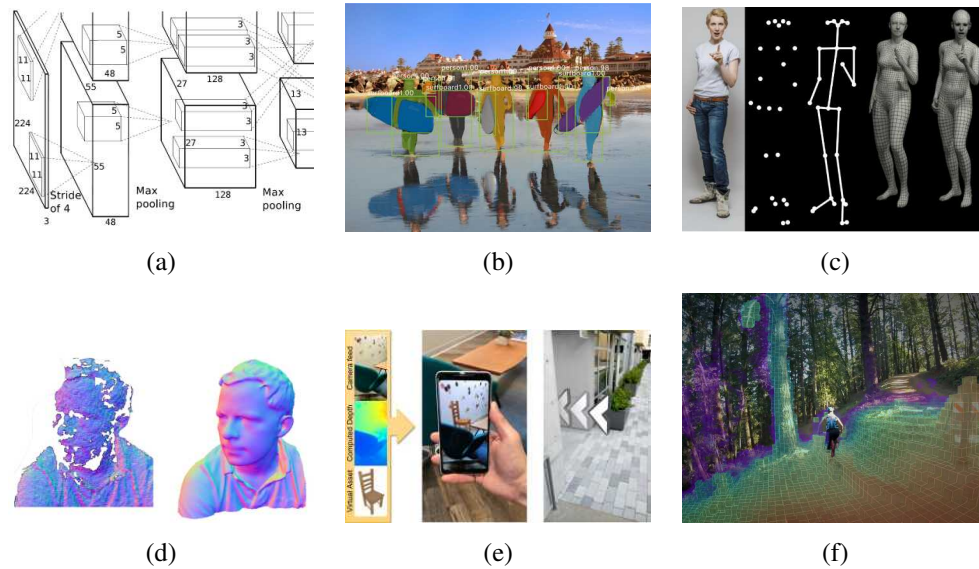


Figure 1.11 Examples of computer vision algorithms from the 2010s: (a) the SuperVision deep neural network © Krizhevsky, Sutskever, and Hinton (2012); (b) object instance segmentation (He, Gkioxari et al. 2017) © 2017 IEEE; (c) whole body, expression, and gesture fitting from a single image (Pavlakos, Choutas et al. 2019) © 2019 IEEE; (d) fusing multiple color depth images using the KinectFusion real-time system (Newcombe, Izadi et al. 2011) © 2011 IEEE; (e) smartphone augmented reality with real-time depth occlusion effects (Valentin, Kowdle et al. 2018) © 2018 ACM; (f) 3D map computed in real-time on a fully autonomous Skydio R1 drone (Cross 2019).

machine learning techniques to computer vision problems (see Chapters 5 and 6). This trend coincided with the increased availability of immense quantities of partially labeled data on the internet, as well as significant increases in computational power, which makes it more feasible to learn object categories without the use of careful human supervision.

2010s. The trend towards using large labeled (and also self-supervised) datasets to develop machine learning algorithms became a tidal wave that totally revolutionized the development of image recognition algorithms as well as other applications, such as denoising and optical flow, which previously used Bayesian and global optimization techniques.

This trend was enabled by the development of high-quality large-scale annotated datasets such as ImageNet (Deng, Dong et al. 2009; Russakovsky, Deng et al. 2015), Microsoft COCO (Common Objects in Context) (Lin, Maire et al. 2014), and LVIS (Gupta, Dollár, and Gir-

shick 2019). These datasets provided not only reliable metrics for tracking the progress of recognition and semantic segmentation algorithms, but more importantly, sufficient labeled data to develop complete solutions based on machine learning.

Another major trend was the dramatic increase in computational power available from the development of general purpose (data-parallel) algorithms on graphical processing units (GPU). The breakthrough SuperVision (“AlexNet”) deep neural network (Figure 1.11a; Krizhevsky, Sutskever, and Hinton 2012), which was the first neural network to win the yearly ImageNet large-scale visual recognition challenge, relied on GPU training, as well as a number of technical advances, for its dramatic performance. After the publication of this paper, progress in using deep convolutional architectures accelerated dramatically, to the point where they are now the only architecture considered for recognition and semantic segmentation tasks (Figure 1.11b), as well as the preferred architecture for many other vision tasks (Chapter 5; LeCun, Bengio, and Hinton 2015), including optical flow (Sun, Yang *et al.* 2018)), denoising, and monocular depth inference (Li, Dekel *et al.* 2019).

Large datasets and GPU architectures, coupled with the rapid dissemination of ideas through timely publications on arXiv as well as the development of languages for deep learning and the open sourcing of neural network models, all contributed to an explosive growth in this area, both in rapid advances and capabilities, and also in the sheer number of publications and researchers now working on these topics. They also enabled the extension of image recognition approaches to video understanding tasks such as action recognition (Feichtenhofer, Fan *et al.* 2019), as well as structured regression tasks such as real-time multi-person body pose estimation (Cao, Simon *et al.* 2017).

Specialized sensors and hardware for computer vision tasks also continued to advance. The Microsoft Kinect depth camera, released in 2010, quickly became an essential component of many 3D modeling (Figure 1.11d) and person tracking (Shotton, Fitzgibbon *et al.* 2011) systems. Over the decade, 3D body shape modeling and tracking systems continued to evolve, to the point where it is now possible to infer a person’s 3D model with gestures and expression from a single image (Figure 1.11c).

And while depth sensors have not yet become ubiquitous (except for security applications on high-end phones), computational photography algorithms run on all of today’s smartphones. Innovations introduced in the computer vision community, such as panoramic image stitching and bracketed high dynamic range image merging, are now standard features, and multi-image low-light denoising algorithms are also becoming commonplace (Liba, Murthy *et al.* 2019). Lightfield imaging algorithms, which allow the creation of soft depth-of-field effects, are now also becoming more available (Garg, Wadhwa *et al.* 2019). Finally, mobile augmented reality applications that perform real-time pose estimation and environment

augmentation using combinations of feature tracking and inertial measurements are commonplace, and are currently being extended to include pixel-accurate depth occlusion effects (Figure 1.11e).

On higher-end platforms such as autonomous vehicles and drones, powerful real-time SLAM (simultaneous localization and mapping) and VIO (visual inertial odometry) algorithms (Engel, Schöps, and Cremers 2014; Forster, Zhang *et al.* 2017; Engel, Koltun, and Cremers 2018) can build accurate 3D maps that enable, e.g., autonomous flight through challenging scenes such as forests (Figure 1.11f).

In summary, this past decade has seen incredible advances in the performance and reliability of computer vision algorithms, brought in part by the shift to machine learning and training on very large sets of real-world data. It has also seen the application of vision algorithms in myriad commercial and consumer scenarios as well as new challenges engendered by their widespread use (Su and Crandall 2021).

1.3 Book overview

In the final part of this introduction, I give a brief tour of the material in this book, as well as a few notes on notation and some additional general references. Since computer vision is such a broad field, it is possible to study certain aspects of it, e.g., geometric image formation and 3D structure recovery, without requiring other parts, e.g., the modeling of reflectance and shading. Some of the chapters in this book are only loosely coupled with others, and it is not strictly necessary to read all of the material in sequence.

Figure 1.12 shows a rough layout of the contents of this book. Since computer vision involves going from images to both a semantic understanding as well as a 3D structural description of the scene, I have positioned the chapters horizontally in terms of where in this spectrum they land, in addition to vertically according to their dependence.⁹

Interspersed throughout the book are sample **applications**, which relate the algorithms and mathematical material being presented in various chapters to useful, real-world applications. Many of these applications are also presented in the exercises sections, so that students can write their own.

At the end of each section, I provide a set of **exercises** that the students can use to implement, test, and refine the algorithms and techniques presented in each section. Some of the exercises are suitable as written homework assignments, others as shorter one-week projects,

⁹For an interesting comparison with what is known about the human visual system, e.g., the largely parallel *what* and *where* pathways (Goodale and Milner 1992), see some textbooks on human perception (Palmer 1999; Livingstone 2008; Frisby and Stone 2010).

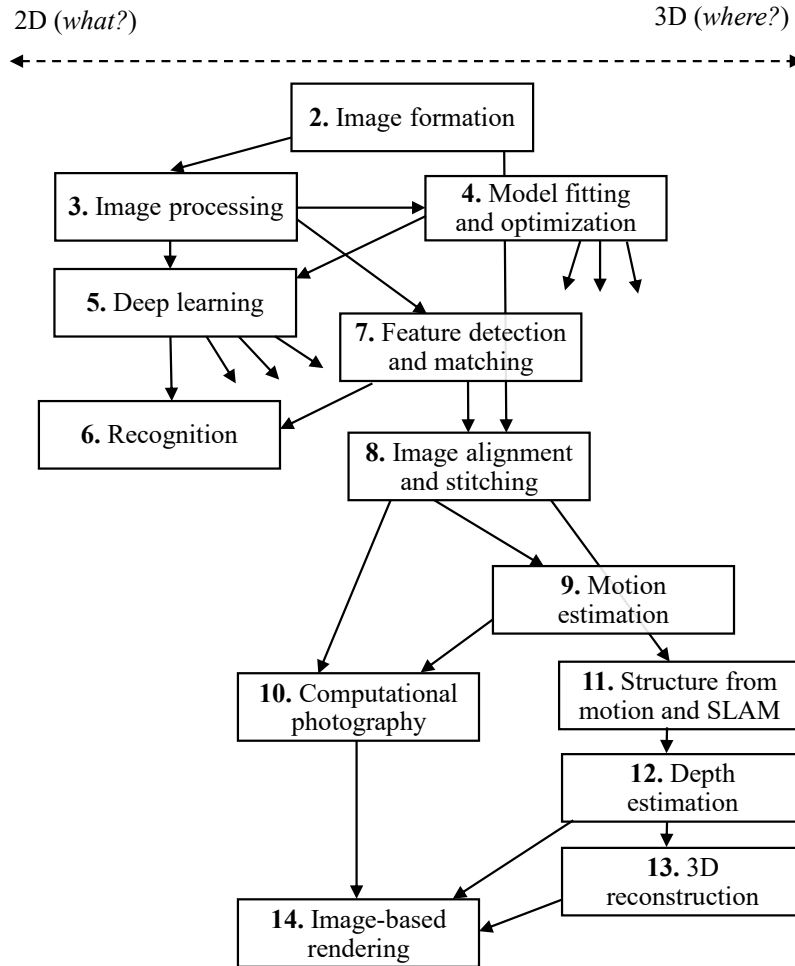


Figure 1.12 A taxonomy of the topics covered in this book, showing the (rough) dependencies between different chapters, which are roughly positioned along the left–right axis depending on whether they are more closely related to images (left) or 3D geometry (right) representations. The “what–where” along the top axis is a reference to separate visual pathways in the visual system (*Goodale and Milner 1992*), but should not be taken too seriously. Foundational techniques such as optimization and deep learning are widely used in subsequent chapters.

and still others as open-ended research problems that make for challenging final projects. Motivated students who implement a reasonable subset of these exercises will, by the end of the book, have a computer vision software library that can be used for a variety of interesting tasks and projects.

If the students or curriculum do not have a strong preference for programming languages, Python, with the NumPy scientific and array arithmetic library plus the OpenCV vision library, are a good environment to develop algorithms and learn about vision. Not only will the students learn how to program using array/tensor notation and linear/matrix algebra (which is a good foundation for later use of PyTorch for deep learning), you can also prepare classroom assignments using Jupyter notebooks, giving you the option to combine descriptive tutorials, sample code, and code to be extended/modified in one convenient location.¹⁰

As this is a reference book, I try wherever possible to discuss which techniques and algorithms work well in practice, as well as provide up-to-date pointers to the latest research results in the areas that I cover. The exercises can be used to build up your own personal library of self-tested and validated vision algorithms, which is more worthwhile in the long term (assuming you have the time) than simply pulling algorithms out of a library whose performance you do not really understand.

The book begins in Chapter 2 with a review of the image formation processes that create the images that we see and capture. Understanding this process is fundamental if you want to take a scientific (model-based) approach to computer vision. Students who are eager to just start implementing algorithms (or courses that have limited time) can skip ahead to the next chapter and dip into this material later. In Chapter 2, we break down image formation into three major components. Geometric image formation (Section 2.1) deals with points, lines, and planes, and how these are mapped onto images using *projective geometry* and other models (including radial lens distortion). Photometric image formation (Section 2.2) covers *radiometry*, which describes how light interacts with surfaces in the world, and *optics*, which projects light onto the sensor plane. Finally, Section 2.3 covers how sensors work, including topics such as sampling and aliasing, color sensing, and in-camera compression.

Chapter 3 covers image processing, which is needed in almost all computer vision applications. This includes topics such as linear and non-linear filtering (Section 3.3), the Fourier transform (Section 3.4), image pyramids and wavelets (Section 3.5), and geometric transformations such as image warping (Section 3.6). Chapter 3 also presents applications such as seamless image blending and image morphing.

Chapter 4 begins with a new section on data fitting and interpolation, which provides a

¹⁰You may also be able to run your notebooks and train your models using the Google Colab service at <https://colab.research.google.com>.

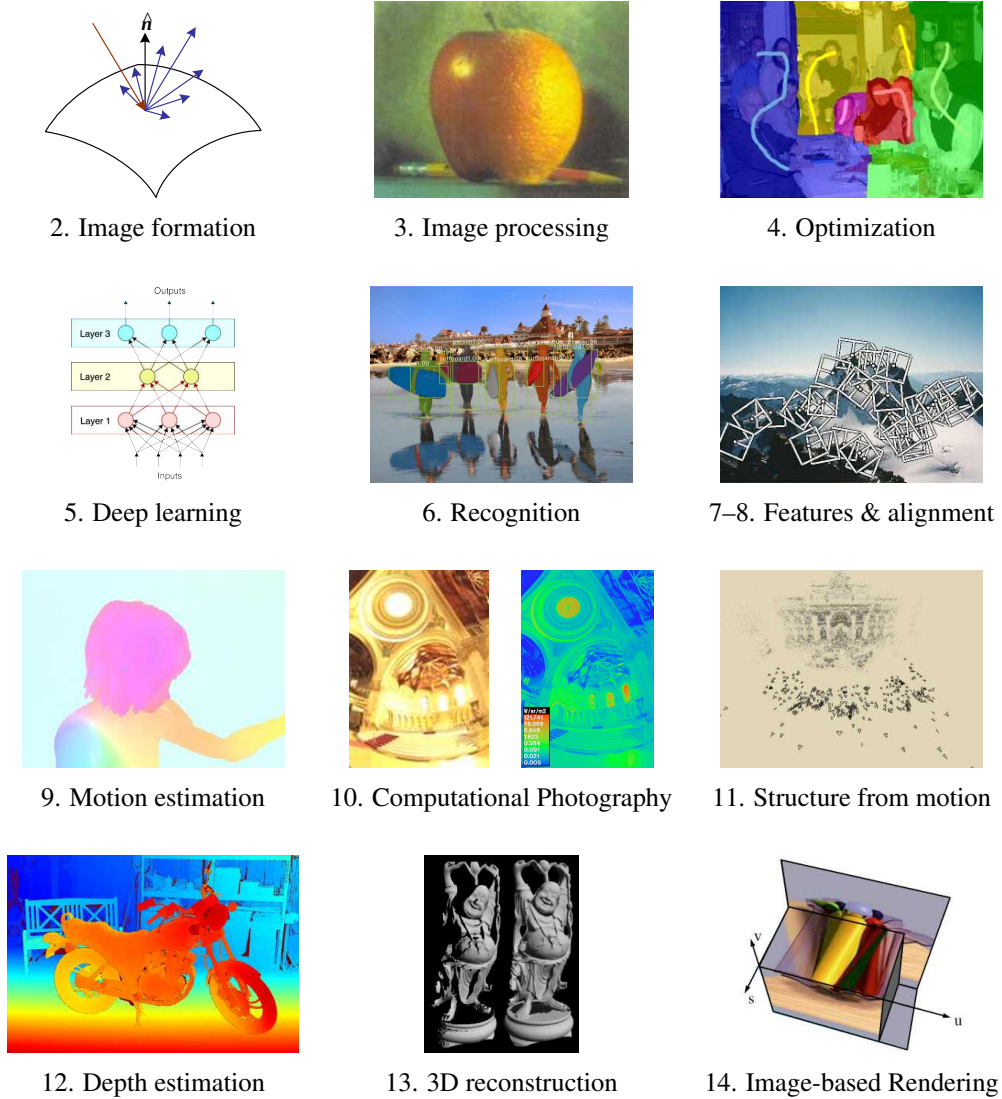


Figure 1.13 A pictorial summary of the chapter contents. Sources: Burt and Adelson (1983b); Agarwala, Dontcheva et al. (2004); Glassner (2018); He, Gkioxari et al. (2017); Brown, Szeliski, and Winder (2005); Butler, Wulff et al. (2012); Debevec and Malik (1997); Snavely, Seitz, and Szeliski (2006); Scharstein, Hirschmüller et al. (2014); Curless and Levoy (1996); Gortler, Grzeszczuk et al. (1996)—see the figures in the respective chapters for copyright information.

conceptual framework for global optimization techniques such as *regularization* and *Markov random fields* (MRFs), as well as *machine learning*, which we cover in the next chapter. Section 4.2 covers classic regularization techniques, i.e., piecewise-continuous smoothing splines (aka *variational techniques*) implemented using fast iterated linear system solvers, which are still often the method of choice in time-critical applications such as mobile augmented reality. The next section (4.3) presents the related topic of *MRFs*, which also serve as an introduction to Bayesian inference techniques, covered at a more abstract level in Appendix B. The chapter also discusses applications to interactive colorization and segmentation.

Chapter 5 is a completely new chapter covering machine learning, deep learning, and deep neural networks. It begins in Section 5.1 with a review of classic *supervised machine learning* approaches, which are designed to classify images (or regress values) based on intermediate-level features. Section 5.2 looks at *unsupervised learning*, which is useful for both understanding unlabeled training data and providing models of real-world distributions. Section 5.3 presents the basic elements of feedforward neural networks, including weights, layers, and activation functions, as well as methods for network training. Section 5.4 goes into more detail on convolutional networks and their applications to both recognition and image processing. The last section in the chapter discusses more complex networks, including 3D, spatio-temporal, recurrent, and generative networks.

Chapter 6 covers the topic of *recognition*. In the first edition of this book this chapter came last, since it built upon earlier methods such as segmentation and feature matching. With the advent of deep networks, many of these intermediate representations are no longer necessary, since the network can learn them as part of the training process. As so much of computer vision research is now devoted to various recognition topics, I decided to move this chapter up so that students can learn about it earlier in the course.

The chapter begins with the classic problem of *instance recognition*, i.e., finding instances of known 3D objects in cluttered scenes. Section 6.2 covers both traditional and deep network approaches to whole *image classification*, i.e., what used to be called *category recognition*. It also discusses the special case of facial recognition. Section 6.3 presents algorithms for *object detection* (drawing bounding boxes around recognized objects), with a brief review of older approaches to face and pedestrian detection. Section 6.4 covers various flavors of *semantic segmentation* (generating per-pixel labels), including *instance segmentation* (delineating separate objects), *pose estimation* (labeling pixels with body parts), and *panoptic segmentation* (labeling both things and stuff). In Section 6.5, we briefly look at some recent papers in *video understanding* and *action recognition*, while in Section 6.6 we mention some recent work in image captioning and visual question answering.

In Chapter 7, we cover feature detection and matching. A lot of current 3D reconstruction

and recognition techniques are built on extracting and matching *feature points* (Section 7.1), so this is a fundamental technique required by many subsequent chapters (Chapters 8 and 11) and even in instance recognition (Section 6.1). We also cover edge and straight line detection in Sections 7.2 and 7.4, contour tracking in Section 7.3, and low-level segmentation techniques in Section 7.5.

Feature detection and matching are used in Chapter 8 to perform *image alignment* (or *registration*) and *image stitching*. We introduce the basic techniques of feature-based alignment and show how this problem can be solved using either linear or non-linear least squares, depending on the motion involved. We also introduce additional concepts, such as uncertainty weighting and robust regression, which are essential to making real-world systems work. Feature-based alignment is then used as a building block for both 2D applications such as image stitching (Section 8.2) and computational photography (Chapter 10), as well as 3D geometric alignment tasks such as pose estimation and structure from motion (Chapter 11).

The second part of Chapter 8 is devoted to *image stitching*, i.e., the construction of large panoramas and composites. While stitching is just one example of *computational photography* (see Chapter 10), there is enough depth here to warrant a separate section. We start by discussing various possible motion models (Section 8.2.1), including planar motion and pure camera rotation. We then discuss global alignment (Section 8.3), which is a special (simplified) case of general bundle adjustment, and then present *panorama recognition*, i.e., techniques for automatically discovering which images actually form overlapping panoramas. Finally, we cover the topics of *image compositing* and *blending* (Section 8.4), which involve both selecting which pixels from which images to use and blending them together so as to disguise exposure differences.

Image stitching is a wonderful application that ties together most of the material covered in earlier parts of this book. It also makes for a good mid-term course project that can build on previously developed techniques such as image warping and feature detection and matching. Sections 8.2–8.4 also present more specialized variants of stitching such as whiteboard and document scanning, video summarization, *panography*, full 360° spherical panoramas, and interactive photomontage for blending repeated action shots together.

In Chapter 9, we generalize the concept of feature-based image alignment to cover dense intensity-based motion estimation, i.e., *optical flow*. We start with the simplest possible motion models, translational motion (Section 9.1), and cover topics such as hierarchical (coarse-to-fine) motion estimation, Fourier-based techniques, and iterative refinement. We then present parametric motion models, which can be used to compensate for camera rotation and zooming, as well as affine or planar perspective motion (Section 9.2). This is then generalized to spline-based motion models (Section 9.2.2) and finally to general per-pixel

optical flow (Section 9.3). We close the chapter in Section 9.4 with a discussion of layered and learned motion models as well as video object segmentation and tracking. Applications of motion estimation techniques include automated morphing, video denoising, and frame interpolation (slow motion).

Chapter 10 presents additional examples of *computational photography*, which is the process of creating new images from one or more input photographs, often based on the careful modeling and calibration of the image formation process (Section 10.1). Computational photography techniques include merging multiple exposures to create *high dynamic range* images (Section 10.2), increasing image resolution through blur removal and *super-resolution* (Section 10.3), and image editing and compositing operations (Section 10.4). We also cover the topics of texture analysis, synthesis, and *inpainting* (hole filling) in Section 10.5, as well as non-photorealistic rendering and style transfer.

Starting in Chapter 11, we delve more deeply into techniques for reconstructing 3D models from images. We begin by introducing methods for *intrinsic* camera calibration in Section 11.1 and *3D pose estimation*, i.e., *extrinsic* calibration, in Section 11.2. These sections also describe the applications of single-view reconstruction of building models and *3D location recognition*. We then cover the topic of *triangulation* (Section 11.2.4), which is the 3D reconstruction of points from matched features when the camera positions are known.

Chapter 11 then moves on to the topic of *structure from motion*, which involves the simultaneous recovery of 3D camera motion and 3D scene structure from a collection of tracked 2D features. We begin with two-frame structure from motion (Section 11.3), for which algebraic techniques exist, as well as robust sampling techniques such as RANSAC that can discount erroneous feature matches. We then cover techniques for multi-frame structure from motion, including factorization (Section 11.4.1), bundle adjustment (Section 11.4.2), and constrained motion and structure models (Section 11.4.8). We present applications in visual effects (*match move*) and sparse 3D model construction for large (e.g., internet) photo collections. The final part of this chapter (Section 11.5) has a new section on *simultaneous localization and mapping* (SLAM) as well as its applications to autonomous navigation and mobile augmented reality (AR).

In Chapter 12, we turn to the topic of stereo correspondence, which can be thought of as a special case of motion estimation where the camera positions are already known (Section 12.1). This additional knowledge enables stereo algorithms to search over a much smaller space of correspondences to produce dense depth estimates using various combinations of matching criteria, optimization algorithm, and/or deep networks (Sections 12.3–12.6). We also cover *multi-view* stereo algorithms that build a true 3D surface representation instead of just a single depth map (Section 12.7), as well as *monocular depth inference* algorithms

that hallucinate depth maps from just a single image (Section 12.8). Applications of stereo matching include head and gaze tracking, as well as depth-based background replacement (*Z-keying*).

Chapter 13 covers additional 3D shape and appearance modeling techniques. These include classic *shape-from-X* techniques such as shape from shading, shape from texture, and shape from focus (Section 13.1). An alternative to all of these *passive* computer vision techniques is to use *active rangefinding* (Section 13.2), i.e., to project patterned light onto scenes and recover the 3D geometry through triangulation. Processing all of these 3D representations often involves interpolating or simplifying the geometry (Section 13.3), or using alternative representations such as surface point sets (Section 13.4) or implicit functions (Section 13.5).

The collection of techniques for going from one or more images to partial or full 3D models is often called *image-based modeling* or *3D photography*. Section 13.6 examines three more specialized application areas (architecture, faces, and human bodies), which can use *model-based reconstruction* to fit parameterized models to the sensed data. Section 13.7 examines the topic of *appearance modeling*, i.e., techniques for estimating the texture maps, albedos, or even sometimes complete *bi-directional reflectance distribution functions* (BRDFs) that describe the appearance of 3D surfaces.

In Chapter 14, we discuss the large number of image-based rendering techniques that have been developed in the last three decades, including simpler techniques such as view interpolation (Section 14.1), layered depth images (Section 14.2), and sprites and layers (Section 14.2.1), as well as the more general framework of light fields and Lumigraphs (Section 14.3) and higher-order fields such as environment mattes (Section 14.4). Applications of these techniques include navigating 3D collections of photographs using *photo tourism*.

Next, we discuss video-based rendering, which is the temporal extension of image-based rendering. The topics we cover include video-based animation (Section 14.5.1), periodic video turned into *video textures* (Section 14.5.2), and 3D video constructed from multiple video streams (Section 14.5.4). Applications of these techniques include animating still images and creating home tours based on 360° video. We finish the chapter with an overview of the new emerging field of *neural rendering*.

To support the book's use as a textbook, the appendices and associated website contain more detailed mathematical topics and additional material. Appendix A covers linear algebra and numerical techniques, including matrix algebra, least squares, and iterative techniques. Appendix B covers Bayesian estimation theory, including maximum likelihood estimation, robust statistics, Markov random fields, and uncertainty modeling. Appendix C describes the supplementary material that can be used to complement this book, including images and datasets, pointers to software, and course slides.

Week	Chapter	Topics
1.	Chapters 1–2	Introduction and image formation
2.	Chapter 3	Image processing
3.	Chapters 4–5	Optimization and learning
4.	Chapter 5	Deep learning
5.	Chapter 6	Recognition
6.	Chapter 7	Feature detection and matching
7.	Chapter 8	Image alignment and stitching
8.	Chapter 9	Motion estimation
9.	Chapter 10	Computational photography
10.	Chapter 11	Structure from motion
11.	Chapter 12	Depth estimation
12.	Chapter 13	3D reconstruction
13.	Chapter 14	Image-based rendering

Table 1.1 *Sample syllabus for a one semester 13-week course. A 10-week quarter could go into lesser depth or omit some topics.*

1.4 Sample syllabus

Teaching all of the material covered in this book in a single quarter or semester course is a Herculean task and likely one not worth attempting.¹¹ It is better to simply pick and choose topics related to the lecturer’s preferred emphasis and tailored to the set of mini-projects envisioned for the students.

Steve Seitz and I have successfully used a 10-week syllabus similar to the one shown in Table 1.1 as both an undergraduate and a graduate-level course in computer vision. The undergraduate course¹² tends to go lighter on the mathematics and takes more time reviewing basics, while the graduate-level course¹³ dives more deeply into techniques and assumes the students already have a decent grounding in either vision or related mathematical techniques. Related courses have also been taught on the topics of 3D photography and computational photography. Appendix C.3 and the book’s website list other courses that use this book to teach a similar curriculum.

¹¹Some universities, such as Stanford (CS231A & 231N), Berkeley (CS194-26/294-26 & 280), and the University of Michigan (EECS 498/598 & 442), now split the material over two courses.

¹²<http://www.cs.washington.edu/education/courses/455>

¹³<http://www.cs.washington.edu/education/courses/576>

When Steve and I teach the course, we prefer to give the students several small programming assignments early in the course rather than focusing on written homework or quizzes. With a suitable choice of topics, it is possible for these projects to build on each other. For example, introducing feature matching early on can be used in a second assignment to do image alignment and stitching. Alternatively, direct (optical flow) techniques can be used to do the alignment and more focus can be put on either graph cut seam selection or multi-resolution blending techniques.

In the past, we have also asked the students to propose a final project (we provide a set of suggested topics for those who need ideas) by the middle of the course and reserved the last week of the class for student presentations. Sometimes, a few of these projects have actually turned into conference submissions!

No matter how you decide to structure the course or how you choose to use this book, I encourage you to try at least a few small programming tasks to get a feel for how vision techniques work and how they fail. Better yet, pick topics that are fun and can be used on your own photographs, and try to push your creative boundaries to come up with surprising results.

1.5 A note on notation

For better or worse, the notation found in computer vision and multi-view geometry textbooks tends to vary all over the map (Faugeras 1993; Hartley and Zisserman 2004; Girod, Greiner, and Niemann 2000; Faugeras and Luong 2001; Forsyth and Ponce 2003). In this book, I use the convention I first learned in my high school physics class (and later multi-variate calculus and computer graphics courses), which is that vectors $\mathbf{v}$ are lower case bold, matrices $\mathbf{M}$ are upper case bold, and scalars (T, s) are mixed case italic. Unless otherwise noted, vectors operate as column vectors, i.e., they post-multiply matrices, $\mathbf{M}\mathbf{v}$, although they are sometimes written as comma-separated parenthesized lists $\mathbf{x} = (x, y)$ instead of bracketed column vectors $\mathbf{x} = [x \ y]^T$. Some commonly used matrices are $\mathbf{R}$ for rotations, $\mathbf{K}$ for calibration matrices, and $\mathbf{I}$ for the identity matrix. Homogeneous coordinates (Section 2.1) are denoted with a tilde over the vector, e.g., $\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{w}) = \tilde{w}(x, y, 1) = \tilde{w}\mathbf{x}$ in $\mathcal{P}^2$. The cross product operator in matrix form is denoted by $[\]_{\times}$.

1.6 Additional reading

This book attempts to be self-contained, so that students can implement the basic assignments and algorithms described here without the need for outside references. However, it does pre-

suppose a general familiarity with basic concepts in linear algebra and numerical techniques, which are reviewed in Appendix A, and image processing, which is reviewed in Chapter 3.

Students who want to delve more deeply into these topics can look in Golub and Van Loan (1996) for matrix algebra and Strang (1988) for linear algebra. In image processing, there are a number of popular textbooks, including Crane (1997), Gomes and Velho (1997), Jähne (1997), Pratt (2007), Russ (2007), Burger and Burge (2008), and Gonzalez and Woods (2017). For computer graphics, popular texts include Hughes, van Dam *et al.* (2013) and Marschner and Shirley (2015), with Glassner (1995) providing a more in-depth look at image formation and rendering. For statistics and machine learning, Chris Bishop's (2006) book is a wonderful and comprehensive introduction with a wealth of exercises, while Murphy (2012) provides a more recent take on the field and Hastie, Tibshirani, and Friedman (2009) a more classic treatment. A great introductory text to deep learning is Glassner (2018), while Goodfellow, Bengio, and Courville (2016) and Zhang, Lipton *et al.* (2021) provide more comprehensive treatments. Students may also want to look in other textbooks on computer vision for material that we do not cover here, as well as for additional project ideas (Nalwa 1993; Trucco and Verri 1998; Hartley and Zisserman 2004; Forsyth and Ponce 2011; Prince 2012; Davies 2017).

There is, however, no substitute for reading the latest research literature, both for the latest ideas and techniques and for the most up-to-date references to related literature.¹⁴ In this book, I have attempted to cite the most recent work in each field so that students can read them directly and use them as inspiration for their own work. Browsing the last few years' conference proceedings from the major vision, graphics, and machine learning conferences, such as CVPR, ECCV, ICCV, SIGGRAPH, and NeurIPS, as well as keeping an eye out for the latest publications on arXiv, will provide a wealth of new ideas. The tutorials offered at these conferences, for which slides or notes are often available online, are also an invaluable resource.

¹⁴For a comprehensive bibliography and taxonomy of computer vision research, Keith Price's Annotated Computer Vision Bibliography <https://www.visionbib.com/bibliography/contents.html> is an invaluable resource.

Chapter 2

Image formation

2.1	Geometric primitives and transformations	36
2.1.1	2D transformations	40
2.1.2	3D transformations	43
2.1.3	3D rotations	45
2.1.4	3D to 2D projections	51
2.1.5	Lens distortions	63
2.2	Photometric image formation	66
2.2.1	Lighting	66
2.2.2	Relectance and shading	67
2.2.3	Optics	74
2.3	The digital camera	79
2.3.1	Sampling and aliasing	84
2.3.2	Color	87
2.3.3	Compression	98
2.4	Additional reading	101
2.5	Exercises	102

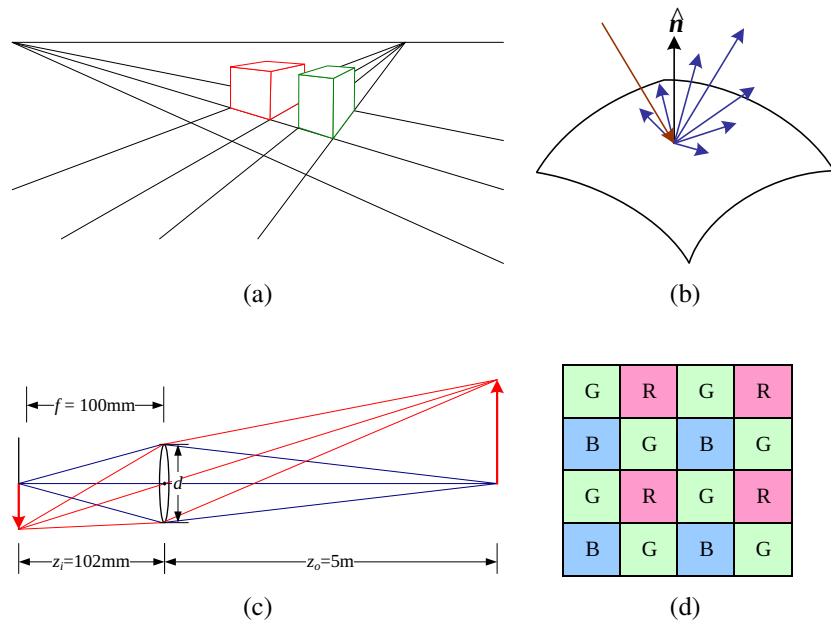


Figure 2.1 A few components of the image formation process: (a) perspective projection; (b) light scattering when hitting a surface; (c) lens optics; (d) Bayer color filter array.

Before we can analyze and manipulate images, we need to establish a vocabulary for describing the geometry of a scene. We also need to understand the image formation process that produced a particular image given a set of lighting conditions, scene geometry, surface properties, and camera optics. In this chapter, we present a simplified model of this image formation process.

Section 2.1 introduces the basic geometric primitives used throughout the book (points, lines, and planes) and the *geometric* transformations that project these 3D quantities into 2D image features (Figure 2.1a). Section 2.2 describes how lighting, surface properties (Figure 2.1b), and camera *optics* (Figure 2.1c) interact to produce the color values that fall onto the image sensor. Section 2.3 describes how continuous color images are turned into discrete digital *samples* inside the image sensor (Figure 2.1d) and how to avoid (or at least characterize) sampling deficiencies, such as aliasing.

The material covered in this chapter is but a brief summary of a very rich and deep set of topics, traditionally covered in a number of separate fields. A more thorough introduction to the geometry of points, lines, planes, and projections can be found in textbooks on multi-view geometry (Hartley and Zisserman 2004; Faugeras and Luong 2001) and computer graphics (Hughes, van Dam *et al.* 2013). The image formation (synthesis) process is traditionally taught as part of a computer graphics curriculum (Glassner 1995; Watt 1995; Hughes, van Dam *et al.* 2013; Marschner and Shirley 2015) but it is also studied in physics-based computer vision (Wolff, Shafer, and Healey 1992a). The behavior of camera lens systems is studied in optics (Möller 1988; Ray 2002; Hecht 2015). Some good books on color theory are Healey and Shafer (1992), Wandell (1995), and Wyszecki and Stiles (2000), with Livingstone (2008) providing a more fun and informal introduction to the topic of color perception. Topics relating to sampling and aliasing are covered in textbooks on signal and image processing (Crane 1997; Jähne 1997; Oppenheim and Schaffer 1996; Oppenheim, Schaffer, and Buck 1999; Pratt 2007; Russ 2007; Burger and Burge 2008; Gonzalez and Woods 2017). The recent book by Ikeuchi, Matsushita *et al.* (2020) also covers 3D geometry, photometry, and sensor models, with an emphasis on *active illumination* systems.

A note to students: If you have already studied computer graphics, you may want to skim the material in Section 2.1, although the sections on projective depth and object-centered projection near the end of Section 2.1.4 may be new to you. Similarly, physics students (as well as computer graphics students) will mostly be familiar with Section 2.2. Finally, students with a good background in image processing will already be familiar with sampling issues (Section 2.3) as well as some of the material in Chapter 3.

2.1 Geometric primitives and transformations

In this section, we introduce the basic 2D and 3D primitives used in this textbook, namely points, lines, and planes. We also describe how 3D features are projected into 2D features. More detailed descriptions of these topics (along with a gentler and more intuitive introduction) can be found in textbooks on multiple-view geometry (Hartley and Zisserman 2004; Faugeras and Luong 2001).

Geometric primitives form the basic building blocks used to describe three-dimensional shapes. In this section, we introduce points, lines, and planes. Later sections of the book discuss curves (Sections 7.3 and 12.2), surfaces (Section 13.3), and volumes (Section 13.5).

2D points. 2D points (pixel coordinates in an image) can be denoted using a pair of values, $\mathbf{x} = (x, y) \in \mathcal{R}^2$, or alternatively,

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}. \quad (2.1)$$

(As stated in the introduction, we use the $(x_1, x_2, \dots)$ notation to denote column vectors.)

2D points can also be represented using *homogeneous coordinates*, $\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{w}) \in \mathcal{P}^2$, where vectors that differ only by scale are considered to be equivalent. $\mathcal{P}^2 = \mathcal{R}^3 - (0, 0, 0)$ is called the 2D *projective space*.

A homogeneous vector $\tilde{\mathbf{x}}$ can be converted back into an *inhomogeneous* vector $\mathbf{x}$ by dividing through by the last element $\tilde{w}$, i.e.,

$$\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{w}) = \tilde{w}(x, y, 1) = \tilde{w}\bar{\mathbf{x}}, \quad (2.2)$$

where $\bar{\mathbf{x}} = (x, y, 1)$ is the *augmented vector*. Homogeneous points whose last element is $\tilde{w} = 0$ are called *ideal points* or *points at infinity* and do not have an equivalent inhomogeneous representation.

2D lines. 2D lines can also be represented using homogeneous coordinates $\tilde{\mathbf{l}} = (a, b, c)$. The corresponding *line equation* is

$$\bar{\mathbf{x}} \cdot \tilde{\mathbf{l}} = ax + by + c = 0. \quad (2.3)$$

We can normalize the line equation vector so that $\mathbf{l} = (\hat{n}_x, \hat{n}_y, d) = (\hat{\mathbf{n}}, d)$ with $\|\hat{\mathbf{n}}\| = 1$. In this case, $\hat{\mathbf{n}}$ is the *normal vector* perpendicular to the line and d is its distance to the origin (Figure 2.2). (The one exception to this normalization is the *line at infinity* $\tilde{\mathbf{l}} = (0, 0, 1)$, which includes all (ideal) points at infinity.)

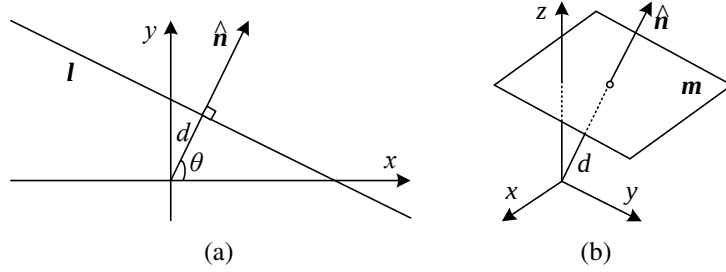


Figure 2.2 (a) 2D line equation and (b) 3D plane equation, expressed in terms of the normal $\hat{n}$ and distance to the origin d .

We can also express $\hat{n}$ as a function of rotation angle θ , $\hat{n} = (\hat{n}_x, \hat{n}_y) = (\cos \theta, \sin \theta)$ (Figure 2.2a). This representation is commonly used in the *Hough transform* line-finding algorithm, which is discussed in Section 7.4.2. The combination (θ, d) is also known as *polar coordinates*.

When using homogeneous coordinates, we can compute the intersection of two lines as

$$\tilde{\mathbf{x}} = \tilde{\mathbf{l}}_1 \times \tilde{\mathbf{l}}_2, \quad (2.4)$$

where $\times$ is the cross product operator. Similarly, the line joining two points can be written as

$$\tilde{\mathbf{l}} = \tilde{\mathbf{x}}_1 \times \tilde{\mathbf{x}}_2. \quad (2.5)$$

When trying to fit an intersection point to multiple lines or, conversely, a line to multiple points, least squares techniques (Section 8.1.1 and Appendix A.2) can be used, as discussed in Exercise 2.1.

2D conics. There are other algebraic curves that can be expressed with simple polynomial homogeneous equations. For example, the *conic sections* (so called because they arise as the intersection of a plane and a 3D cone) can be written using a *quadric* equation

$$\tilde{\mathbf{x}}^T \mathbf{Q} \tilde{\mathbf{x}} = 0. \quad (2.6)$$

Quadric equations play useful roles in the study of multi-view geometry and camera calibration (Hartley and Zisserman 2004; Faugeras and Luong 2001) but are not used extensively in this book.

3D points. Point coordinates in three dimensions can be written using inhomogeneous coordinates $\mathbf{x} = (x, y, z) \in \mathcal{R}^3$ or homogeneous coordinates $\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{z}, \tilde{w}) \in \mathcal{P}^3$. As before,

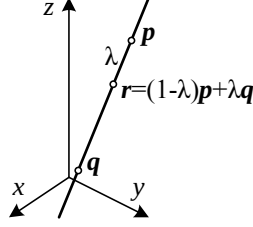


Figure 2.3 3D line equation, $\mathbf{r} = (1 - \lambda)\mathbf{p} + \lambda\mathbf{q}$.

it is sometimes useful to denote a 3D point using the augmented vector $\bar{\mathbf{x}} = (x, y, z, 1)$ with $\tilde{\mathbf{x}} = \tilde{w}\bar{\mathbf{x}}$.

3D planes. 3D planes can also be represented as homogeneous coordinates $\tilde{\mathbf{m}} = (a, b, c, d)$ with a corresponding plane equation

$$\bar{\mathbf{x}} \cdot \tilde{\mathbf{m}} = ax + by + cz + d = 0. \quad (2.7)$$

We can also normalize the plane equation as $\mathbf{m} = (\hat{n}_x, \hat{n}_y, \hat{n}_z, d) = (\hat{\mathbf{n}}, d)$ with $\|\hat{\mathbf{n}}\| = 1$. In this case, $\hat{\mathbf{n}}$ is the *normal vector* perpendicular to the plane and d is its distance to the origin (Figure 2.2b). As with the case of 2D lines, the *plane at infinity* $\tilde{\mathbf{m}} = (0, 0, 0, 1)$, which contains all the points at infinity, cannot be normalized (i.e., it does not have a unique normal or a finite distance).

We can express $\hat{\mathbf{n}}$ as a function of two angles (θ, ϕ) ,

$$\hat{\mathbf{n}} = (\cos \theta \cos \phi, \sin \theta \cos \phi, \sin \phi), \quad (2.8)$$

i.e., using *spherical coordinates*, but these are less commonly used than polar coordinates since they do not uniformly sample the space of possible normal vectors.

3D lines. Lines in 3D are less elegant than either lines in 2D or planes in 3D. One possible representation is to use two points on the line, $(\mathbf{p}, \mathbf{q})$. Any other point on the line can be expressed as a linear combination of these two points

$$\mathbf{r} = (1 - \lambda)\mathbf{p} + \lambda\mathbf{q}, \quad (2.9)$$

as shown in Figure 2.3. If we restrict $0 \leq \lambda \leq 1$, we get the *line segment* joining $\mathbf{p}$ and $\mathbf{q}$.

If we use homogeneous coordinates, we can write the line as

$$\tilde{\mathbf{r}} = \mu\tilde{\mathbf{p}} + \lambda\tilde{\mathbf{q}}. \quad (2.10)$$

A special case of this is when the second point is at infinity, i.e., $\tilde{\mathbf{q}} = (\hat{d}_x, \hat{d}_y, \hat{d}_z, 0) = (\hat{\mathbf{d}}, 0)$. Here, we see that $\hat{\mathbf{d}}$ is the *direction* of the line. We can then re-write the inhomogeneous 3D line equation as

$$\mathbf{r} = \mathbf{p} + \lambda \hat{\mathbf{d}}. \quad (2.11)$$

A disadvantage of the endpoint representation for 3D lines is that it has too many degrees of freedom, i.e., six (three for each endpoint) instead of the four degrees that a 3D line truly has. However, if we fix the two points on the line to lie in specific planes, we obtain a representation with four degrees of freedom. For example, if we are representing nearly vertical lines, then $z = 0$ and $z = 1$ form two suitable planes, i.e., the (x, y) coordinates in both planes provide the four coordinates describing the line. This kind of two-plane parameterization is used in the *light field* and *Lumigraph* image-based rendering systems described in Chapter 14 to represent the collection of rays seen by a camera as it moves in front of an object. The two-endpoint representation is also useful for representing line segments, even when their exact endpoints cannot be seen (only guessed at).

If we wish to represent all possible lines without bias towards any particular orientation, we can use *Plücker coordinates* (Hartley and Zisserman 2004, Section 3.2; Faugeras and Luong 2001, Chapter 3). These coordinates are the six independent non-zero entries in the 4×4 skew symmetric matrix

$$\mathbf{L} = \tilde{\mathbf{p}}\tilde{\mathbf{q}}^T - \tilde{\mathbf{q}}\tilde{\mathbf{p}}^T, \quad (2.12)$$

where $\tilde{\mathbf{p}}$ and $\tilde{\mathbf{q}}$ are *any* two (non-identical) points on the line. This representation has only four degrees of freedom, since $\mathbf{L}$ is homogeneous and also satisfies $|\mathbf{L}| = 0$, which results in a quadratic constraint on the Plücker coordinates.

In practice, the minimal representation is not essential for most applications. An adequate model of 3D lines can be obtained by estimating their direction (which may be known ahead of time, e.g., for architecture) and some point within the visible portion of the line (see Section 11.4.8) or by using the two endpoints, since lines are most often visible as finite line segments. However, if you are interested in more details about the topic of minimal line parameterizations, Förstner (2005) discusses various ways to infer and model 3D lines in projective geometry, as well as how to estimate the uncertainty in such fitted models.

3D quadrics. The 3D analog of a conic section is a quadric surface

$$\bar{\mathbf{x}}^T \mathbf{Q} \bar{\mathbf{x}} = 0 \quad (2.13)$$

(Hartley and Zisserman 2004, Chapter 3). Again, while quadric surfaces are useful in the study of multi-view geometry and can also serve as useful modeling primitives (spheres, ellipsoids, cylinders), we do not study them in great detail in this book.

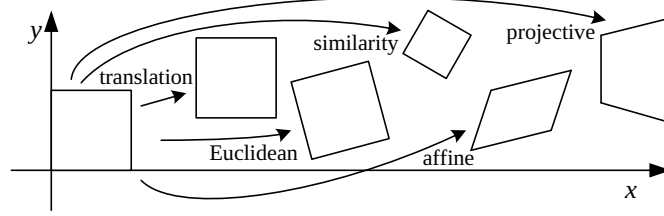


Figure 2.4 Basic set of 2D planar transformations.

2.1.1 2D transformations

Having defined our basic primitives, we can now turn our attention to how they can be transformed. The simplest transformations occur in the 2D plane are illustrated in Figure 2.4.

Translation. 2D translations can be written as $\mathbf{x}' = \mathbf{x} + \mathbf{t}$ or

$$\mathbf{x}' = \begin{bmatrix} \mathbf{I} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}}, \quad (2.14)$$

where $\mathbf{I}$ is the (2×2) identity matrix or

$$\bar{\mathbf{x}}' = \begin{bmatrix} \mathbf{I} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \bar{\mathbf{x}}, \quad (2.15)$$

where $\mathbf{0}$ is the zero vector. Using a 2×3 matrix results in a more compact notation, whereas using a full-rank 3×3 matrix (which can be obtained from the 2×3 matrix by appending a $[\mathbf{0}^T \ 1]$ row) makes it possible to chain transformations using matrix multiplication as well as to compute inverse transforms. Note that in any equation where an augmented vector such as $\bar{\mathbf{x}}$ appears on both sides, it can always be replaced with a full homogeneous vector $\tilde{\mathbf{x}}$.

Rotation + translation. This transformation is also known as *2D rigid body motion* or the *2D Euclidean transformation* (since Euclidean distances are preserved). It can be written as $\mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t}$ or

$$\mathbf{x}' = \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}}. \quad (2.16)$$

where

$$\mathbf{R} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (2.17)$$

is an orthonormal rotation matrix with $\mathbf{R}\mathbf{R}^T = \mathbf{I}$ and $|\mathbf{R}| = 1$.

Scaled rotation. Also known as the *similarity transform*, this transformation can be expressed as $\mathbf{x}' = s\mathbf{R}\mathbf{x} + \mathbf{t}$, where s is an arbitrary scale factor. It can also be written as

$$\mathbf{x}' = \begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}} = \begin{bmatrix} a & -b & t_x \\ b & a & t_y \end{bmatrix} \bar{\mathbf{x}}, \quad (2.18)$$

where we no longer require that $a^2 + b^2 = 1$. The similarity transform preserves angles between lines.

Affine. The affine transformation is written as $\mathbf{x}' = \mathbf{A}\bar{\mathbf{x}}$, where $\mathbf{A}$ is an arbitrary 2×3 matrix, i.e.,

$$\mathbf{x}' = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} \bar{\mathbf{x}}. \quad (2.19)$$

Parallel lines remain parallel under affine transformations.

Projective. This transformation, also known as a *perspective transform* or *homography*, operates on homogeneous coordinates,

$$\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}\tilde{\mathbf{x}}, \quad (2.20)$$

where $\tilde{\mathbf{H}}$ is an arbitrary 3×3 matrix. Note that $\tilde{\mathbf{H}}$ is homogeneous, i.e., it is only defined up to a scale, and that two $\tilde{\mathbf{H}}$ matrices that differ only by scale are equivalent. The resulting homogeneous coordinate $\tilde{\mathbf{x}}'$ must be normalized in order to obtain an inhomogeneous result $\mathbf{x}$, i.e.,

$$x' = \frac{h_{00}x + h_{01}y + h_{02}}{h_{20}x + h_{21}y + h_{22}} \quad \text{and} \quad y' = \frac{h_{10}x + h_{11}y + h_{12}}{h_{20}x + h_{21}y + h_{22}}. \quad (2.21)$$

Perspective transformations preserve straight lines (i.e., they remain straight after the transformation).

Hierarchy of 2D transformations. The preceding set of transformations are illustrated in Figure 2.4 and summarized in Table 2.1. The easiest way to think of them is as a set of (potentially restricted) 3×3 matrices operating on 2D homogeneous coordinate vectors. Hartley and Zisserman (2004) contains a more detailed description of the hierarchy of 2D planar transformations.

The above transformations form a nested set of *groups*, i.e., they are closed under composition and have an inverse that is a member of the same group. (This will be important later when applying these transformations to images in Section 3.6.) Each (simpler) group is a subgroup of the more complex group below it. The mathematics of such *Lie groups* and their

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} \mathbf{I} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	2	orientation	
rigid (Euclidean)	$\begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	3	lengths	
similarity	$\begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	4	angles	
affine	$\begin{bmatrix} \mathbf{A} \end{bmatrix}_{2 \times 3}$	6	parallelism	
projective	$\begin{bmatrix} \tilde{\mathbf{H}} \end{bmatrix}_{3 \times 3}$	8	straight lines	

Table 2.1 *Hierarchy of 2D coordinate transformations, listing the transformation name, its matrix form, the number of degrees of freedom, what geometric properties it preserves, and a mnemonic icon. Each transformation also preserves the properties listed in the rows below it, i.e., similarity preserves not only angles but also parallelism and straight lines. The 2×3 matrices are extended with a third $[\mathbf{0}^T \ 1]$ row to form a full 3×3 matrix for homogeneous coordinate transformations.*

related algebras (tangent spaces at the origin) are discussed in a number of recent robotics tutorials (Dellaert and Kaess 2017; Blanco 2019; Solà, Deray, and Atchuthan 2019), where the 2D rotation and rigid transforms are called $\text{SO}(2)$ and $\text{SE}(2)$, which stand for the *special orthogonal* and *special Euclidean* groups.¹

Co-vectors. While the above transformations can be used to transform points in a 2D plane, can they also be used directly to transform a line equation? Consider the homogeneous equation $\tilde{\mathbf{l}} \cdot \tilde{\mathbf{x}} = 0$. If we transform $\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}\tilde{\mathbf{x}}$, we obtain

$$\tilde{\mathbf{l}}' \cdot \tilde{\mathbf{x}}' = \tilde{\mathbf{l}}'^T \tilde{\mathbf{H}}\tilde{\mathbf{x}} = (\tilde{\mathbf{H}}^T \tilde{\mathbf{l}}')^T \tilde{\mathbf{x}} = \tilde{\mathbf{l}} \cdot \tilde{\mathbf{x}} = 0, \quad (2.22)$$

i.e., $\tilde{\mathbf{l}}' = \tilde{\mathbf{H}}^{-T} \tilde{\mathbf{l}}$. Thus, the action of a projective transformation on a *co-vector* such as a 2D line or 3D normal can be represented by the transposed inverse of the matrix, which is equivalent to the *adjoint* of $\tilde{\mathbf{H}}$, since projective transformation matrices are homogeneous. Jim Blinn (1998) describes (in Chapters 9 and 10) the ins and outs of notating and manipulating co-vectors.

¹The term *special* refers to the desired condition of no reflection, i.e., $\det[\mathbf{R}] = 1$.

While the above transformations are the ones we use most extensively, a number of additional transformations are sometimes used.

Stretch/squash. This transformation changes the aspect ratio of an image,

$$\begin{aligned}x' &= s_x x + t_x \\ y' &= s_y y + t_y,\end{aligned}$$

and is a restricted form of an affine transformation. Unfortunately, it does not nest cleanly with the groups listed in Table 2.1.

Planar surface flow. This eight-parameter transformation (Horn 1986; Bergen, Anandan *et al.* 1992; Girod, Greiner, and Niemann 2000),

$$\begin{aligned}x' &= a_0 + a_1 x + a_2 y + a_6 x^2 + a_7 xy \\ y' &= a_3 + a_4 x + a_5 y + a_6 xy + a_7 y^2,\end{aligned}$$

arises when a planar surface undergoes a small 3D motion. It can thus be thought of as a small motion approximation to a full homography. Its main attraction is that it is *linear* in the motion parameters, a_k , which are often the quantities being estimated.

Bilinear interpolant. This eight-parameter transform (Wolberg 1990),

$$\begin{aligned}x' &= a_0 + a_1 x + a_2 y + a_6 xy \\ y' &= a_3 + a_4 x + a_5 y + a_7 xy,\end{aligned}$$

can be used to interpolate the deformation due to the motion of the four corner points of a square. (In fact, it can interpolate the motion of any four non-collinear points.) While the deformation is linear in the motion parameters, it does not generally preserve straight lines (only lines parallel to the square axes). However, it is often quite useful, e.g., in the interpolation of sparse grids using splines (Section 9.2.2).

2.1.2 3D transformations

The set of three-dimensional coordinate transformations is very similar to that available for 2D transformations and is summarized in Table 2.2. As in 2D, these transformations form a nested set of groups. Hartley and Zisserman (2004, Section 2.4) give a more detailed description of this hierarchy.

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} \mathbf{I} & \mathbf{t} \end{bmatrix}_{3 \times 4}$	3	orientation	
rigid (Euclidean)	$\begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix}_{3 \times 4}$	6	lengths	
similarity	$\begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix}_{3 \times 4}$	7	angles	
affine	$\begin{bmatrix} \mathbf{A} \end{bmatrix}_{3 \times 4}$	12	parallelism	
projective	$\begin{bmatrix} \tilde{\mathbf{H}} \end{bmatrix}_{4 \times 4}$	15	straight lines	

Table 2.2 *Hierarchy of 3D coordinate transformations. Each transformation also preserves the properties listed in the rows below it, i.e., similarity preserves not only angles but also parallelism and straight lines. The 3×4 matrices are extended with a fourth $[\mathbf{0}^T \ 1]$ row to form a full 4×4 matrix for homogeneous coordinate transformations. The mnemonic icons are drawn in 2D but are meant to suggest transformations occurring in a full 3D cube.*

Translation. 3D translations can be written as $\mathbf{x}' = \mathbf{x} + \mathbf{t}$ or

$$\mathbf{x}' = \begin{bmatrix} \mathbf{I} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}}, \quad (2.23)$$

where $\mathbf{I}$ is the (3×3) identity matrix.

Rotation + translation. Also known as 3D *rigid body motion* or the 3D *Euclidean transformation* or $\text{SE}(3)$, it can be written as $\mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t}$ or

$$\mathbf{x}' = \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}}, \quad (2.24)$$

where $\mathbf{R}$ is a 3×3 orthonormal rotation matrix with $\mathbf{R}\mathbf{R}^T = \mathbf{I}$ and $|\mathbf{R}| = 1$. Note that sometimes it is more convenient to describe a rigid motion using

$$\mathbf{x}' = \mathbf{R}(\mathbf{x} - \mathbf{c}) = \mathbf{R}\mathbf{x} - \mathbf{R}\mathbf{c}, \quad (2.25)$$

where $\mathbf{c}$ is the center of rotation (often the camera center).

Compactly parameterizing a 3D rotation is a non-trivial task, which we describe in more detail below.

Scaled rotation. The 3D *similarity transform* can be expressed as $\mathbf{x}' = s\mathbf{R}\mathbf{x} + \mathbf{t}$ where s is an arbitrary scale factor. It can also be written as

$$\mathbf{x}' = \begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}}. \quad (2.26)$$

This transformation preserves angles between lines and planes.

Affine. The affine transform is written as $\mathbf{x}' = \mathbf{A}\bar{\mathbf{x}}$, where $\mathbf{A}$ is an arbitrary 3×4 matrix, i.e.,

$$\mathbf{x}' = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{bmatrix} \bar{\mathbf{x}}. \quad (2.27)$$

Parallel lines and planes remain parallel under affine transformations.

Projective. This transformation, variously known as a *3D perspective transform*, *homography*, or *collineation*, operates on homogeneous coordinates,

$$\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}\tilde{\mathbf{x}}, \quad (2.28)$$

where $\tilde{\mathbf{H}}$ is an arbitrary 4×4 homogeneous matrix. As in 2D, the resulting homogeneous coordinate $\tilde{\mathbf{x}}'$ must be normalized in order to obtain an inhomogeneous result $\mathbf{x}$. Perspective transformations preserve straight lines (i.e., they remain straight after the transformation).

2.1.3 3D rotations

The biggest difference between 2D and 3D coordinate transformations is that the parameterization of the 3D rotation matrix $\mathbf{R}$ is not as straightforward, as several different possibilities exist.

Euler angles

A rotation matrix can be formed as the product of three rotations around three cardinal axes, e.g., x , y , and z , or x , y , and x . This is generally a bad idea, as the result depends on the order in which the transforms are applied.² What is worse, it is not always possible to move smoothly in the parameter space, i.e., sometimes one or more of the Euler angles change dramatically in response to a small change in rotation.³ For these reasons, we do not even

²However, in special situations, such as describing the motion of a pan-tilt head, these angles may be more intuitive.

³In robotics, this is sometimes referred to as *gimbal lock*.

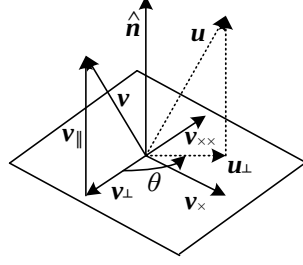


Figure 2.5 Rotation around an axis $\hat{\mathbf{n}}$ by an angle θ .

give the formula for Euler angles in this book—interested readers can look in other textbooks or technical reports (Faugeras 1993; Diebel 2006). Note that, in some applications, if the rotations are known to be a set of uni-axial transforms, they can always be represented using an explicit set of rigid transformations.

Axis/angle (exponential twist)

A rotation can be represented by a rotation axis $\hat{\mathbf{n}}$ and an angle θ , or equivalently by a 3D vector $\boldsymbol{\omega} = \theta\hat{\mathbf{n}}$. Figure 2.5 shows how we can compute the equivalent rotation. First, we project the vector $\mathbf{v}$ onto the axis $\hat{\mathbf{n}}$ to obtain

$$\mathbf{v}_{\parallel} = \hat{\mathbf{n}}(\hat{\mathbf{n}} \cdot \mathbf{v}) = (\hat{\mathbf{n}}\hat{\mathbf{n}}^T)\mathbf{v}, \quad (2.29)$$

which is the component of $\mathbf{v}$ that is not affected by the rotation. Next, we compute the perpendicular residual of $\mathbf{v}$ from $\hat{\mathbf{n}}$,

$$\mathbf{v}_{\perp} = \mathbf{v} - \mathbf{v}_{\parallel} = (\mathbf{I} - \hat{\mathbf{n}}\hat{\mathbf{n}}^T)\mathbf{v}. \quad (2.30)$$

We can rotate this vector by 90° using the cross product,

$$\mathbf{v}_{\times} = \hat{\mathbf{n}} \times \mathbf{v}_{\perp} = \hat{\mathbf{n}} \times \mathbf{v} = [\hat{\mathbf{n}}]_{\times} \mathbf{v}, \quad (2.31)$$

where $[\hat{\mathbf{n}}]_{\times}$ is the matrix form of the cross product operator with the vector $\hat{\mathbf{n}} = (\hat{n}_x, \hat{n}_y, \hat{n}_z)$,

$$[\hat{\mathbf{n}}]_{\times} = \begin{bmatrix} 0 & -\hat{n}_z & \hat{n}_y \\ \hat{n}_z & 0 & -\hat{n}_x \\ -\hat{n}_y & \hat{n}_x & 0 \end{bmatrix}. \quad (2.32)$$

Note that rotating this vector by another 90° is equivalent to taking the cross product again,

$$\mathbf{v}_{\times\times} = \hat{\mathbf{n}} \times \mathbf{v}_{\times} = [\hat{\mathbf{n}}]_{\times}^2 \mathbf{v} = -\mathbf{v}_{\perp},$$

and hence

$$\mathbf{v}_{\parallel} = \mathbf{v} - \mathbf{v}_{\perp} = \mathbf{v} + \mathbf{v}_{\times \times} = (\mathbf{I} + [\hat{\mathbf{n}}]_{\times}^2) \mathbf{v}.$$

We can now compute the in-plane component of the rotated vector $\mathbf{u}$ as

$$\mathbf{u}_{\perp} = \cos \theta \mathbf{v}_{\perp} + \sin \theta \mathbf{v}_{\times} = (\sin \theta [\hat{\mathbf{n}}]_{\times} - \cos \theta [\hat{\mathbf{n}}]_{\times}^2) \mathbf{v}.$$

Putting all these terms together, we obtain the final rotated vector as

$$\mathbf{u} = \mathbf{u}_{\perp} + \mathbf{v}_{\parallel} = (\mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_{\times} + (1 - \cos \theta) [\hat{\mathbf{n}}]_{\times}^2) \mathbf{v}. \quad (2.33)$$

We can therefore write the rotation matrix corresponding to a rotation by θ around an axis $\hat{\mathbf{n}}$ as

$$\mathbf{R}(\hat{\mathbf{n}}, \theta) = \mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_{\times} + (1 - \cos \theta) [\hat{\mathbf{n}}]_{\times}^2, \quad (2.34)$$

which is known as *Rodrigues' formula* (Ayache 1989).

The product of the axis $\hat{\mathbf{n}}$ and angle θ , $\boldsymbol{\omega} = \theta \hat{\mathbf{n}} = (\omega_x, \omega_y, \omega_z)$, is a minimal representation for a 3D rotation. Rotations through common angles such as multiples of 90° can be represented exactly (and converted to exact matrices) if θ is stored in degrees. Unfortunately, this representation is not unique, since we can always add a multiple of 360° (2π radians) to θ and get the same rotation matrix. As well, $(\hat{\mathbf{n}}, \theta)$ and $(-\hat{\mathbf{n}}, -\theta)$ represent the same rotation.

However, for small rotations (e.g., corrections to rotations), this is an excellent choice. In particular, for small (infinitesimal or instantaneous) rotations and θ expressed in radians, Rodrigues' formula simplifies to

$$\mathbf{R}(\boldsymbol{\omega}) \approx \mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_{\times} \approx \mathbf{I} + [\theta \hat{\mathbf{n}}]_{\times} = \begin{bmatrix} 1 & -\omega_z & \omega_y \\ \omega_z & 1 & -\omega_x \\ -\omega_y & \omega_x & 1 \end{bmatrix}, \quad (2.35)$$

which gives a nice linearized relationship between the rotation parameters $\boldsymbol{\omega}$ and $\mathbf{R}$. We can also write $\mathbf{R}(\boldsymbol{\omega}) \mathbf{v} \approx \mathbf{v} + \boldsymbol{\omega} \times \mathbf{v}$, which is handy when we want to compute the derivative of $\mathbf{R} \mathbf{v}$ with respect to $\boldsymbol{\omega}$,

$$\frac{\partial \mathbf{R} \mathbf{v}}{\partial \boldsymbol{\omega}^T} = -[\mathbf{v}]_{\times} = \begin{bmatrix} 0 & z & -y \\ -z & 0 & x \\ y & -x & 0 \end{bmatrix}. \quad (2.36)$$

Another way to derive a rotation through a finite angle is called the *exponential twist* (Murray, Li, and Sastry 1994). A rotation by an angle θ is equivalent to k rotations through θ/k . In the limit as $k \rightarrow \infty$, we obtain

$$\mathbf{R}(\hat{\mathbf{n}}, \theta) = \lim_{k \rightarrow \infty} (\mathbf{I} + \frac{1}{k} [\theta \hat{\mathbf{n}}]_{\times})^k = \exp [\boldsymbol{\omega}]_{\times}. \quad (2.37)$$

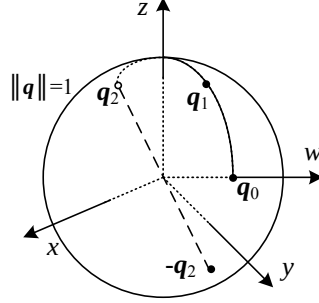


Figure 2.6 Unit quaternions live on the unit sphere $\|\mathbf{q}\| = 1$. This figure shows a smooth trajectory through the three quaternions $\mathbf{q}_0$, $\mathbf{q}_1$, and $\mathbf{q}_2$. The antipodal point to $\mathbf{q}_2$, namely $-\mathbf{q}_2$, represents the same rotation as $\mathbf{q}_2$.

If we expand the matrix exponential as a Taylor series (using the identity $[\hat{\mathbf{n}}]_{\times}^{k+2} = -[\hat{\mathbf{n}}]_{\times}^k$, $k > 0$, and again assuming θ is in radians),

$$\begin{aligned}
 \exp[\omega]_{\times} &= \mathbf{I} + \theta[\hat{\mathbf{n}}]_{\times} + \frac{\theta^2}{2}[\hat{\mathbf{n}}]_{\times}^2 + \frac{\theta^3}{3!}[\hat{\mathbf{n}}]_{\times}^3 + \cdots \\
 &= \mathbf{I} + \left(\theta - \frac{\theta^3}{3!} + \cdots\right)[\hat{\mathbf{n}}]_{\times} + \left(\frac{\theta^2}{2} - \frac{\theta^4}{4!} + \cdots\right)[\hat{\mathbf{n}}]_{\times}^2 \\
 &= \mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_{\times} + (1 - \cos \theta) [\hat{\mathbf{n}}]_{\times}^2,
 \end{aligned} \tag{2.38}$$

which yields the familiar Rodrigues' formula.

In robotics (and group theory), rotations are called $\text{SO}(3)$, i.e., the *special orthogonal* group in 3D. The incremental rotations ω are associated with a Lie algebra $\text{se}(3)$ and are the preferred way to formulate rotation derivatives and to model uncertainties in rotation estimates (Blanco 2019; Solà, Deray, and Atchuthan 2019).

Unit quaternions

The unit quaternion representation is closely related to the angle/axis representation. A unit quaternion is a unit length 4-vector whose components can be written as $\mathbf{q} = (q_x, q_y, q_z, q_w)$ or $\mathbf{q} = (x, y, z, w)$ for short. Unit quaternions live on the unit sphere $\|\mathbf{q}\| = 1$ and *antipodal* (opposite sign) quaternions, $\mathbf{q}$ and $-\mathbf{q}$, represent the same rotation (Figure 2.6). Other than this ambiguity (dual covering), the unit quaternion representation of a rotation is unique. Furthermore, the representation is *continuous*, i.e., as rotation matrices vary continuously, you can find a continuous quaternion representation, although the path on the quaternion sphere may wrap all the way around before returning to the “origin” $\mathbf{q}_o = (0, 0, 0, 1)$. For

these and other reasons given below, quaternions are a very popular representation for pose and for pose interpolation in computer graphics (Shoemake 1985).

Quaternions can be derived from the axis/angle representation through the formula

$$\mathbf{q} = (\mathbf{v}, w) = \left(\sin \frac{\theta}{2} \hat{\mathbf{n}}, \cos \frac{\theta}{2} \right), \quad (2.39)$$

where $\hat{\mathbf{n}}$ and θ are the rotation axis and angle. Using the trigonometric identities $\sin \theta = 2 \sin \frac{\theta}{2} \cos \frac{\theta}{2}$ and $(1 - \cos \theta) = 2 \sin^2 \frac{\theta}{2}$, Rodrigues' formula can be converted to

$$\begin{aligned} \mathbf{R}(\hat{\mathbf{n}}, \theta) &= \mathbf{I} + \sin \theta [\hat{\mathbf{n}}]_{\times} + (1 - \cos \theta) [\hat{\mathbf{n}}]_{\times}^2 \\ &= \mathbf{I} + 2w[\mathbf{v}]_{\times} + 2[\mathbf{v}]_{\times}^2. \end{aligned} \quad (2.40)$$

This suggests a quick way to rotate a vector $\mathbf{v}$ by a quaternion using a series of cross products, scalings, and additions. To obtain a formula for $\mathbf{R}(\mathbf{q})$ as a function of (x, y, z, w) , recall that

$$[\mathbf{v}]_{\times} = \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix} \quad \text{and} \quad [\mathbf{v}]_{\times}^2 = \begin{bmatrix} -y^2 - z^2 & xy & xz \\ xy & -x^2 - z^2 & yz \\ xz & yz & -x^2 - y^2 \end{bmatrix}.$$

We thus obtain

$$\mathbf{R}(\mathbf{q}) = \begin{bmatrix} 1 - 2(y^2 + z^2) & 2(xy - zw) & 2(xz + yw) \\ 2(xy + zw) & 1 - 2(x^2 + z^2) & 2(yz - xw) \\ 2(xz - yw) & 2(yz + xw) & 1 - 2(x^2 + y^2) \end{bmatrix}. \quad (2.41)$$

The diagonal terms can be made more symmetrical by replacing $1 - 2(y^2 + z^2)$ with $(x^2 + w^2 - y^2 - z^2)$, etc.

The nicest aspect of unit quaternions is that there is a simple algebra for composing rotations expressed as unit quaternions. Given two quaternions $\mathbf{q}_0 = (\mathbf{v}_0, w_0)$ and $\mathbf{q}_1 = (\mathbf{v}_1, w_1)$, the *quaternion multiply* operator is defined as

$$\mathbf{q}_2 = \mathbf{q}_0 \mathbf{q}_1 = (\mathbf{v}_0 \times \mathbf{v}_1 + w_0 \mathbf{v}_1 + w_1 \mathbf{v}_0, w_0 w_1 - \mathbf{v}_0 \cdot \mathbf{v}_1), \quad (2.42)$$

with the property that $\mathbf{R}(\mathbf{q}_2) = \mathbf{R}(\mathbf{q}_0)\mathbf{R}(\mathbf{q}_1)$. Note that quaternion multiplication is *not* commutative, just as 3D rotations and matrix multiplications are not.

Taking the inverse of a quaternion is easy: Just flip the sign of $\mathbf{v}$ or w (but not both!). (You can verify this has the desired effect of transposing the $\mathbf{R}$ matrix in (2.41).) Thus, we can also define *quaternion division* as

$$\mathbf{q}_2 = \mathbf{q}_0 / \mathbf{q}_1 = \mathbf{q}_0 \mathbf{q}_1^{-1} = (\mathbf{v}_0 \times \mathbf{v}_1 + w_0 \mathbf{v}_1 - w_1 \mathbf{v}_0, -w_0 w_1 - \mathbf{v}_0 \cdot \mathbf{v}_1). \quad (2.43)$$

procedure *slerp*($\mathbf{q}_0, \mathbf{q}_1, \alpha$):

1. $\mathbf{q}_r = \mathbf{q}_1 / \mathbf{q}_0 = (\mathbf{v}_r, w_r)$
2. if $w_r < 0$ then $\mathbf{q}_r \leftarrow -\mathbf{q}_r$
3. $\theta_r = 2 \tan^{-1}(\|\mathbf{v}_r\|/w_r)$
4. $\hat{\mathbf{n}}_r = \mathcal{N}(\mathbf{v}_r) = \mathbf{v}_r / \|\mathbf{v}_r\|$
5. $\theta_\alpha = \alpha \theta_r$
6. $\mathbf{q}_\alpha = (\sin \frac{\theta_\alpha}{2} \hat{\mathbf{n}}_r, \cos \frac{\theta_\alpha}{2})$
7. **return** $\mathbf{q}_2 = \mathbf{q}_\alpha \mathbf{q}_0$

Algorithm 2.1 *Spherical linear interpolation (slerp). The axis and total angle are first computed from the quaternion ratio. (This computation can be lifted outside an inner loop that generates a set of interpolated position for animation.) An incremental quaternion is then computed and multiplied by the starting rotation quaternion.*

This is useful when the *incremental rotation* between two rotations is desired.

In particular, if we want to determine a rotation that is partway between two given rotations, we can compute the incremental rotation, take a fraction of the angle, and compute the new rotation. This procedure is called *spherical linear interpolation* or *slerp* for short (Shoemake 1985) and is given in Algorithm 2.1. Note that Shoemake presents two formulas other than the one given here. The first exponentiates $\mathbf{q}_r$ by alpha before multiplying the original quaternion,

$$\mathbf{q}_2 = \mathbf{q}_r^\alpha \mathbf{q}_0, \quad (2.44)$$

while the second treats the quaternions as 4-vectors on a sphere and uses

$$\mathbf{q}_2 = \frac{\sin(1-\alpha)\theta}{\sin \theta} \mathbf{q}_0 + \frac{\sin \alpha\theta}{\sin \theta} \mathbf{q}_1, \quad (2.45)$$

where $\theta = \cos^{-1}(\mathbf{q}_0 \cdot \mathbf{q}_1)$ and the dot product is directly between the quaternion 4-vectors. All of these formulas give comparable results, although care should be taken when $\mathbf{q}_0$ and $\mathbf{q}_1$ are close together, which is why I prefer to use an arctangent to establish the rotation angle.

Which rotation representation is better?

The choice of representation for 3D rotations depends partly on the application.

The axis/angle representation is minimal, and hence does not require any additional constraints on the parameters (no need to re-normalize after each update). If the angle is expressed in degrees, it is easier to understand the pose (say, 90° twist around x -axis), and also easier to express exact rotations. When the angle is in radians, the derivatives of $\mathbf{R}$ with respect to ω can easily be computed (2.36).

Quaternions, on the other hand, are better if you want to keep track of a smoothly moving camera, since there are no discontinuities in the representation. It is also easier to interpolate between rotations and to chain rigid transformations (Murray, Li, and Sastry 1994; Bregler and Malik 1998).

My usual preference is to use quaternions, but to update their estimates using an incremental rotation, as described in Section 11.2.2.

2.1.4 3D to 2D projections

Now that we know how to represent 2D and 3D geometric primitives and how to transform them spatially, we need to specify how 3D primitives are projected onto the image plane. We can do this using a linear 3D to 2D projection matrix. The simplest model is orthography, which requires no division to get the final (inhomogeneous) result. The more commonly used model is perspective, since this more accurately models the behavior of real cameras.

Orthography and para-perspective

An orthographic projection simply drops the z component of the three-dimensional coordinate $\mathbf{p}$ to obtain the 2D point $\mathbf{x}$. (In this section, we use $\mathbf{p}$ to denote 3D points and $\mathbf{x}$ to denote 2D points.) This can be written as

$$\mathbf{x} = [\mathbf{I}_{2 \times 2} | \mathbf{0}] \mathbf{p}. \quad (2.46)$$

If we are using homogeneous (projective) coordinates, we can write

$$\tilde{\mathbf{x}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{p}}, \quad (2.47)$$

i.e., we drop the z component but keep the w component. Orthography is an approximate model for long focal length (telephoto) lenses and objects whose depth is *shallow* relative to their distance to the camera (Sawhney and Hanson 1991). It is exact only for *telecentric* lenses (Baker and Nayar 1999, 2001).

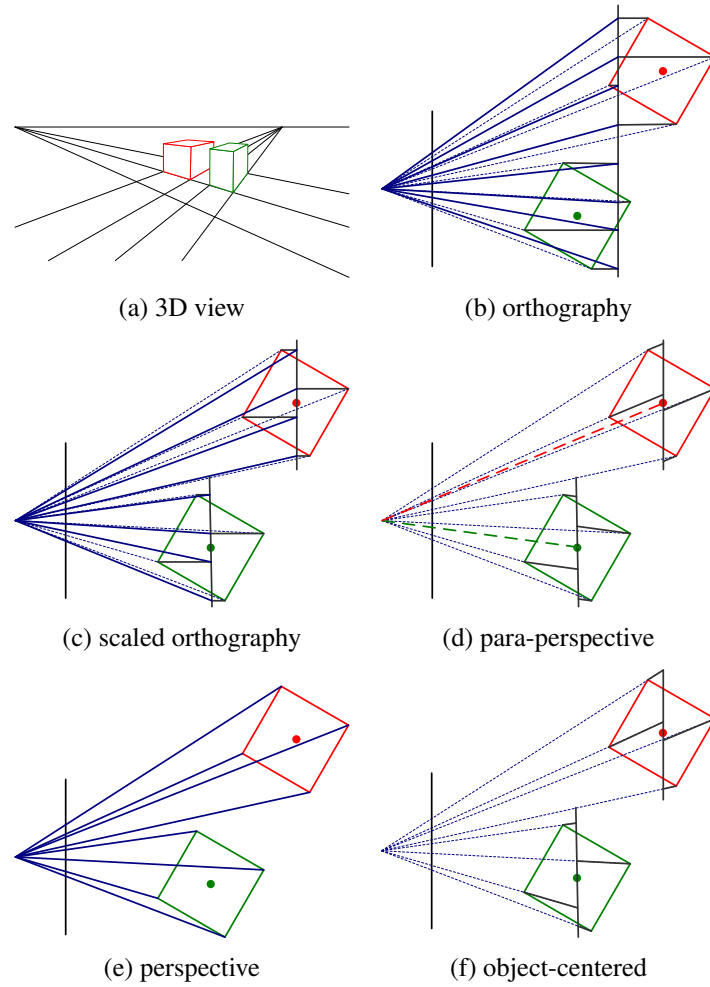


Figure 2.7 Commonly used projection models: (a) 3D view of world, (b) orthography, (c) scaled orthography, (d) para-perspective, (e) perspective, (f) object-centered. Each diagram shows a top-down view of the projection. Note how parallel lines on the ground plane and box sides remain parallel in the non-perspective projections.

In practice, world coordinates (which may measure dimensions in meters) need to be scaled to fit onto an image sensor (physically measured in millimeters, but ultimately measured in pixels). For this reason, *scaled orthography* is actually more commonly used,

$$\mathbf{x} = [s\mathbf{I}_{2 \times 2} | \mathbf{0}] \mathbf{p}. \quad (2.48)$$

This model is equivalent to first projecting the world points onto a local fronto-parallel image plane and then scaling this image using regular perspective projection. The scaling can be the same for all parts of the scene (Figure 2.7b) or it can be different for objects that are being modeled independently (Figure 2.7c). More importantly, the scaling can vary from frame to frame when estimating *structure from motion*, which can better model the scale change that occurs as an object approaches the camera.

Scaled orthography is a popular model for reconstructing the 3D shape of objects far away from the camera, since it greatly simplifies certain computations. For example, *pose* (camera orientation) can be estimated using simple least squares (Section 11.2.1). Under orthography, structure and motion can simultaneously be estimated using *factorization* (singular value decomposition), as discussed in Section 11.4.1 (Tomasi and Kanade 1992).

A closely related projection model is *para-perspective* (Aloimonos 1990; Poelman and Kanade 1997). In this model, object points are again first projected onto a local reference plane parallel to the image plane. However, rather than being projected orthogonally to this plane, they are projected *parallel* to the line of sight to the object center (Figure 2.7d). This is followed by the usual projection onto the final image plane, which again amounts to a scaling. The combination of these two projections is therefore *affine* and can be written as

$$\tilde{\mathbf{x}} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{p}}. \quad (2.49)$$

Note how parallel lines in 3D remain parallel after projection in Figure 2.7b–d. Para-perspective provides a more accurate projection model than scaled orthography, without incurring the added complexity of per-pixel perspective division, which invalidates traditional factorization methods (Poelman and Kanade 1997).

Perspective

The most commonly used projection in computer graphics and computer vision is true 3D *perspective* (Figure 2.7e). Here, points are projected onto the image plane by dividing them

by their z component. Using inhomogeneous coordinates, this can be written as

$$\bar{\mathbf{x}} = \mathcal{P}_z(\mathbf{p}) = \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix}. \quad (2.50)$$

In homogeneous coordinates, the projection has a simple linear form,

$$\tilde{\mathbf{x}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tilde{\mathbf{p}}, \quad (2.51)$$

i.e., we drop the w component of $\mathbf{p}$. Thus, after projection, it is not possible to recover the *distance* of the 3D point from the image, which makes sense for a 2D imaging sensor.

A form often seen in computer graphics systems is a two-step projection that first projects 3D coordinates into *normalized device coordinates* $(x, y, z) \in [-1, 1] \times [-1, 1] \times [0, 1]$, and then rescales these coordinates to integer pixel coordinates using a *viewport* transformation (Watt 1995; OpenGL-ARB 1997). The (initial) perspective projection is then represented using a 4×4 matrix

$$\tilde{\mathbf{x}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -z_{\text{far}}/z_{\text{range}} & z_{\text{near}}z_{\text{far}}/z_{\text{range}} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tilde{\mathbf{p}}, \quad (2.52)$$

where z_{near} and z_{far} are the near and far z *clipping planes* and $z_{\text{range}} = z_{\text{far}} - z_{\text{near}}$. Note that the first two rows are actually scaled by the focal length and the aspect ratio so that visible rays are mapped to $(x, y, z) \in [-1, 1]^2$. The reason for keeping the third row, rather than dropping it, is that visibility operations, such as *z-buffering*, require a depth for every graphical element that is being rendered.

If we set $z_{\text{near}} = 1$, $z_{\text{far}} \rightarrow \infty$, and switch the sign of the third row, the third element of the normalized screen vector becomes the inverse depth, i.e., the *disparity* (Okutomi and Kanade 1993). This can be quite convenient in many cases since, for cameras moving around outdoors, the inverse depth to the camera is often a more well-conditioned parameterization than direct 3D distance.

While a regular 2D image sensor has no way of measuring distance to a surface point, *range sensors* (Section 13.2) and stereo matching algorithms (Chapter 12) can compute such values. It is then convenient to be able to map from a sensor-based depth or disparity value d directly back to a 3D location using the inverse of a 4×4 matrix (Section 2.1.4). We can do this if we represent perspective projection using a full-rank 4×4 matrix, as in (2.64).

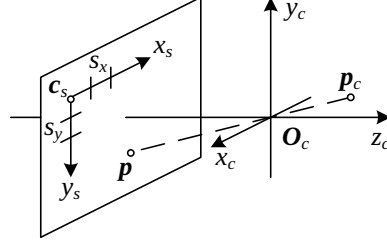


Figure 2.8 Projection of a 3D camera-centered point $\mathbf{p}_c$ onto the sensor planes at location $\mathbf{p}$. $\mathbf{O}_c$ is the optical center (nodal point), $\mathbf{c}_s$ is the 3D origin of the sensor plane coordinate system, and s_x and s_y are the pixel spacings.

Camera intrinsics

Once we have projected a 3D point through an ideal pinhole using a projection matrix, we must still transform the resulting coordinates according to the pixel sensor spacing and the relative position of the sensor plane to the origin. Figure 2.8 shows an illustration of the geometry involved. In this section, we first present a mapping from 2D pixel coordinates to 3D rays using a sensor homography $\mathbf{M}_s$, since this is easier to explain in terms of physically measurable quantities. We then relate these quantities to the more commonly used camera intrinsic matrix $\mathbf{K}$, which is used to map 3D camera-centered points $\mathbf{p}_c$ to 2D pixel coordinates $\tilde{\mathbf{x}}_s$.

Image sensors return pixel values indexed by integer *pixel coordinates* (x_s, y_s) , often with the coordinates starting at the upper-left corner of the image and moving down and to the right. (This convention is not obeyed by all imaging libraries, but the adjustment for other coordinate systems is straightforward.) To map pixel centers to 3D coordinates, we first scale the (x_s, y_s) values by the pixel spacings (s_x, s_y) (sometimes expressed in microns for solid-state sensors) and then describe the orientation of the sensor array relative to the camera projection center $\mathbf{O}_c$ with an origin $\mathbf{c}_s$ and a 3D rotation $\mathbf{R}_s$ (Figure 2.8).

The combined 2D to 3D projection can then be written as

$$\mathbf{p} = \begin{bmatrix} \mathbf{R}_s & \mathbf{c}_s \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix} = \mathbf{M}_s \tilde{\mathbf{x}}_s. \quad (2.53)$$

The first two columns of the 3×3 matrix $\mathbf{M}_s$ are the 3D vectors corresponding to unit steps in the image pixel array along the x_s and y_s directions, while the third column is the 3D

image array origin $\mathbf{c}_s$.

The matrix $\mathbf{M}_s$ is parameterized by eight unknowns: the three parameters describing the rotation $\mathbf{R}_s$, the three parameters describing the translation $\mathbf{c}_s$, and the two scale factors (s_x, s_y) . Note that we ignore here the possibility of *skew* between the two axes on the image plane, since solid-state manufacturing techniques render this negligible. In practice, unless we have accurate external knowledge of the sensor spacing or sensor orientation, there are only seven degrees of freedom, since the distance of the sensor from the origin cannot be teased apart from the sensor spacing, based on external image measurement alone.

However, estimating a camera model $\mathbf{M}_s$ with the required seven degrees of freedom (i.e., where the first two columns are orthogonal after an appropriate re-scaling) is impractical, so most practitioners assume a general 3×3 homogeneous matrix form.

The relationship between the 3D pixel center $\mathbf{p}$ and the 3D camera-centered point $\mathbf{p}_c$ is given by an unknown scaling s , $\mathbf{p} = s\mathbf{p}_c$. We can therefore write the complete projection between $\mathbf{p}_c$ and a homogeneous version of the pixel address $\tilde{\mathbf{x}}_s$ as

$$\tilde{\mathbf{x}}_s = \alpha \mathbf{M}_s^{-1} \mathbf{p}_c = \mathbf{K} \mathbf{p}_c. \quad (2.54)$$

The 3×3 matrix $\mathbf{K}$ is called the *calibration matrix* and describes the camera *intrinsics* (as opposed to the camera's orientation in space, which are called the *extrinsics*).

From the above discussion, we see that $\mathbf{K}$ has seven degrees of freedom in theory and eight degrees of freedom (the full dimensionality of a 3×3 homogeneous matrix) in practice. Why, then, do most textbooks on 3D computer vision and multi-view geometry (Faugeras 1993; Hartley and Zisserman 2004; Faugeras and Luong 2001) treat $\mathbf{K}$ as an upper-triangular matrix with five degrees of freedom?

While this is usually not made explicit in these books, it is because we cannot recover the full $\mathbf{K}$ matrix based on external measurement alone. When calibrating a camera (Section 11.1) based on external 3D points or other measurements (Tsai 1987), we end up estimating the intrinsic ($\mathbf{K}$) and extrinsic ($\mathbf{R}, \mathbf{t}$) camera parameters simultaneously using a series of measurements,

$$\tilde{\mathbf{x}}_s = \mathbf{K} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \mathbf{p}_w = \mathbf{P} \mathbf{p}_w, \quad (2.55)$$

where $\mathbf{p}_w$ are known 3D world coordinates and

$$\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}] \quad (2.56)$$

is known as the *camera matrix*. Inspecting this equation, we see that we can post-multiply $\mathbf{K}$ by $\mathbf{R}_1$ and pre-multiply $[\mathbf{R}|\mathbf{t}]$ by $\mathbf{R}_1^T$, and still end up with a valid calibration. Thus, it is impossible based on image measurements alone to know the true orientation of the sensor and the true camera intrinsics.

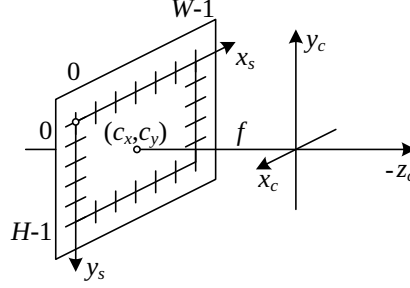


Figure 2.9 Simplified camera intrinsics showing the focal length f and the image center (c_x, c_y) . The image width and height are W and H .

The choice of an upper-triangular form for $\mathbf{K}$ seems to be conventional. Given a full 3×4 camera matrix $\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}]$, we can compute an upper-triangular $\mathbf{K}$ matrix using QR factorization (Golub and Van Loan 1996). (Note the unfortunate clash of terminologies: In matrix algebra textbooks, $\mathbf{R}$ represents an upper-triangular (right of the diagonal) matrix; in computer vision, $\mathbf{R}$ is an orthogonal rotation.)

There are several ways to write the upper-triangular form of $\mathbf{K}$. One possibility is

$$\mathbf{K} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad (2.57)$$

which uses independent focal lengths f_x and f_y for the sensor x and y dimensions. The entry s encodes any possible skew between the sensor axes due to the sensor not being mounted perpendicular to the optical axis and (c_x, c_y) denotes the image center expressed in pixel coordinates. The image center is also often called the *principal point* in the computer vision literature (Hartley and Zisserman 2004), although in optics, the principal points are 3D points usually inside the lens where the principal planes intersect the principal (optical) axis (Hecht 2015). Another possibility is

$$\mathbf{K} = \begin{bmatrix} f & s & c_x \\ 0 & af & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad (2.58)$$

where the *aspect ratio* a has been made explicit and a common focal length f is used.

In practice, for many applications an even simpler form can be obtained by setting $a = 1$

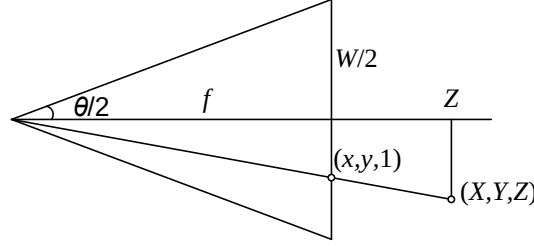


Figure 2.10 Central projection, showing the relationship between the 3D and 2D coordinates, $\mathbf{p}$ and $\mathbf{x}$, as well as the relationship between the focal length f , image width W , and the horizontal field of view θ_H .

and $s = 0$,

$$\mathbf{K} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}. \quad (2.59)$$

Often, setting the origin at roughly the center of the image, e.g., $(c_x, c_y) = (W/2, H/2)$, where W and H are the image width and height, respectively, can result in a perfectly usable camera model with a single unknown, i.e., the focal length f .

Figure 2.9 shows how these quantities can be visualized as part of a simplified imaging model. Note that now we have placed the image plane *in front* of the nodal point (projection center of the lens). The sense of the y -axis has also been flipped to get a coordinate system compatible with the way that most imaging libraries treat the vertical (row) coordinate.

A note on focal lengths

The issue of how to express focal lengths is one that often causes confusion in implementing computer vision algorithms and discussing their results. This is because the focal length depends on the units used to measure pixels.

If we number pixel coordinates using integer values, say $[0, W) \times [0, H)$, the focal length f and camera center (c_x, c_y) in (2.59) can be expressed as pixel values. How do these quantities relate to the more familiar focal lengths used by photographers?

Figure 2.10 illustrates the relationship between the focal length f , the sensor width W , and the horizontal field of view θ_H , which obey the formula

$$\tan \frac{\theta_H}{2} = \frac{W}{2f} \quad \text{or} \quad f = \frac{W}{2} \left[\tan \frac{\theta_H}{2} \right]^{-1}. \quad (2.60)$$

For a traditional 35mm film camera, whose active exposure area is $24\text{mm} \times 36\text{mm}$, we have $W = 36\text{mm}$, and hence f is also expressed in millimeters.⁴ For example, the “stock” lens that often comes with SLR (single lens reflex) cameras is 50mm, which is a good length, whereas 85mm is the standard for portrait photography. Since we work with digital images, however, it is more convenient to express W in pixels so that the focal length f can be used directly in the calibration matrix $\mathbf{K}$ as in (2.59).

Another possibility is to scale the pixel coordinates so that they go from $[-1, 1)$ along the longer image dimension and $[-a^{-1}, a^{-1})$ along the shorter axis, where $a \geq 1$ is the *image aspect ratio* (as opposed to the *sensor cell aspect ratio* introduced earlier). This can be accomplished using *modified normalized device coordinates*,

$$x'_s = (2x_s - W)/S \quad \text{and} \quad y'_s = (2y_s - H)/S, \quad \text{where} \quad S = \max(W, H). \quad (2.61)$$

This has the advantage that the focal length f and image center (c_x, c_y) become independent of the image resolution, which can be useful when using multi-resolution, image-processing algorithms, such as image pyramids (Section 3.5).⁵ The use of S instead of W also makes the focal length the same for landscape (horizontal) and portrait (vertical) pictures, as is the case in 35mm photography. (In some computer graphics textbooks and systems, normalized device coordinates go from $[-1, 1] \times [-1, 1]$, which requires the use of two different focal lengths to describe the camera intrinsics (Watt 1995).) Setting $S = W = 2$ in (2.60), we obtain the simpler (unitless) relationship

$$f^{-1} = \tan \frac{\theta_H}{2}. \quad (2.62)$$

The conversion between the various focal length representations is straightforward, e.g., to go from a unitless f to one expressed in pixels, multiply by $W/2$, while to convert from an f expressed in pixels to the equivalent 35mm focal length, multiply by 18mm.

Camera matrix

Now that we have shown how to parameterize the calibration matrix $\mathbf{K}$, we can put the camera intrinsics and extrinsics together to obtain a single 3×4 *camera matrix*

$$\mathbf{P} = \mathbf{K} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix}. \quad (2.63)$$

⁴35mm denotes the width of the film strip, of which 24mm is used for exposing each frame and the remaining 11mm for perforation and frame numbering.

⁵To make the conversion truly accurate after a downsampling step in a pyramid, floating point values of W and H would have to be maintained, as they can become non-integer if they are ever odd at a larger resolution in the pyramid.

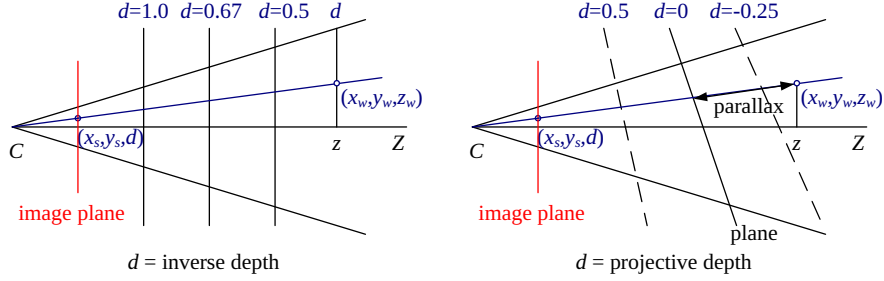


Figure 2.11 Regular disparity (inverse depth) and projective depth (parallax from a reference plane).

It is sometimes preferable to use an invertible 4×4 matrix, which can be obtained by not dropping the last row in the $\mathbf{P}$ matrix,

$$\tilde{\mathbf{P}} = \begin{bmatrix} \mathbf{K} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} = \tilde{\mathbf{K}}\mathbf{E}, \quad (2.64)$$

where $\mathbf{E}$ is a 3D rigid-body (Euclidean) transformation and $\tilde{\mathbf{K}}$ is the full-rank calibration matrix. The 4×4 camera matrix $\tilde{\mathbf{P}}$ can be used to map directly from 3D world coordinates $\bar{\mathbf{p}}_w = (x_w, y_w, z_w, 1)$ to screen coordinates (plus disparity), $\mathbf{x}_s = (x_s, y_s, 1, d)$,

$$\mathbf{x}_s \sim \tilde{\mathbf{P}}\bar{\mathbf{p}}_w, \quad (2.65)$$

where $\sim$ indicates equality up to scale. Note that after multiplication by $\tilde{\mathbf{P}}$, the vector is divided by the *third* element of the vector to obtain the normalized form $\mathbf{x}_s = (x_s, y_s, 1, d)$.

Plane plus parallax (projective depth)

In general, when using the 4×4 matrix $\tilde{\mathbf{P}}$, we have the freedom to remap the last row to whatever suits our purpose (rather than just being the “standard” interpretation of disparity as inverse depth). Let us re-write the last row of $\tilde{\mathbf{P}}$ as $\mathbf{p}_3 = s_3[\hat{\mathbf{n}}_0 | c_0]$, where $\|\hat{\mathbf{n}}_0\| = 1$. We then have the equation

$$d = \frac{s_3}{z}(\hat{\mathbf{n}}_0 \cdot \mathbf{p}_w + c_0), \quad (2.66)$$

where $z = \mathbf{p}_2 \cdot \bar{\mathbf{p}}_w = \mathbf{r}_z \cdot (\mathbf{p}_w - \mathbf{c})$ is the distance of $\mathbf{p}_w$ from the camera center C (2.25) along the optical axis Z (Figure 2.11). Thus, we can interpret d as the *projective disparity* or *projective depth* of a 3D scene point $\mathbf{p}_w$ from the *reference plane* $\hat{\mathbf{n}}_0 \cdot \mathbf{p}_w + c_0 = 0$ (Szeliski and Coughlan 1997; Szeliski and Golland 1999; Shade, Gortler *et al.* 1998; Baker,

Szeliski, and Anandan 1998). (The projective depth is also sometimes called *parallax* in reconstruction algorithms that use the term *plane plus parallax* (Kumar, Anandan, and Hanna 1994; Sawhney 1994).) Setting $\hat{\mathbf{n}}_0 = \mathbf{0}$ and $c_0 = 1$, i.e., putting the reference plane at infinity, results in the more standard $d = 1/z$ version of disparity (Okutomi and Kanade 1993).

Another way to see this is to invert the $\tilde{\mathbf{P}}$ matrix so that we can map pixels plus disparity directly back to 3D points,

$$\tilde{\mathbf{p}}_w = \tilde{\mathbf{P}}^{-1} \mathbf{x}_s. \quad (2.67)$$

In general, we can choose $\tilde{\mathbf{P}}$ to have whatever form is convenient, i.e., to sample space using an arbitrary projection. This can come in particularly handy when setting up multi-view stereo reconstruction algorithms, since it allows us to sweep a series of planes (Section 12.1.2) through space with a variable (projective) sampling that best matches the sensed image motions (Collins 1996; Szeliski and Golland 1999; Saito and Kanade 1999).

Mapping from one camera to another

What happens when we take two images of a 3D scene from different camera positions or orientations (Figure 2.12a)? Using the full rank 4×4 camera matrix $\tilde{\mathbf{P}} = \tilde{\mathbf{K}}\mathbf{E}$ from (2.64), we can write the projection from world to screen coordinates as

$$\tilde{\mathbf{x}}_0 \sim \tilde{\mathbf{K}}_0 \mathbf{E}_0 \mathbf{p} = \tilde{\mathbf{P}}_0 \mathbf{p}. \quad (2.68)$$

Assuming that we know the z-buffer or disparity value d_0 for a pixel in one image, we can compute the 3D point location $\mathbf{p}$ using

$$\mathbf{p} \sim \mathbf{E}_0^{-1} \tilde{\mathbf{K}}_0^{-1} \tilde{\mathbf{x}}_0 \quad (2.69)$$

and then project it into another image yielding

$$\tilde{\mathbf{x}}_1 \sim \tilde{\mathbf{K}}_1 \mathbf{E}_1 \mathbf{p} = \tilde{\mathbf{K}}_1 \mathbf{E}_1 \mathbf{E}_0^{-1} \tilde{\mathbf{K}}_0^{-1} \tilde{\mathbf{x}}_0 = \tilde{\mathbf{P}}_1 \tilde{\mathbf{P}}_0^{-1} \tilde{\mathbf{x}}_0 = \mathbf{M}_{10} \tilde{\mathbf{x}}_0. \quad (2.70)$$

Unfortunately, we do not usually have access to the depth coordinates of pixels in a regular photographic image. However, for a *planar scene*, as discussed above in (2.66), we can replace the last row of $\mathbf{P}_0$ in (2.64) with a general *plane equation*, $\hat{\mathbf{n}}_0 \cdot \mathbf{p} + c_0$, that maps points on the plane to $d_0 = 0$ values (Figure 2.12b). Thus, if we set $d_0 = 0$, we can ignore the last column of $\mathbf{M}_{10}$ in (2.70) and also its last row, since we do not care about the final z-buffer depth. The mapping Equation (2.70) thus reduces to

$$\tilde{\mathbf{x}}_1 \sim \tilde{\mathbf{H}}_{10} \tilde{\mathbf{x}}_0, \quad (2.71)$$

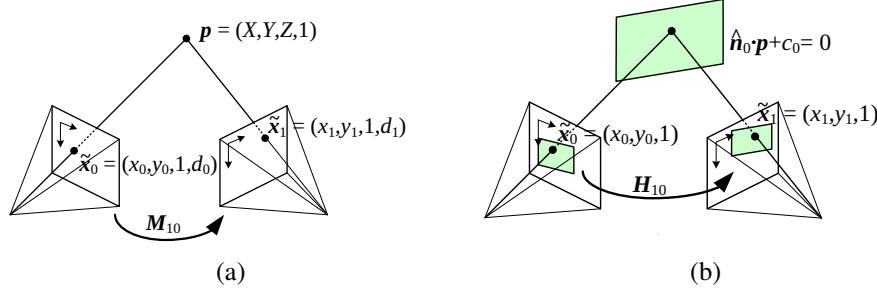


Figure 2.12 A point is projected into two images: (a) relationship between the 3D point coordinate $(X, Y, Z, 1)$ and the 2D projected point $(x, y, 1, d)$; (b) planar homography induced by points all lying on a common plane $\hat{\mathbf{n}}_0 \cdot \mathbf{p} + c_0 = 0$.

where $\tilde{\mathbf{H}}_{10}$ is a general 3×3 homography matrix and $\tilde{\mathbf{x}}_1$ and $\tilde{\mathbf{x}}_0$ are now 2D homogeneous coordinates (i.e., 3-vectors) (Szeliski 1996). This justifies the use of the 8-parameter homography as a general alignment model for mosaics of planar scenes (Mann and Picard 1994; Szeliski 1996).

The other special case where we do not need to know depth to perform inter-camera mapping is when the camera is undergoing pure rotation (Section 8.2.3), i.e., when $\mathbf{t}_0 = \mathbf{t}_1$. In this case, we can write

$$\tilde{\mathbf{x}}_1 \sim \mathbf{K}_1 \mathbf{R}_1 \mathbf{R}_0^{-1} \mathbf{K}_0^{-1} \tilde{\mathbf{x}}_0 = \mathbf{K}_1 \mathbf{R}_{10} \mathbf{K}_0^{-1} \tilde{\mathbf{x}}_0, \quad (2.72)$$

which again can be represented with a 3×3 homography. If we assume that the calibration matrices have known aspect ratios and centers of projection (2.59), this homography can be parameterized by the rotation amount and the two unknown focal lengths. This particular formulation is commonly used in image-stitching applications (Section 8.2.3).

Object-centered projection

When working with long focal length lenses, it often becomes difficult to reliably estimate the focal length from image measurements alone. This is because the focal length and the distance to the object are highly correlated and it becomes difficult to tease these two effects apart. For example, the change in scale of an object viewed through a zoom telephoto lens can either be due to a zoom change or to a motion towards the user. (This effect was put to dramatic use in some scenes of Alfred Hitchcock's film *Vertigo*, where the simultaneous change of zoom and camera motion produces a disquieting effect.)

This ambiguity becomes clearer if we write out the projection equation corresponding to

the simple calibration matrix $\mathbf{K}$ (2.59),

$$x_s = f \frac{\mathbf{r}_x \cdot \mathbf{p} + t_x}{\mathbf{r}_z \cdot \mathbf{p} + t_z} + c_x \quad (2.73)$$

$$y_s = f \frac{\mathbf{r}_y \cdot \mathbf{p} + t_y}{\mathbf{r}_z \cdot \mathbf{p} + t_z} + c_y, \quad (2.74)$$

where $\mathbf{r}_x$, $\mathbf{r}_y$, and $\mathbf{r}_z$ are the three rows of $\mathbf{R}$. If the distance to the object center $t_z \gg \|\mathbf{p}\|$ (the size of the object), the denominator is approximately t_z and the overall scale of the projected object depends on the ratio of f to t_z . It therefore becomes difficult to disentangle these two quantities.

To see this more clearly, let $\eta_z = t_z^{-1}$ and $s = \eta_z f$. We can then re-write the above equations as

$$x_s = s \frac{\mathbf{r}_x \cdot \mathbf{p} + t_x}{1 + \eta_z \mathbf{r}_z \cdot \mathbf{p}} + c_x \quad (2.75)$$

$$y_s = s \frac{\mathbf{r}_y \cdot \mathbf{p} + t_y}{1 + \eta_z \mathbf{r}_z \cdot \mathbf{p}} + c_y \quad (2.76)$$

(Szeliski and Kang 1994; Pighin, Hecker *et al.* 1998). The scale of the projection s can be reliably estimated if we are looking at a known object (i.e., the 3D coordinates $\mathbf{p}$ are known). The inverse distance η_z is now mostly decoupled from the estimates of s and can be estimated from the amount of *foreshortening* as the object rotates. Furthermore, as the lens becomes longer, i.e., the projection model becomes orthographic, there is no need to replace a perspective imaging model with an orthographic one, since the same equation can be used, with $\eta_z \rightarrow 0$ (as opposed to f and t_z both going to infinity). This allows us to form a natural link between orthographic reconstruction techniques such as factorization and their projective/perspective counterparts (Section 11.4.1).

2.1.5 Lens distortions

The above imaging models all assume that cameras obey a *linear* projection model where straight lines in the world result in straight lines in the image. (This follows as a natural consequence of linear matrix operations being applied to homogeneous coordinates.) Unfortunately, many wide-angle lenses have noticeable *radial distortion*, which manifests itself as a visible curvature in the projection of straight lines. (See Section 2.2.3 for a more detailed discussion of lens optics, including chromatic aberration.) Unless this distortion is taken into account, it becomes impossible to create highly accurate photorealistic reconstructions. For example, image mosaics constructed without taking radial distortion into account will often exhibit blurring due to the misregistration of corresponding features before pixel blending (Section 8.2).

Fortunately, compensating for radial distortion is not that difficult in practice. For most lenses, a simple quartic model of distortion can produce good results. Let (x_c, y_c) be the pixel coordinates obtained *after* perspective division but *before* scaling by focal length f and shifting by the image center (c_x, c_y) , i.e.,

$$\begin{aligned} x_c &= \frac{\mathbf{r}_x \cdot \mathbf{p} + t_x}{\mathbf{r}_z \cdot \mathbf{p} + t_z} \\ y_c &= \frac{\mathbf{r}_y \cdot \mathbf{p} + t_y}{\mathbf{r}_z \cdot \mathbf{p} + t_z}. \end{aligned} \quad (2.77)$$

The radial distortion model says that coordinates in the observed images are displaced towards (*barrel* distortion) or away (*pincushion* distortion) from the image center by an amount proportional to their radial distance (Figure 2.13a–b).⁶ The simplest radial distortion models use low-order polynomials, e.g.,

$$\begin{aligned} \hat{x}_c &= x_c(1 + \kappa_1 r_c^2 + \kappa_2 r_c^4) \\ \hat{y}_c &= y_c(1 + \kappa_1 r_c^2 + \kappa_2 r_c^4), \end{aligned} \quad (2.78)$$

where $r_c^2 = x_c^2 + y_c^2$ and κ_1 and κ_2 are called the *radial distortion parameters*.⁷ This model, which also includes a *tangential* component to account for lens decentering, was first proposed in the photogrammetry literature by Brown (1966), and so is sometimes called the *Brown* or *Brown–Conrady* model. However, the tangential components of the distortion are usually ignored because they can lead to less stable estimates (Zhang 2000).

After the radial distortion step, the final pixel coordinates can be computed using

$$\begin{aligned} x_s &= f\hat{x}_c + c_x \\ y_s &= f\hat{y}_c + c_y. \end{aligned} \quad (2.79)$$

A variety of techniques can be used to estimate the radial distortion parameters for a given lens, as discussed in Section 11.1.4.

Sometimes the above simplified model does not model the true distortions produced by complex lenses accurately enough (especially at very wide angles). A more complete analytic model also includes *tangential distortions* and *decentering distortions* (Slama 1980).

Fisheye lenses (Figure 2.13c) require a model that differs from traditional polynomial models of radial distortion. Fisheye lenses behave, to a first approximation, as *equi-distance*

⁶Anamorphic lenses, which are widely used in feature film production, do not follow this radial distortion model. Instead, they can be thought of, to a first approximation, as inducing different vertical and horizontal scaling, i.e., non-square pixels.

⁷Sometimes the relationship between x_c and $\hat{x}_c$ is expressed the other way around, i.e., $x_c = \hat{x}_c(1 + \kappa_1 \hat{r}_c^2 + \kappa_2 \hat{r}_c^4)$. This is convenient if we map image pixels into (warped) rays by dividing through by f . We can then undistort the rays and have true 3D rays in space.

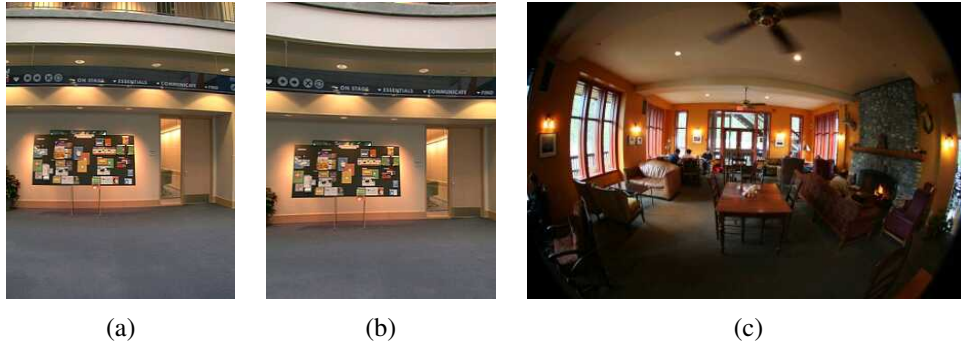


Figure 2.13 *Radial lens distortions: (a) barrel, (b) pincushion, and (c) fisheye. The fisheye image spans almost 180° from side-to-side.*

projectors of angles away from the optical axis (Xiong and Turkowski 1997),

$$r = f\theta, \quad (2.80)$$

which is the same as the *polar projection* described by Equations (8.55–8.57). Because of the mostly linear mapping between distance from the center (pixels) and viewing angle, such lenses are sometimes called *f-theta lenses*, which is likely where the popular RICOH THETA 360° camera got its name. Xiong and Turkowski (1997) describe how this model can be extended with the addition of an extra quadratic correction in ϕ and how the unknown parameters (center of projection, scaling factor s , etc.) can be estimated from a set of overlapping fisheye images using a direct (intensity-based) non-linear minimization algorithm.

For even larger, less regular distortions, a parametric distortion model using splines may be necessary (Goshtasby 1989). If the lens does not have a single center of projection, it may become necessary to model the 3D *line* (as opposed to *direction*) corresponding to each pixel separately (Gremban, Thorpe, and Kanade 1988; Champleboux, Lavallée *et al.* 1992a; Grossberg and Nayar 2001; Sturm and Ramalingam 2004; Tardif, Sturm *et al.* 2009). Some of these techniques are described in more detail in Section 11.1.4, which discusses how to calibrate lens distortions.

There is one subtle issue associated with the simple radial distortion model that is often glossed over. We have introduced a non-linearity between the perspective projection and final sensor array projection steps. Therefore, we cannot, in general, post-multiply an arbitrary 3×3 matrix $\mathbf{K}$ with a rotation to put it into upper-triangular form and absorb this into the global rotation. However, this situation is not as bad as it may at first appear. For many applications, keeping the simplified diagonal form of (2.59) is still an adequate model. Furthermore, if we correct radial and other distortions to an accuracy where straight lines are preserved, we have

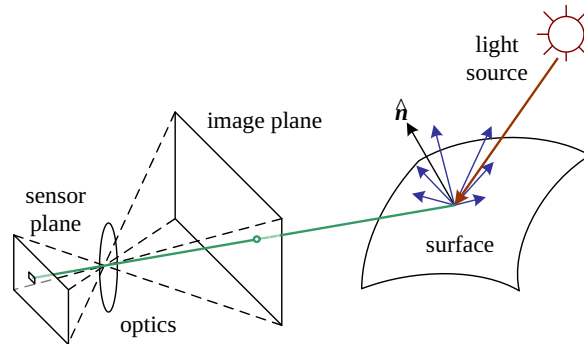


Figure 2.14 A simplified model of photometric image formation. Light is emitted by one or more light sources and is then reflected from an object's surface. A portion of this light is directed towards the camera. This simplified model ignores multiple reflections, which often occur in real-world scenes.

essentially converted the sensor back into a linear imager and the previous decomposition still applies.

2.2 Photometric image formation

In modeling the image formation process, we have described how 3D geometric features in the world are projected into 2D features in an image. However, images are not composed of 2D features. Instead, they are made up of discrete color or intensity values. Where do these values come from? How do they relate to the lighting in the environment, surface properties and geometry, camera optics, and sensor properties (Figure 2.14)? In this section, we develop a set of models to describe these interactions and formulate a generative process of image formation. A more detailed treatment of these topics can be found in textbooks on computer graphics and image synthesis (Cohen and Wallace 1993; Sillion and Puech 1994; Watt 1995; Glassner 1995; Weyrich, Lawrence *et al.* 2009; Hughes, van Dam *et al.* 2013; Marschner and Shirley 2015).

2.2.1 Lighting

Images cannot exist without light. To produce an image, the scene must be illuminated with one or more light sources. (Certain modalities such as fluorescence microscopy and X-ray tomography do not fit this model, but we do not deal with them in this book.) Light sources can generally be divided into point and area light sources.

A point light source originates at a single location in space (e.g., a small light bulb), potentially at infinity (e.g., the Sun). (Note that for some applications such as modeling soft shadows (*penumbras*), the Sun may have to be treated as an area light source.) In addition to its location, a point light source has an intensity and a color spectrum, i.e., a distribution over wavelengths $L(\lambda)$. The intensity of a light source falls off with the square of the distance between the source and the object being lit, because the same light is being spread over a larger (spherical) area. A light source may also have a directional falloff (dependence), but we ignore this in our simplified model.

Area light sources are more complicated. A simple area light source such as a fluorescent ceiling light fixture with a diffuser can be modeled as a finite rectangular area emitting light equally in all directions (Cohen and Wallace 1993; Sillion and Puech 1994; Glassner 1995). When the distribution is strongly directional, a four-dimensional lightfield can be used instead (Ashdown 1993).

A more complex light distribution that approximates, say, the incident illumination on an object sitting in an outdoor courtyard, can often be represented using an *environment map* (Greene 1986) (originally called a *reflection map* (Blinn and Newell 1976)). This representation maps incident light directions $\hat{\mathbf{v}}$ to color values (or wavelengths, λ),

$$L(\hat{\mathbf{v}}; \lambda), \quad (2.81)$$

and is equivalent to assuming that all light sources are at infinity. Environment maps can be represented as a collection of cubical faces (Greene 1986), as a single longitude–latitude map (Blinn and Newell 1976), or as the image of a reflecting sphere (Watt 1995). A convenient way to get a rough model of a real-world environment map is to take an image of a reflective mirrored sphere (sometimes accompanied by a darker sphere to capture highlights) and to unwrap this image onto the desired environment map (Debevec 1998). Watt (1995) gives a nice discussion of environment mapping, including the formulas needed to map directions to pixels for the three most commonly used representations.

2.2.2 Reflectance and shading

When light hits an object's surface, it is scattered and reflected (Figure 2.15a). Many different models have been developed to describe this interaction. In this section, we first describe the most general form, the bidirectional reflectance distribution function, and then look at some more specialized models, including the diffuse, specular, and Phong shading models. We also discuss how these models can be used to compute the *global illumination* corresponding to a scene.

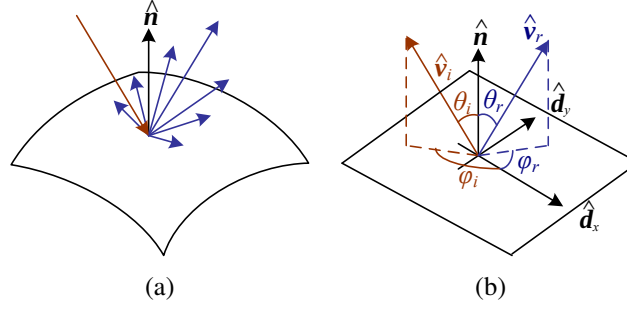


Figure 2.15 (a) Light scatters when it hits a surface. (b) The bidirectional reflectance distribution function (BRDF) $f(\theta_i, \phi_i, \theta_r, \phi_r)$ is parameterized by the angles that the incident, $\hat{v}_i$, and reflected, $\hat{v}_r$, light ray directions make with the local surface coordinate frame $(\hat{d}_x, \hat{d}_y, \hat{n})$.

The Bidirectional Reflectance Distribution Function (BRDF)

The most general model of light scattering is the *bidirectional reflectance distribution function* (BRDF).⁸ Relative to some local coordinate frame on the surface, the BRDF is a four-dimensional function that describes how much of each wavelength arriving at an *incident* direction $\hat{v}_i$ is emitted in a *reflected* direction $\hat{v}_r$ (Figure 2.15b). The function can be written in terms of the angles of the incident and reflected directions relative to the surface frame as

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r; \lambda). \quad (2.82)$$

The BRDF is *reciprocal*, i.e., because of the physics of light transport, you can interchange the roles of $\hat{v}_i$ and $\hat{v}_r$ and still get the same answer (this is sometimes called *Helmholtz reciprocity*).

Most surfaces are *isotropic*, i.e., there are no preferred directions on the surface as far as light transport is concerned. (The exceptions are *anisotropic* surfaces such as brushed (scratched) aluminum, where the reflectance depends on the light orientation relative to the direction of the scratches.) For an isotropic material, we can simplify the BRDF to

$$f_r(\theta_i, \theta_r, |\phi_r - \phi_i|; \lambda) \quad \text{or} \quad f_r(\hat{v}_i, \hat{v}_r, \hat{n}; \lambda), \quad (2.83)$$

as the quantities θ_i , θ_r , and $\phi_r - \phi_i$ can be computed from the directions $\hat{v}_i$, $\hat{v}_r$, and $\hat{n}$.

⁸Actually, even more general models of light transport exist, including some that model spatial variation along the surface, sub-surface scattering, and atmospheric effects—see Section 13.7.1—(Dorsey, Rushmeier, and Sillion 2007; Weyrich, Lawrence *et al.* 2009).



Figure 2.16 This close-up of a statue shows both diffuse (smooth shading) and specular (shiny highlight) reflection, as well as darkening in the grooves and creases due to reduced light visibility and interreflections. (Photo courtesy of the Caltech Vision Lab, <http://www.vision.caltech.edu/archive.html>.)

To calculate the amount of light exiting a surface point $\mathbf{p}$ in a direction $\hat{\mathbf{v}}_r$ under a given lighting condition, we integrate the product of the incoming light $L_i(\hat{\mathbf{v}}_i; \lambda)$ with the BRDF (some authors call this step a *convolution*). Taking into account the *foreshortening* factor $\cos^+ \theta_i$, we obtain

$$L_r(\hat{\mathbf{v}}_r; \lambda) = \int L_i(\hat{\mathbf{v}}_i; \lambda) f_r(\hat{\mathbf{v}}_i, \hat{\mathbf{v}}_r, \hat{\mathbf{n}}; \lambda) \cos^+ \theta_i d\hat{\mathbf{v}}_i, \quad (2.84)$$

where

$$\cos^+ \theta_i = \max(0, \cos \theta_i). \quad (2.85)$$

If the light sources are discrete (a finite number of point light sources), we can replace the integral with a summation,

$$L_r(\hat{\mathbf{v}}_r; \lambda) = \sum_i L_i(\lambda) f_r(\hat{\mathbf{v}}_i, \hat{\mathbf{v}}_r, \hat{\mathbf{n}}; \lambda) \cos^+ \theta_i. \quad (2.86)$$

BRDFs for a given surface can be obtained through physical modeling (Torrance and Sparrow 1967; Cook and Torrance 1982; Glassner 1995), heuristic modeling (Phong 1975; Lafortune, Foo *et al.* 1997), or through empirical observation (Ward 1992; Westin, Arvo, and Torrance 1992; Dana, van Ginneken *et al.* 1999; Marschner, Westin *et al.* 2000; Matusik, Pfister *et al.* 2003; Dorsey, Rushmeier, and Sillion 2007; Weyrich, Lawrence *et al.* 2009; Shi, Mo *et al.* 2019).⁹ Typical BRDFs can often be split into their *diffuse* and *specular* components, as described below.

⁹See <http://www1.cs.columbia.edu/CAVE/software/curet> for a database of some empirically sampled BRDFs.

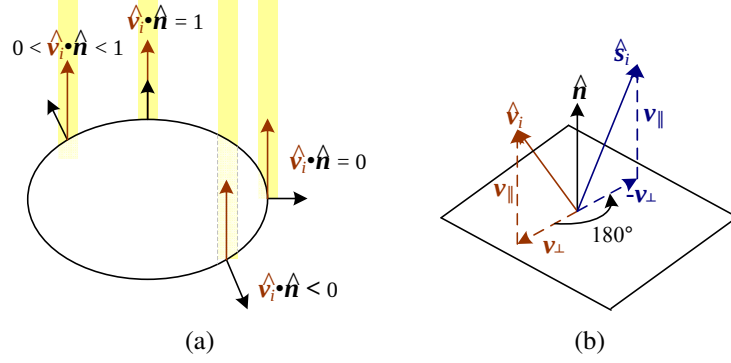


Figure 2.17 (a) The diminution of returned light caused by foreshortening depends on $\hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}$, the cosine of the angle between the incident light direction $\hat{\mathbf{v}}_i$ and the surface normal $\hat{\mathbf{n}}$. (b) Mirror (specular) reflection: The incident light ray direction $\hat{\mathbf{v}}_i$ is reflected onto the specular direction $\hat{\mathbf{s}}_i$ around the surface normal $\hat{\mathbf{n}}$.

Diffuse reflection

The diffuse component (also known as *Lambertian* or *matte* reflection) scatters light uniformly in all directions and is the phenomenon we most normally associate with *shading*, e.g., the smooth (non-shiny) variation of intensity with surface normal that is seen when observing a statue (Figure 2.16). Diffuse reflection also often imparts a strong *body color* to the light, as it is caused by selective absorption and re-emission of light inside the object's material (Shafer 1985; Glassner 1995).

While light is scattered uniformly in all directions, i.e., the BRDF is constant,

$$f_d(\hat{\mathbf{v}}_i, \hat{\mathbf{v}}_r, \hat{\mathbf{n}}; \lambda) = f_d(\lambda), \quad (2.87)$$

the amount of light depends on the angle between the incident light direction and the surface normal θ_i . This is because the surface area exposed to a given amount of light becomes larger at oblique angles, becoming completely self-shadowed as the outgoing surface normal points away from the light (Figure 2.17a). (Think about how you orient yourself towards the Sun or fireplace to get maximum warmth and how a flashlight projected obliquely against a wall is less bright than one pointing directly at it.) The *shading equation* for diffuse reflection can thus be written as

$$L_d(\hat{\mathbf{v}}_r; \lambda) = \sum_i L_i(\lambda) f_d(\lambda) \cos^+ \theta_i = \sum_i L_i(\lambda) f_d(\lambda) [\hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}]^+, \quad (2.88)$$

where

$$[\hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}]^+ = \max(0, \hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}). \quad (2.89)$$

Specular reflection

The second major component of a typical BRDF is *specular* (gloss or highlight) reflection, which depends strongly on the direction of the outgoing light. Consider light reflecting off a mirrored surface (Figure 2.17b). Incident light rays are reflected in a direction that is rotated by 180° around the surface normal $\hat{\mathbf{n}}$. Using the same notation as in Equations (2.29–2.30), we can compute the *specular reflection* direction $\hat{\mathbf{s}}_i$ as

$$\hat{\mathbf{s}}_i = \mathbf{v}_{\parallel} - \mathbf{v}_{\perp} = (2\hat{\mathbf{n}}\hat{\mathbf{n}}^T - \mathbf{I})\mathbf{v}_i. \quad (2.90)$$

The amount of light reflected in a given direction $\hat{\mathbf{v}}_r$ thus depends on the angle $\theta_s = \cos^{-1}(\hat{\mathbf{v}}_r \cdot \hat{\mathbf{s}}_i)$ between the view direction $\hat{\mathbf{v}}_r$ and the specular direction $\hat{\mathbf{s}}_i$. For example, the Phong (1975) model uses a power of the cosine of the angle,

$$f_s(\theta_s; \lambda) = k_s(\lambda) \cos^{k_e} \theta_s, \quad (2.91)$$

while the Torrance and Sparrow (1967) micro-facet model uses a Gaussian,

$$f_s(\theta_s; \lambda) = k_s(\lambda) \exp(-c_s^2 \theta_s^2). \quad (2.92)$$

Larger exponents k_e (or inverse Gaussian widths c_s) correspond to more specular surfaces with distinct highlights, while smaller exponents better model materials with softer gloss.

Phong shading

Phong (1975) combined the diffuse and specular components of reflection with another term, which he called the *ambient illumination*. This term accounts for the fact that objects are generally illuminated not only by point light sources but also by a general diffuse illumination corresponding to inter-reflection (e.g., the walls in a room) or distant sources, such as the blue sky. In the Phong model, the ambient term does not depend on surface orientation, but depends on the color of both the ambient illumination $L_a(\lambda)$ and the object $k_a(\lambda)$,

$$f_a(\lambda) = k_a(\lambda)L_a(\lambda). \quad (2.93)$$

Putting all of these terms together, we arrive at the *Phong shading* model,

$$L_r(\hat{\mathbf{v}}_r; \lambda) = k_a(\lambda)L_a(\lambda) + k_d(\lambda) \sum_i L_i(\lambda)[\hat{\mathbf{v}}_i \cdot \hat{\mathbf{n}}]^+ + k_s(\lambda) \sum_i L_i(\lambda)(\hat{\mathbf{v}}_r \cdot \hat{\mathbf{s}}_i)^{k_e}. \quad (2.94)$$

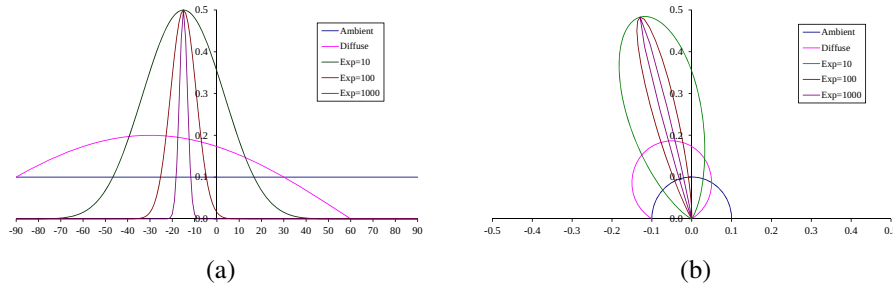


Figure 2.18 Cross-section through a Phong shading model BRDF for a fixed incident illumination direction: (a) component values as a function of angle away from surface normal; (b) polar plot. The value of the Phong exponent k_e is indicated by the “Exp” labels and the light source is at an angle of 30° away from the normal.

Figure 2.18 shows a typical set of Phong shading model components as a function of the angle away from the surface normal (in a plane containing both the lighting direction and the viewer).

Typically, the ambient and diffuse reflection color distributions $k_a(\lambda)$ and $k_d(\lambda)$ are the same, since they are both due to sub-surface scattering (body reflection) inside the surface material (Shafer 1985). The specular reflection distribution $k_s(\lambda)$ is often uniform (white), since it is caused by interface reflections that do not change the light color. (The exception to this is emphmetallic materials, such as copper, as opposed to the more common *dielectric* materials, such as plastics.)

The ambient illumination $L_a(\lambda)$ often has a different color cast from the direct light sources $L_i(\lambda)$, e.g., it may be blue for a sunny outdoor scene or yellow for an interior lit with candles or incandescent lights. (The presence of ambient sky illumination in shadowed areas is what often causes shadows to appear bluer than the corresponding lit portions of a scene). Note also that the diffuse component of the Phong model (or of any shading model) depends on the angle of the *incoming* light source $\hat{\mathbf{v}}_i$, while the specular component depends on the relative angle between the viewer $\mathbf{v}_r$ and the specular reflection direction $\hat{\mathbf{s}}_i$ (which itself depends on the incoming light direction $\hat{\mathbf{v}}_i$ and the surface normal $\hat{\mathbf{n}}$).

The Phong shading model has been superseded in terms of physical accuracy by newer models in computer graphics, including the model developed by Cook and Torrance (1982) based on the original micro-facet model of Torrance and Sparrow (1967). While, initially, computer graphics hardware implemented the Phong model, the advent of programmable pixel shaders has made the use of more complex models feasible.

Di-chromatic reflection model

The **Torrance and Sparrow (1967)** model of reflection also forms the basis of Shafer's (1985) *di-chromatic reflection model*, which states that the apparent color of a uniform material lit from a single source depends on the sum of two terms,

$$L_r(\hat{\mathbf{v}}_r; \lambda) = L_i(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}}; \lambda) + L_b(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}}; \lambda) \quad (2.95)$$

$$= c_i(\lambda)m_i(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}}) + c_b(\lambda)m_b(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}}), \quad (2.96)$$

i.e., the radiance of the light reflected at the *interface*, L_i , and the radiance reflected at the *surface body*, L_b . Each of these, in turn, is a simple product between a relative power spectrum $c(\lambda)$, which depends only on wavelength, and a magnitude $m(\hat{\mathbf{v}}_r, \hat{\mathbf{v}}_i, \hat{\mathbf{n}})$, which depends only on geometry. (This model can easily be derived from a generalized version of Phong's model by assuming a single light source and no ambient illumination, and rearranging terms.) The di-chromatic model has been successfully used in computer vision to segment specular colored objects with large variations in shading (**Klinker 1993**) and has inspired local two-color models for applications such as Bayer pattern demosaicing (**Bennett, Uyttendaele et al. 2006**).

Global illumination (ray tracing and radiosity)

The simple shading model presented thus far assumes that light rays leave the light sources, bounce off surfaces visible to the camera, thereby changing in intensity or color, and arrive at the camera. In reality, light sources can be shadowed by occluders and rays can bounce multiple times around a scene while making their trip from a light source to the camera.

Two methods have traditionally been used to model such effects. If the scene is mostly specular (the classic example being scenes made of glass objects and mirrored or highly polished balls), the preferred approach is *ray tracing* or *path tracing* (**Glassner 1995; Akenine-Möller and Haines 2002; Marschner and Shirley 2015**), which follows individual rays from the camera across multiple bounces towards the light sources (or vice versa). If the scene is composed mostly of uniform albedo simple geometry illuminators and surfaces, *radiosity* (*global illumination*) techniques are preferred (**Cohen and Wallace 1993; Sillion and Puech 1994; Glassner 1995**). Combinations of the two techniques have also been developed (**Wallace, Cohen, and Greenberg 1987**), as well as more general *light transport* techniques for simulating effects such as the *caustics* cast by rippling water.

The basic ray tracing algorithm associates a light ray with each pixel in the camera image and finds its intersection with the nearest surface. A *primary* contribution can then be computed using the simple shading equations presented previously (e.g., Equation (2.94))

for all light sources that are visible for that surface element. (An alternative technique for computing which surfaces are illuminated by a light source is to compute a *shadow map*, or *shadow buffer*, i.e., a rendering of the scene from the light source's perspective, and then compare the depth of pixels being rendered with the map (Williams 1983; Akenine-Möller and Haines 2002).) Additional *secondary* rays can then be cast along the specular direction towards other objects in the scene, keeping track of any attenuation or color change that the specular reflection induces.

Radiosity works by associating lightness values with rectangular surface areas in the scene (including area light sources). The amount of light interchanged between any two (mutually visible) areas in the scene can be captured as a *form factor*, which depends on their relative orientation and surface reflectance properties, as well as the $1/r^2$ fall-off as light is distributed over a larger effective sphere the further away it is (Cohen and Wallace 1993; Sillion and Puech 1994; Glassner 1995). A large linear system can then be set up to solve for the final lightness of each area patch, using the light sources as the forcing function (right-hand side). Once the system has been solved, the scene can be rendered from any desired point of view. Under certain circumstances, it is possible to recover the global illumination in a scene from photographs using computer vision techniques (Yu, Debevec *et al.* 1999).

The basic radiosity algorithm does not take into account certain *near field* effects, such as the darkening inside corners and scratches, or the limited ambient illumination caused by partial shadowing from other surfaces. Such effects have been exploited in a number of computer vision algorithms (Nayar, Ikeuchi, and Kanade 1991; Langer and Zucker 1994).

While all of these global illumination effects can have a strong effect on the appearance of a scene, and hence its 3D interpretation, they are not covered in more detail in this book. (But see Section 13.7.1 for a discussion of recovering BRDFs from real scenes and objects.)

2.2.3 Optics

Once the light from a scene reaches the camera, it must still pass through the lens before reaching the analog or digital sensor. For many applications, it suffices to treat the lens as an ideal pinhole that simply projects all rays through a common center of projection (Figures 2.8 and 2.9).

However, if we want to deal with issues such as focus, exposure, vignetting, and aberration, we need to develop a more sophisticated model, which is where the study of *optics* comes in (Möller 1988; Ray 2002; Hecht 2015).

Figure 2.19 shows a diagram of the most basic lens model, i.e., the *thin lens* composed of a single piece of glass with very low, equal curvature on both sides. According to the *lens law* (which can be derived using simple geometric arguments on light ray refraction), the

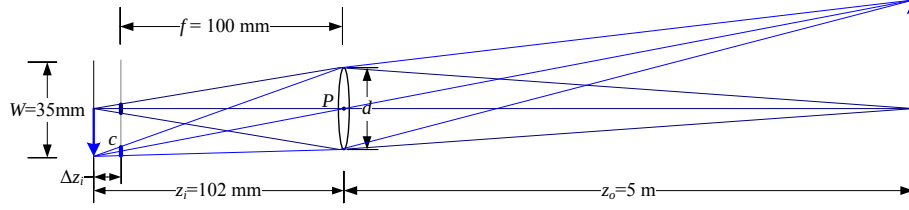


Figure 2.19 A thin lens of focal length f focuses the light from a plane at a distance z_o in front of the lens onto a plane at a distance z_i behind the lens, where $\frac{1}{z_o} + \frac{1}{z_i} = \frac{1}{f}$. If the focal plane (vertical gray line next to c) is moved forward, the images are no longer in focus and the circle of confusion c (small thick line segments) depends on the distance of the image plane motion Δz_i relative to the lens aperture diameter d . The field of view (f.o.v.) depends on the ratio between the sensor width W and the focal length f (or, more precisely, the focusing distance z_i , which is usually quite close to f).

relationship between the distance to an object z_o and the distance behind the lens at which a focused image is formed z_i can be expressed as

$$\frac{1}{z_o} + \frac{1}{z_i} = \frac{1}{f}, \quad (2.97)$$

where f is called the *focal length* of the lens. If we let $z_o \rightarrow \infty$, i.e., we adjust the lens (move the image plane) so that objects at infinity are in focus, we get $z_i = f$, which is why we can think of a lens of focal length f as being equivalent (to a first approximation) to a pinhole at a distance f from the focal plane (Figure 2.10), whose field of view is given by (2.60).

If the focal plane is moved away from its proper in-focus setting of z_i (e.g., by twisting the focus ring on the lens), objects at z_o are no longer in focus, as shown by the gray plane in Figure 2.19. The amount of misfocus is measured by the *circle of confusion* c (shown as short thick blue line segments on the gray plane).¹⁰ The equation for the circle of confusion can be derived using similar triangles; it depends on the distance of travel in the focal plane Δz_i relative to the original focus distance z_i and the diameter of the aperture d (see Exercise 2.4).

The allowable depth variation in the scene that limits the circle of confusion to an acceptable number is commonly called the *depth of field* and is a function of both the focus distance and the aperture, as shown diagrammatically by many lens markings (Figure 2.20). Since this

¹⁰If the aperture is not completely circular, e.g., if it is caused by a hexagonal diaphragm, it is sometimes possible to see this effect in the actual blur function (Levin, Fergus *et al.* 2007; Joshi, Szeliski, and Kriegman 2008) or in the “glints” that are seen when shooting into the Sun.

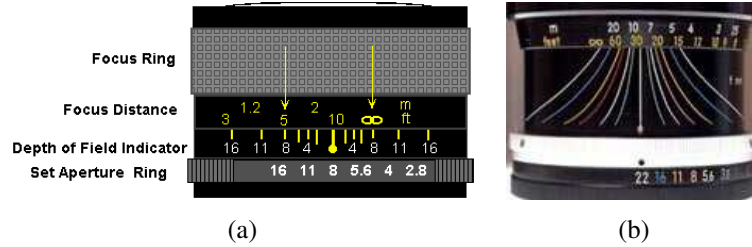


Figure 2.20 Regular and zoom lens depth of field indicators.

depth of field depends on the aperture diameter d , we also have to know how this varies with the commonly displayed f -number, which is usually denoted as $f/\#$ or N and is defined as

$$f/\# = N = \frac{f}{d}, \quad (2.98)$$

where the focal length f and the aperture diameter d are measured in the same unit (say, millimeters).

The usual way to write the f-number is to replace the $\#$ in $f/\#$ with the actual number, i.e., $f/1.4, f/2, f/2.8, \dots, f/22$. (Alternatively, we can say $N = 1.4$, etc.) An easy way to interpret these numbers is to notice that dividing the focal length by the f-number gives us the diameter d , so these are just formulas for the aperture diameter.¹¹

Notice that the usual progression for f-numbers is in *full stops*, which are multiples of $\sqrt{2}$, since this corresponds to doubling the area of the entrance pupil each time a smaller f-number is selected. (This doubling is also called changing the exposure by one *exposure value* or EV. It has the same effect on the amount of light reaching the sensor as doubling the exposure duration, e.g., from $1/250$ to $1/125$; see Exercise 2.5.)

Now that you know how to convert between f-numbers and aperture diameters, you can construct your own plots for the depth of field as a function of focal length f , circle of confusion c , and focus distance z_o , as explained in Exercise 2.4, and see how well these match what you observe on actual lenses, such as those shown in Figure 2.20.

Of course, real lenses are not infinitely thin and therefore suffer from geometric aberrations, unless compound elements are used to correct for them. The classic five *Seidel aberrations*, which arise when using *third-order optics*, include spherical aberration, coma, astigmatism, curvature of field, and distortion (Möller 1988; Ray 2002; Hecht 2015).

¹¹This also explains why, with zoom lenses, the f-number varies with the current zoom (focal length) setting.

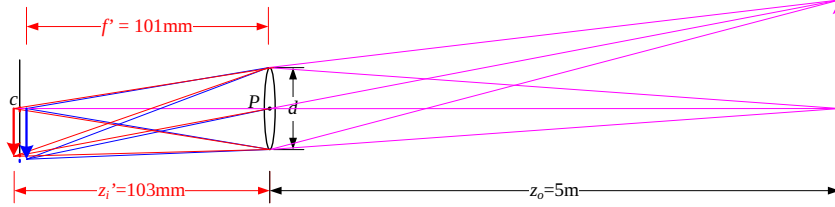


Figure 2.21 In a lens subject to chromatic aberration, light at different wavelengths (e.g., the red and blue arrows) is focused with a different focal length f' and hence a different depth z_i' , resulting in both a geometric (in-plane) displacement and a loss of focus.

Chromatic aberration

Because the index of refraction for glass varies slightly as a function of wavelength, simple lenses suffer from *chromatic aberration*, which is the tendency for light of different colors to focus at slightly different distances (and hence also with slightly different magnification factors), as shown in Figure 2.21. The wavelength-dependent magnification factor, i.e., the *transverse chromatic aberration*, can be modeled as a per-color radial distortion (Section 2.1.5) and, hence, calibrated using the techniques described in Section 11.1.4. The wavelength-dependent blur caused by *longitudinal chromatic aberration* can be calibrated using techniques described in Section 10.1.4. Unfortunately, the blur induced by longitudinal aberration can be harder to undo, as higher frequencies can get strongly attenuated and hence hard to recover.

To reduce chromatic and other kinds of aberrations, most photographic lenses today are *compound lenses* made of different glass elements (with different coatings). Such lenses can no longer be modeled as having a single *nodal point* P through which all of the rays must pass (when approximating the lens with a pinhole model). Instead, these lenses have both a *front nodal point*, through which the rays enter the lens, and a *rear nodal point*, through which they leave on their way to the sensor. In practice, only the location of the front nodal point is of interest when performing careful camera calibration, e.g., when determining the point around which to rotate to capture a parallax-free panorama (see Section 8.2.3 and Littlefield (2006) and Houghton (2013)).

Not all lenses, however, can be modeled as having a single nodal point. In particular, very wide-angle lenses such as fisheye lenses (Section 2.1.5) and certain *catadioptric* imaging systems consisting of lenses and curved mirrors (Baker and Nayar 1999) do not have a single point through which all of the acquired light rays pass. In such cases, it is preferable to

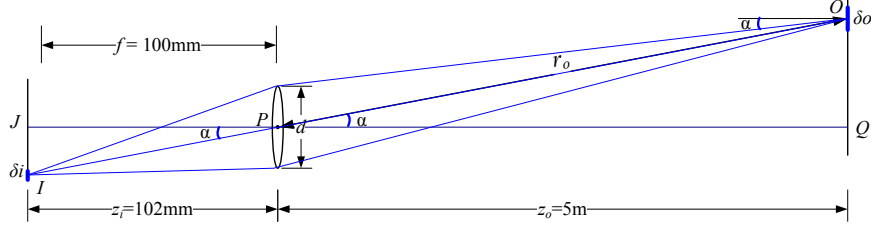


Figure 2.22 The amount of light hitting a pixel of surface area δi depends on the square of the ratio of the aperture diameter d to the focal length f , as well as the fourth power of the off-axis angle α cosine, $\cos^4 \alpha$.

explicitly construct a mapping function (look-up table) between pixel coordinates and 3D rays in space (Gremban, Thorpe, and Kanade 1988; Champlébourg, Lavallée *et al.* 1992a; Grossberg and Nayar 2001; Sturm and Ramalingam 2004; Tardif, Sturm *et al.* 2009), as mentioned in Section 2.1.5.

Vignetting

Another property of real-world lenses is *vignetting*, which is the tendency for the brightness of the image to fall off towards the edge of the image.

Two kinds of phenomena usually contribute to this effect (Ray 2002). The first is called *natural vignetting* and is due to the foreshortening in the object surface, projected pixel, and lens aperture, as shown in Figure 2.22. Consider the light leaving the object surface patch of size δo located at an *off-axis angle* α . Because this patch is foreshortened with respect to the camera lens, the amount of light reaching the lens is reduced by a factor $\cos \alpha$. The amount of light reaching the lens is also subject to the usual $1/r^2$ fall-off; in this case, the distance $r_o = z_o / \cos \alpha$. The actual area of the aperture through which the light passes is foreshortened by an additional factor $\cos \alpha$, i.e., the aperture as seen from point O is an ellipse of dimensions $d \times d \cos \alpha$. Putting all of these factors together, we see that the amount of light leaving O and passing through the aperture on its way to the image pixel located at I is proportional to

$$\frac{\delta o \cos \alpha}{r_o^2} \pi \left(\frac{d}{2} \right)^2 \cos \alpha = \delta o \frac{\pi d^2}{4 z_o^2} \cos^4 \alpha. \quad (2.99)$$

Since triangles $\triangle OPQ$ and $\triangle IPJ$ are similar, the projected areas of the object surface δo and image pixel δi are in the same (squared) ratio as $z_o : z_i$,

$$\frac{\delta o}{\delta i} = \frac{z_o^2}{z_i^2}. \quad (2.100)$$

Putting these together, we obtain the final relationship between the amount of light reaching pixel i and the aperture diameter d , the focusing distance $z_i \approx f$, and the off-axis angle α ,

$$\delta o \frac{\pi d^2}{4 z_o^2} \cos^4 \alpha = \delta i \frac{\pi d^2}{4 z_i^2} \cos^4 \alpha \approx \delta i \frac{\pi}{4} \left(\frac{d}{f} \right)^2 \cos^4 \alpha, \quad (2.101)$$

which is called the *fundamental radiometric relation* between the scene radiance L and the light (irradiance) E reaching the pixel sensor,

$$E = L \frac{\pi}{4} \left(\frac{d}{f} \right)^2 \cos^4 \alpha, \quad (2.102)$$

(Horn 1986; Nalwa 1993; Ray 2002; Hecht 2015). Notice in this equation how the amount of light depends on the pixel surface area (which is why the smaller sensors in point-and-shoot cameras are so much noisier than digital single lens reflex (SLR) cameras), the inverse square of the f-stop $N = f/d$ (2.98), and the fourth power of the $\cos^4 \alpha$ off-axis fall-off, which is the natural vignetting term.

The other major kind of vignetting, called *mechanical vignetting*, is caused by the internal occlusion of rays near the periphery of lens elements in a compound lens, and cannot easily be described mathematically without performing a full ray-tracing of the actual lens design.¹² However, unlike natural vignetting, mechanical vignetting can be decreased by reducing the camera aperture (increasing the f-number). It can also be calibrated (along with natural vignetting) using special devices such as integrating spheres, uniformly illuminated targets, or camera rotation, as discussed in Section 10.1.3.

2.3 The digital camera

After starting from one or more light sources, reflecting off one or more surfaces in the world, and passing through the camera's optics (lenses), light finally reaches the imaging sensor. How are the photons arriving at this sensor converted into the digital (R, G, B) values that we observe when we look at a digital image? In this section, we develop a simple model that accounts for the most important effects, such as exposure (gain and shutter speed), non-linear mappings, sampling and aliasing, and noise. Figure 2.23, which is based on camera models developed by Healey and Kondepudy (1994), Tsin, Ramesh, and Kanade (2001), and Liu, Szeliski *et al.* (2008), shows a simple version of the processing stages that occur in modern digital cameras. Chakrabarti, Scharstein, and Zickler (2009) developed a sophisticated 24-parameter model that is an even better match to the processing performed in digital cameras, while Kim, Lin *et al.* (2012), Hasinoff, Sharlet *et al.* (2016), and Karaimer and Brown

¹²There are some empirical models that work well in practice (Kang and Weiss 2000; Zheng, Lin, and Kang 2006).

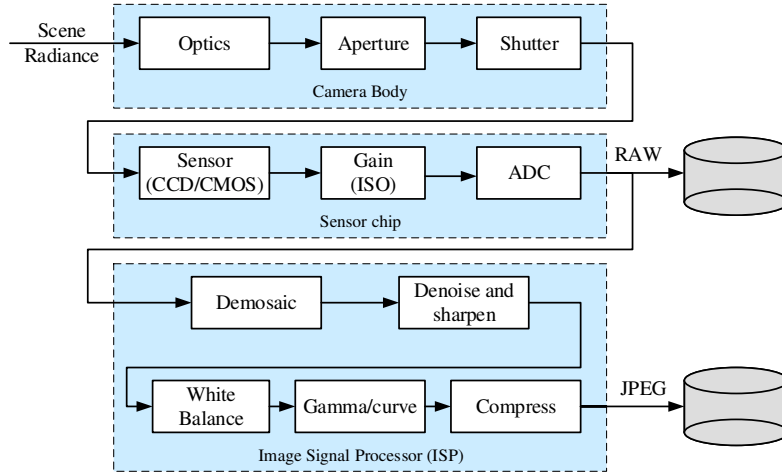


Figure 2.23 Image sensing pipeline, showing the various sources of noise as well as typical digital post-processing steps.

(2016) provide more recent models of modern in-camera processing pipelines. Most recently, Brooks, Mildenhall *et al.* (2019) have developed detailed models of in-camera image processing pipelines to invert (*unprocess*) noisy JPEG images into their RAW originals, so that they can be better denoised, while Tseng, Yu *et al.* (2019) develop a tunable model of camera processing pipelines that can be used for image quality optimization.

Light falling on an imaging sensor is usually picked up by an *active sensing area*, integrated for the duration of the exposure (usually expressed as the shutter speed in a fraction of a second, e.g., $\frac{1}{125}$, $\frac{1}{60}$, $\frac{1}{30}$), and then passed to a set of *sense amplifiers*. The two main kinds of sensor used in digital still and video cameras today are charge-coupled device (CCD) and complementary metal oxide on silicon (CMOS).

In a CCD, photons are accumulated in each active *well* during the exposure time. Then, in a *transfer* phase, the charges are transferred from well to well in a kind of “bucket brigade” until they are deposited at the sense amplifiers, which amplify the signal and pass it to an analog-to-digital converter (ADC).¹³ Older CCD sensors were prone to *blooming*, when charges from one over-exposed pixel spilled into adjacent ones, but most newer CCDs have anti-blooming technology (“troughs” into which the excess charge can spill).

In CMOS, the photons hitting the sensor directly affect the conductivity (or gain) of a

¹³In digital still cameras, a complete frame is captured and then read out sequentially at once. However, if video is being captured, a *rolling shutter*, which exposes and transfers each line separately, is often used. In older video cameras, the even fields (lines) were scanned first, followed by the odd fields, in a process that is called *interlacing*.

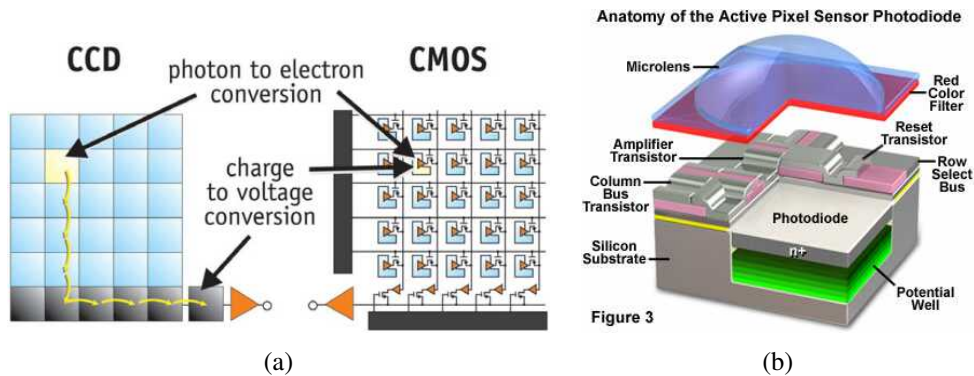


Figure 2.24 Digital imaging sensors: (a) CCDs move photogenerated charge from pixel to pixel and convert it to voltage at the output node; CMOS imagers convert charge to voltage inside each pixel (Litwiller 2005) © 2005 Photonics Spectra; (b) cutaway diagram of a CMOS pixel sensor, from <https://micro.magnet.fsu.edu/primer/digitalimaging/cmosimagesensors.html>.

photodetector, which can be selectively gated to control exposure duration, and locally amplified before being read out using a multiplexing scheme. Traditionally, CCD sensors outperformed CMOS in quality-sensitive applications, such as digital SLRs, while CMOS was better for low-power applications, but today CMOS is used in most digital cameras.

The main factors affecting the performance of a digital image sensor are the shutter speed, sampling pitch, fill factor, chip size, analog gain, sensor noise, and the resolution (and quality) of the analog-to-digital converter. Many of the actual values for these parameters can be read from the EXIF tags embedded with digital images, while others can be obtained from the camera manufacturers' specification sheets or from camera review or calibration websites.¹⁴

Shutter speed. The shutter speed (exposure time) directly controls the amount of light reaching the sensor and hence determines if images are under- or over-exposed. (For bright scenes, where a large aperture or slow shutter speed is desired to get a shallow depth of field or motion blur, *neutral density filters* are sometimes used by photographers.) For dynamic scenes, the shutter speed also determines the amount of *motion blur* in the resulting picture. Usually, a higher shutter speed (less motion blur) makes subsequent analysis easier (see Section 10.3 for techniques to remove such blur). However, when video is being captured for display, some motion blur may be desirable to avoid stroboscopic effects.

¹⁴<http://www.clarkvision.com/imagedetail/digital.sensor.performance.summary>

Sampling pitch. The sampling pitch is the physical spacing between adjacent sensor cells on the imaging chip (Figure 2.24). A sensor with a smaller sampling pitch has a higher *sampling density* and hence provides a higher *resolution* (in terms of pixels) for a given active chip area. However, a smaller pitch also means that each sensor has a smaller area and cannot accumulate as many photons; this makes it not as *light sensitive* and more prone to noise.

Fill factor. The fill factor is the active sensing area size as a fraction of the theoretically available sensing area (the product of the horizontal and vertical sampling pitches). Higher fill factors are usually preferable, as they result in more light capture and less *aliasing* (see Section 2.3.1). While the fill factor was originally limited by the need to place additional electronics between the active sensing areas, modern *backside illumination* (or *back-illuminated*) sensors, coupled with efficient microlens designs, have largely removed this limitation (Fontaine 2015).¹⁵ The fill factor of a camera can be determined empirically using a photometric camera calibration process (see Section 10.1.4).

Chip size. Video and point-and-shoot cameras have traditionally used small chip areas ($\frac{1}{4}$ -inch to $\frac{1}{2}$ -inch sensors¹⁶), while digital SLR cameras try to come closer to the traditional size of a 35mm film frame.¹⁷ When overall device size is not important, having a larger chip size is preferable, since each sensor cell can be more photo-sensitive. (For compact cameras, a smaller chip means that all of the optics can be shrunk down proportionately.) However, larger chips are more expensive to produce, not only because fewer chips can be packed into each wafer, but also because the probability of a chip defect goes up exponentially with the chip area.

Analog gain. Before analog-to-digital conversion, the sensed signal is usually boosted by a *sense amplifier*. In video cameras, the gain on these amplifiers was traditionally controlled by *automatic gain control* (AGC) logic, which would adjust these values to obtain a good overall exposure. In newer digital still cameras, the user now has some additional control over this gain through the *ISO setting*, which is typically expressed in ISO standard units such as 100, 200, or 400. Since the automated exposure control in most cameras also adjusts

¹⁵https://en.wikipedia.org/wiki/Back-illuminated_sensor

¹⁶These numbers refer to the “tube diameter” of the old vidicon tubes used in video cameras. The 1/2.5” sensor on the Canon SD800 camera actually measures 5.76mm $\times$ 4.29mm, i.e., a sixth of the size (on side) of a 35mm full-frame (36mm $\times$ 24mm) DSLR sensor.

¹⁷When a DSLR chip does not fill the 35mm full frame, it results in a *multiplier effect* on the lens focal length. For example, a chip that is only 0.6 the dimension of a 35mm frame will make a 50mm lens image the same angular extent as a $50/0.6 = 50 \times 1.6 = 80$ mm lens, as demonstrated in (2.60).

the aperture and shutter speed, setting the ISO manually removes one degree of freedom from the camera's control, just as manually specifying aperture and shutter speed does. In theory, a higher gain allows the camera to perform better under low light conditions (less motion blur due to long exposure times when the aperture is already maxed out). In practice, however, higher ISO settings usually amplify the *sensor noise*.

Sensor noise. Throughout the whole sensing process, noise is added from various sources, which may include *fixed pattern noise*, *dark current noise*, *shot noise*, *amplifier noise*, and *quantization noise* (Healey and Kondepudy 1994; Tsin, Ramesh, and Kanade 2001). The final amount of noise present in a sampled image depends on all of these quantities, as well as the incoming light (controlled by the scene radiance and aperture), the exposure time, and the sensor gain. Also, for low light conditions where the noise is due to low photon counts, a Poisson model of noise may be more appropriate than a Gaussian model (Alter, Matsushita, and Tang 2006; Matsushita and Lin 2007a; Wilburn, Xu, and Matsushita 2008; Takamatsu, Matsushita, and Ikeuchi 2008).

As discussed in more detail in Section 10.1.1, Liu, Szeliski *et al.* (2008) use this model, along with an empirical database of camera response functions (CRFs) obtained by Grossberg and Nayar (2004), to estimate the *noise level function* (NLF) for a given image, which predicts the overall noise variance at a given pixel as a function of its brightness (a separate NLF is estimated for each color channel). An alternative approach, when you have access to the camera before taking pictures, is to pre-calibrate the NLF by taking repeated shots of a scene containing a variety of colors and luminances, such as the Macbeth Color Chart shown in Figure 10.3b (McCamy, Marcus, and Davidson 1976). (When estimating the variance, be sure to throw away or downweight pixels with large gradients, as small shifts between exposures will affect the sensed values at such pixels.) Unfortunately, the pre-calibration process may have to be repeated for different exposure times and gain settings because of the complex interactions occurring within the sensing system.

In practice, most computer vision algorithms, such as image denoising, edge detection, and stereo matching, all benefit from at least a rudimentary estimate of the noise level. Barring the ability to pre-calibrate the camera or to take repeated shots of the same scene, the simplest approach is to look for regions of near-constant value and to estimate the noise variance in such regions (Liu, Szeliski *et al.* 2008).

ADC resolution. The final step in the analog processing chain occurring within an imaging sensor is the *analog to digital conversion* (ADC). While a variety of techniques can be used to implement this process, the two quantities of interest are the *resolution* of this process

(how many bits it yields) and its noise level (how many of these bits are useful in practice). For most cameras, the number of bits quoted (eight bits for compressed JPEG images and a nominal 16 bits for the RAW formats provided by some DSLRs) exceeds the actual number of usable bits. The best way to tell is to simply calibrate the noise of a given sensor, e.g., by taking repeated shots of the same scene and plotting the estimated noise as a function of brightness (Exercise 2.6).

Digital post-processing. Once the irradiance values arriving at the sensor have been converted to digital bits, most cameras perform a variety of *digital signal processing* (DSP) operations to enhance the image before compressing and storing the pixel values. These include color filter array (CFA) demosaicing, white point setting, and mapping of the luminance values through a *gamma function* to increase the perceived dynamic range of the signal. We cover these topics in Section 2.3.2 but, before we do, we return to the topic of aliasing, which was mentioned in connection with sensor array fill factors.

Newer imaging sensors. The capabilities of imaging sensor and related technologies such as depth sensors continue to evolve rapidly. Conferences that track these developments include the IS&T Symposium on Electronic Imaging Science and Technology sponsored by the Society for Imaging Science and Technology and the Image Sensors World blog.

2.3.1 Sampling and aliasing

What happens when a field of light impinging on the image sensor falls onto the active sense areas in the imaging chip? The photons arriving at each active cell are integrated and then digitized, as shown in Figure 2.24. However, if the fill factor on the chip is small and the signal is not otherwise *band-limited*, visually displeasing aliasing can occur.

To explore the phenomenon of aliasing, let us first look at a one-dimensional signal (Figure 2.25), in which we have two sine waves, one at a frequency of $f = 3/4$ and the other at $f = 5/4$. If we sample these two signals at a frequency of $f = 2$, we see that they produce the same samples (shown in black), and so we say that they are *aliased*.¹⁸ Why is this a bad effect? In essence, we can no longer reconstruct the original signal, since we do not know which of the two original frequencies was present.

In fact, Shannon's Sampling Theorem shows that the minimum sampling (Oppenheim and Schafer 1996; Oppenheim, Schafer, and Buck 1999) rate required to reconstruct a signal

¹⁸An alias is an alternate name for someone, so the sampled signal corresponds to two different *aliases*.

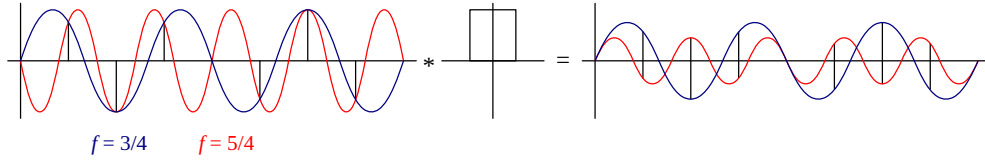


Figure 2.25 Aliasing of a one-dimensional signal: The blue sine wave at $f = 3/4$ and the red sine wave at $f = 5/4$ have the same digital samples, when sampled at $f = 2$. Even after convolution with a 100% fill factor box filter, the two signals, while no longer of the same magnitude, are still aliased in the sense that the sampled red signal looks like an inverted lower magnitude version of the blue signal. (The image on the right is scaled up for better visibility. The actual sine magnitudes are 30% and -18% of their original values.)

from its instantaneous samples must be at least twice the highest frequency,¹⁹

$$f_s \geq 2f_{\max}. \quad (2.103)$$

The maximum frequency in a signal is known as the *Nyquist frequency* and the inverse of the minimum sampling frequency $r_s = 1/f_s$ is known as the *Nyquist rate*.

However, you may ask, as an imaging chip actually *averages* the light field over a finite area, are the results on point sampling still applicable? Averaging over the sensor area does tend to attenuate some of the higher frequencies. However, even if the fill factor is 100%, as in the right image of Figure 2.25, frequencies above the Nyquist limit (half the sampling frequency) still produce an aliased signal, although with a smaller magnitude than the corresponding band-limited signals.

A more convincing argument as to why aliasing is bad can be seen by downsampling a signal using a poor quality filter such as a box (square) filter. Figure 2.26 shows a high-frequency *chirp* image (so called because the frequencies increase over time), along with the results of sampling it with a 25% fill-factor area sensor, a 100% fill-factor sensor, and a high-quality 9-tap filter. Additional examples of downsampling (*decimation*) filters can be found in Section 3.5.2 and Figure 3.29.

The best way to predict the amount of aliasing that an imaging system (or even an image processing algorithm) will produce is to estimate the *point spread function* (PSF), which represents the response of a particular pixel sensor to an ideal point light source. The PSF is a combination (convolution) of the blur induced by the optical system (lens) and the finite

¹⁹The actual theorem states that f_s must be at least twice the signal *bandwidth* but, as we are not dealing with modulated signals such as radio waves during image capture, the maximum frequency suffices.

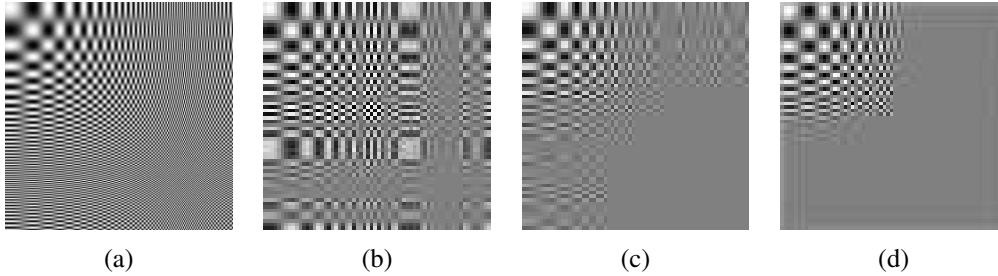


Figure 2.26 *Aliasing of a two-dimensional signal: (a) original full-resolution image; (b) downsampled $4 \times$ with a 25% fill factor box filter; (c) downsampled $4 \times$ with a 100% fill factor box filter; (d) downsampled $4 \times$ with a high-quality 9-tap filter. Notice how the higher frequencies are aliased into visible frequencies with the lower quality filters, while the 9-tap filter completely removes these higher frequencies.*

integration area of a chip sensor.²⁰

If we know the blur function of the lens and the fill factor (sensor area shape and spacing) for the imaging chip (plus, optionally, the response of the anti-aliasing filter), we can convolve these (as described in Section 3.2) to obtain the PSF. Figure 2.27a shows the one-dimensional cross-section of a PSF for a lens whose blur function is assumed to be a disc with a radius equal to the pixel spacing s plus a sensing chip whose horizontal fill factor is 80%. Taking the Fourier transform of this PSF (Section 3.4), we obtain the *modulation transfer function* (MTF), from which we can estimate the amount of aliasing as the area of the Fourier magnitude outside the $f \leq f_s$ Nyquist frequency.²¹ If we defocus the lens so that the blur function has a radius of $2s$ (Figure 2.27c), we see that the amount of aliasing decreases significantly, but so does the amount of image detail (frequencies closer to $f = f_s$).

Under laboratory conditions, the PSF can be estimated (to pixel precision) by looking at a point light source such as a pinhole in a black piece of cardboard lit from behind. However, this PSF (the actual image of the pinhole) is only accurate to a pixel resolution and, while it can model larger blur (such as blur caused by defocus), it cannot model the sub-pixel shape of the PSF and predict the amount of aliasing. An alternative technique, described in Section 10.1.4, is to look at a calibration pattern (e.g., one consisting of slanted step edges (Reichenbach, Park, and Narayanswamy 1991; Williams and Burns 2001; Joshi, Szeliski, and

²⁰Imaging chips usually interpose an optical *anti-aliasing filter* just before the imaging chip to reduce or control the amount of aliasing.

²¹The complex Fourier transform of the PSF is actually called the *optical transfer function* (OTF) (Williams 1999). Its magnitude is called the *modulation transfer function* (MTF) and its phase is called the *phase transfer function* (PTF).

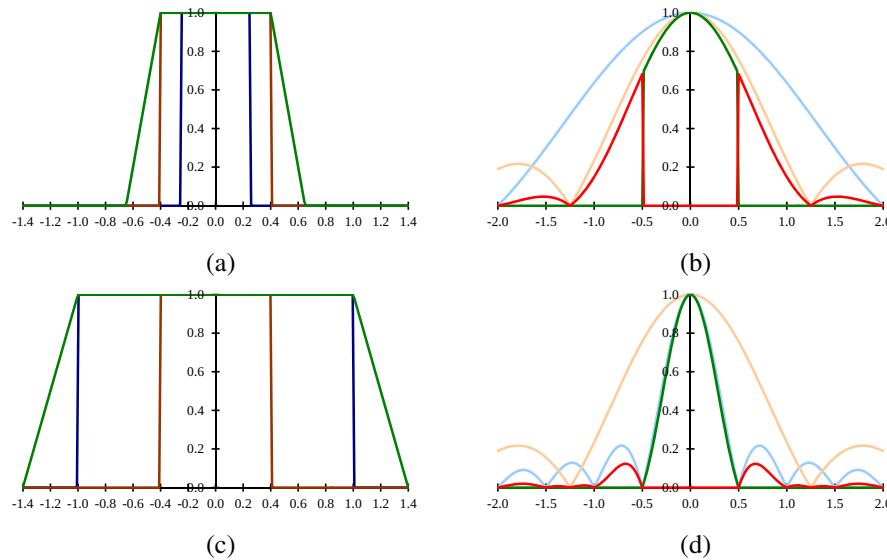


Figure 2.27 Sample point spread functions (PSF): The diameter of the blur disc (blue) in (a) is equal to half the pixel spacing, while the diameter in (c) is twice the pixel spacing. The horizontal fill factor of the sensing chip is 80% and is shown in brown. The convolution of these two kernels gives the point spread function, shown in green. The Fourier response of the PSF (the MTF) is plotted in (b) and (d). The area above the Nyquist frequency where aliasing occurs is shown in red.

(Kriegman 2008)) whose ideal appearance can be re-synthesized to sub-pixel precision.

In addition to occurring during image acquisition, aliasing can also be introduced in various image processing operations, such as resampling, upsampling, and downsampling. Sections 3.4 and 3.5.2 discuss these issues and show how careful selection of filters can reduce the amount of aliasing.

2.3.2 Color

In Section 2.2, we saw how lighting and surface reflections are functions of wavelength. When the incoming light hits the imaging sensor, light from different parts of the spectrum is somehow integrated into the discrete red, green, and blue (RGB) color values that we see in a digital image. How does this process work and how can we analyze and manipulate color values?

You probably recall from your childhood days the magical process of mixing paint colors

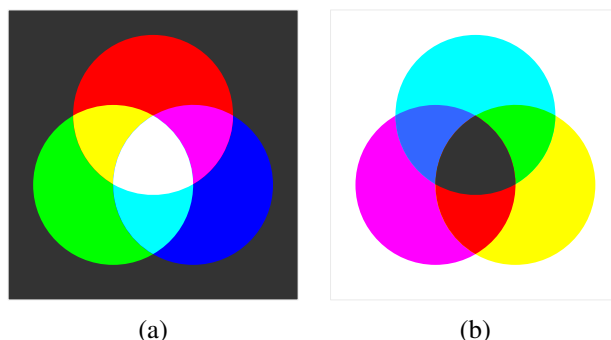


Figure 2.28 *Primary and secondary colors: (a) additive colors red, green, and blue can be mixed to produce cyan, magenta, yellow, and white; (b) subtractive colors cyan, magenta, and yellow can be mixed to produce red, green, blue, and black.*

to obtain new ones. You may recall that blue+yellow makes green, red+blue makes purple, and red+green makes brown. If you revisited this topic at a later age, you may have learned that the proper *subtractive* primaries are actually cyan (a light blue-green), magenta (pink), and yellow (Figure 2.28b), although black is also often used in four-color printing (CMYK).²² If you ever subsequently took any painting classes, you learned that colors can have even more fanciful names, such as alizarin crimson, cerulean blue, and chartreuse. The subtractive colors are called subtractive because pigments in the paint absorb certain wavelengths in the color spectrum.

Later on, you may have learned about the *additive* primary colors (red, green, and blue) and how they can be added (with a slide projector or on a computer monitor) to produce cyan, magenta, yellow, white, and all the other colors we typically see on our TV sets and monitors (Figure 2.28a).

Through what process is it possible for two different colors, such as red and green, to interact to produce a third color like yellow? Are the wavelengths somehow mixed up to produce a new wavelength?

You probably know that the correct answer has nothing to do with physically mixing wavelengths. Instead, the existence of three primaries is a result of the *tri-stimulus* (or *trichromatic*) nature of the human visual system, since we have three different kinds of cells called cones, each of which responds selectively to a different portion of the color spectrum (Glassner 1995; Wandell 1995; Wyszecki and Stiles 2000; Livingstone 2008; Frisby and Stone 2010; Reinhard, Heidrich *et al.* 2010; Fairchild 2013).²³ Note that for machine

²²It is possible to use additional inks such as orange, green, and violet to further extend the color gamut.

²³See also Mark Fairchild's web page, http://markfairchild.org/WhyIsColor/books_links.html.

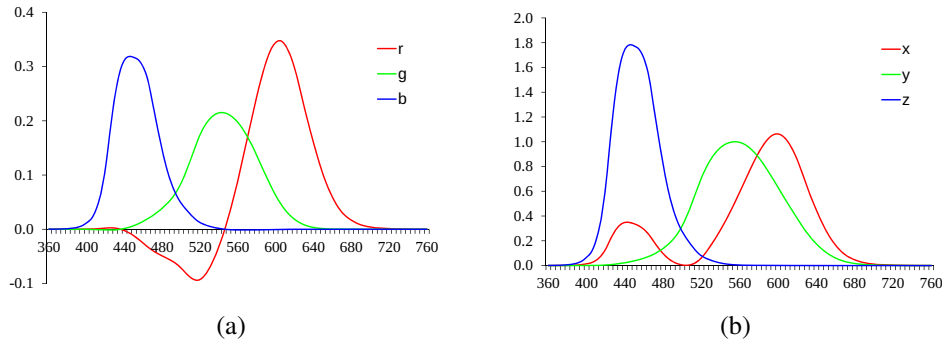


Figure 2.29 Standard CIE color matching functions: (a) $\bar{r}(\lambda)$, $\bar{g}(\lambda)$, $\bar{b}(\lambda)$ color spectra obtained from matching pure colors to the $R=700.0\text{nm}$, $G=546.1\text{nm}$, and $B=435.8\text{nm}$ primaries; (b) $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, $\bar{z}(\lambda)$ color matching functions, which are linear combinations of the $(\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda))$ spectra.

vision applications, such as remote sensing and terrain classification, it is preferable to use many more wavelengths. Similarly, surveillance applications can often benefit from sensing in the near-infrared (NIR) range.

CIE RGB and XYZ

To test and quantify the tri-chromatic theory of perception, we can attempt to reproduce all *monochromatic* (single wavelength) colors as a mixture of three suitably chosen primaries. (Pure wavelength light can be obtained using either a prism or specially manufactured color filters.) In the 1930s, the Commission Internationale d’Eclairage (CIE) standardized the RGB representation by performing such *color matching* experiments using the primary colors of red (700.0nm wavelength), green (546.1nm), and blue (435.8nm).

Figure 2.29 shows the results of performing these experiments with a *standard observer*, i.e., averaging perceptual results over a large number of subjects.²⁴ You will notice that for certain pure spectra in the blue–green range, a *negative* amount of red light has to be added, i.e., a certain amount of red has to be added to the color being matched to get a color match. These results also provided a simple explanation for the existence of *metamers*, which are colors with different spectra that are perceptually indistinguishable. Note that two fabrics or paint colors that are metamers under one light may no longer be so under different lighting.

Because of the problem associated with mixing negative light, the CIE also developed a

²⁴As Michael Brown notes in his tutorial on color (Brown 2019), the standard observer is actually an average taken over only 17 British subjects in the 1920s.

new color space called XYZ, which contains all of the pure spectral colors within its positive octant. (It also maps the Y axis to the *luminance*, i.e., perceived relative brightness, and maps pure white to a diagonal (equal-valued) vector.) The transformation from RGB to XYZ is given by

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \frac{1}{0.17697} \begin{bmatrix} 0.49 & 0.31 & 0.20 \\ 0.17697 & 0.81240 & 0.01063 \\ 0.00 & 0.01 & 0.99 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \quad (2.104)$$

While the official definition of the CIE XYZ standard has the matrix normalized so that the Y value corresponding to pure red is 1, a more commonly used form is to omit the leading fraction, so that the second row adds up to one, i.e., the RGB triplet (1, 1, 1) maps to a Y value of 1. Linearly blending the $(\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda))$ curves in Figure 2.29a according to (2.104), we obtain the resulting $(\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda))$ curves shown in Figure 2.29b. Notice how all three spectra (color matching functions) now have only positive values and how the $\bar{y}(\lambda)$ curve matches that of the luminance perceived by humans.

If we divide the XYZ values by the sum of $X+Y+Z$, we obtain the *chromaticity coordinates*

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z}, \quad (2.105)$$

which sum to 1. The chromaticity coordinates discard the absolute intensity of a given color sample and just represent its pure color. If we sweep the monochromatic color λ parameter in Figure 2.29b from $\lambda = 380\text{nm}$ to $\lambda = 800\text{nm}$, we obtain the familiar *chromaticity diagram* shown in Figure 2.30a. This figure shows the (x, y) value for every color value perceivable by most humans. (Of course, the CMYK reproduction process in this book does not actually span the whole gamut of perceivable colors.) The outer curved rim represents where all of the pure monochromatic color values map in (x, y) space, while the lower straight line, which connects the two endpoints, is known as the *purple line*. The inset triangle spans the red, green, and blue single-wavelength primaries used in the original color matching experiments, while E denotes the white point.

A convenient representation for color values, when we want to tease apart luminance and chromaticity, is therefore Yxy (luminance plus the two most distinctive chrominance components).

L*a*b* color space

While the XYZ color space has many convenient properties, including the ability to separate luminance from chrominance, it does not actually predict how well humans perceive *differences* in color or luminance.

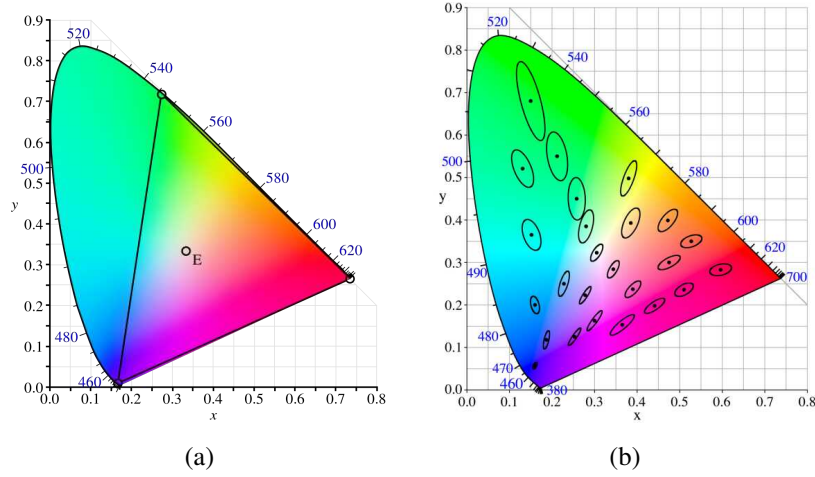


Figure 2.30 CIE chromaticity diagram, showing the pure single-wavelength spectral colors along the perimeter and the white point at E, plotted along their corresponding (x, y) values. (a) the red, green, and blue primaries do not span the complete gamut, so that negative amounts of red need to be added to span the blue–green range; (b) the MacAdam ellipses show color regions of equal discriminability, and form the basis of the Lab perceptual color space.

Because the response of the human visual system is roughly logarithmic (we can perceive *relative* luminance differences of about 1%), the CIE defined a non-linear re-mapping of the XYZ space called $L^*a^*b^*$ (also sometimes called CIELAB), where differences in luminance or chrominance are more perceptually uniform, as shown in Figure 2.30b.²⁵

The L^* component of *lightness* is defined as

$$L^* = 116f\left(\frac{Y}{Y_n}\right), \quad (2.106)$$

where Y_n is the luminance value for nominal white (Fairchild 2013) and

$$f(t) = \begin{cases} t^{1/3} & t > \delta^3 \\ t/(3\delta^2) + 2\delta/3 & \text{else,} \end{cases} \quad (2.107)$$

is a finite-slope approximation to the cube root with $\delta = 6/29$. The resulting 0...100 scale roughly measures equal amounts of lightness perceptibility.

²⁵Another perceptually motivated color space called $L^*u^*v^*$ was developed and standardized simultaneously (Fairchild 2013).

In a similar fashion, the a^* and b^* components are defined as

$$a^* = 500 \left[f \left(\frac{X}{X_n} \right) - f \left(\frac{Y}{Y_n} \right) \right] \quad \text{and} \quad b^* = 200 \left[f \left(\frac{Y}{Y_n} \right) - f \left(\frac{Z}{Z_n} \right) \right], \quad (2.108)$$

where again, (X_n, Y_n, Z_n) is the measured white point. Figure 2.33i–k show the $L^*a^*b^*$ representation for a sample color image.

Color cameras

While the preceding discussion tells us how we can uniquely describe the perceived tri-stimulus description of any color (spectral distribution), it does not tell us how RGB still and video cameras actually work. Do they just measure the amount of light at the nominal wavelengths of red (700.0nm), green (546.1nm), and blue (435.8nm)? Do color monitors just emit exactly these wavelengths and, if so, how can they emit negative red light to reproduce colors in the cyan range?

In fact, the design of RGB video cameras has historically been based around the availability of colored phosphors that go into television sets. When standard-definition color television was invented (NTSC), a mapping was defined between the RGB values that would drive the three color guns in the cathode ray tube (CRT) and the XYZ values that unambiguously define perceived color (this standard was called ITU-R BT.601). With the advent of HDTV and newer monitors, a new standard called ITU-R BT.709 was created, which specifies the XYZ values of each of the color primaries,

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \begin{bmatrix} R_{709} \\ G_{709} \\ B_{709} \end{bmatrix}. \quad (2.109)$$

In practice, each color camera integrates light according to the *spectral response function* of its red, green, and blue sensors,

$$\begin{aligned} R &= \int L(\lambda) S_R(\lambda) d\lambda, \\ G &= \int L(\lambda) S_G(\lambda) d\lambda, \\ B &= \int L(\lambda) S_B(\lambda) d\lambda, \end{aligned} \quad (2.110)$$

where $L(\lambda)$ is the incoming spectrum of light at a given pixel and $\{S_R(\lambda), S_G(\lambda), S_B(\lambda)\}$ are the red, green, and blue *spectral sensitivities* of the corresponding sensors.

Can we tell what spectral sensitivities the cameras actually have? Unless the camera manufacturer provides us with these data or we observe the response of the camera to a whole spectrum of monochromatic lights, these sensitivities are *not* specified by a standard such as BT.709. Instead, all that matters is that the tri-stimulus values for a given color produce the specified RGB values. The manufacturer is free to use sensors with sensitivities that do not match the standard XYZ definitions, so long as they can later be converted (through a linear transform) to the standard colors.

Similarly, while TV and computer monitors are supposed to produce RGB values as specified by Equation (2.109), there is no reason that they cannot use digital logic to transform the incoming RGB values into different signals to drive each of the color channels.²⁶ Properly calibrated monitors make this information available to software applications that perform *color management*, so that colors in real life, on the screen, and on the printer all match as closely as possible.

Color filter arrays

While early color TV cameras used three *vidicons* (tubes) to perform their sensing and later cameras used three separate RGB sensing chips, most of today's digital still and video cameras use a *color filter array* (CFA), where alternating sensors are covered by different colored filters (Figure 2.24).²⁷

The most commonly used pattern in color cameras today is the *Bayer pattern* (Bayer 1976), which places green filters over half of the sensors (in a checkerboard pattern), and red and blue filters over the remaining ones (Figure 2.31). The reason that there are twice as many green filters as red and blue is because the luminance signal is mostly determined by green values and the visual system is much more sensitive to high-frequency detail in luminance than in chrominance (a fact that is exploited in color image compression—see Section 2.3.3). The process of *interpolating* the missing color values so that we have valid RGB values for all the pixels is known as *demosaicing* and is covered in detail in Section 10.3.1.

Similarly, color LCD monitors typically use alternating stripes of red, green, and blue filters placed in front of each liquid crystal active area to simulate the experience of a full color display. As before, because the visual system has higher resolution (acuity) in luminance than chrominance, it is possible to digitally prefilter RGB (and monochrome) images to enhance

²⁶The latest OLED TV monitors are now introducing higher dynamic range (HDR) and wide color gamut (WCG), <https://www.cnet.com/how-to/what-is-wide-color-gamut-wcg>.

²⁷A chip design by Foveon stacked the red, green, and blue sensors beneath each other, but it never gained widespread adoption. Descriptions of alternative color filter arrays that have been proposed over the years can be found at https://en.wikipedia.org/wiki/Color_filter_array.

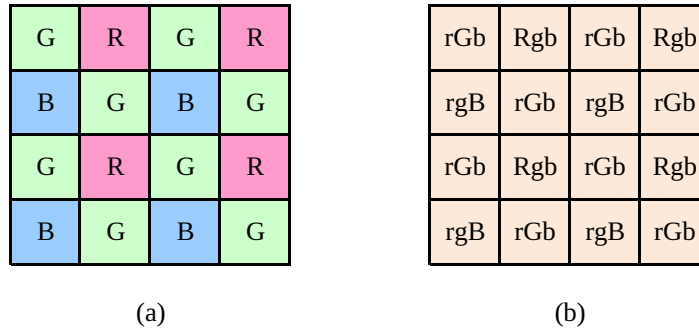


Figure 2.31 Bayer RGB pattern: (a) color filter array layout; (b) interpolated pixel values, with unknown (guessed) values shown as lower case.

the perception of crispness (Betrissey, Blinn *et al.* 2000; Platt 2000b).

Color balance

Before encoding the sensed RGB values, most cameras perform some kind of *color balancing* operation in an attempt to move the white point of a given image closer to pure white (equal RGB values). If the color system and the illumination are the same (the BT.709 system uses the daylight illuminant D_{65} as its reference white), the change may be minimal. However, if the illuminant is strongly colored, such as incandescent indoor lighting (which generally results in a yellow or orange hue), the compensation can be quite significant.

A simple way to perform color correction is to multiply each of the RGB values by a different factor (i.e., to apply a diagonal matrix transform to the RGB color space). More complicated transforms, which are sometimes the result of mapping to XYZ space and back, actually perform a *color twist*, i.e., they use a general 3×3 color transform matrix.²⁸ Exercise 2.8 has you explore some of these issues.

Gamma

In the early days of black and white television, the phosphors in the CRT used to display the TV signal responded non-linearly to their input voltage. The relationship between the voltage and the resulting brightness was characterized by a number called *gamma* (γ), since the formula was roughly

$$B = V^\gamma, \quad (2.111)$$

²⁸Those of you old enough to remember the early days of color television will naturally think of the *hue* adjustment knob on the television set, which could produce truly bizarre results.

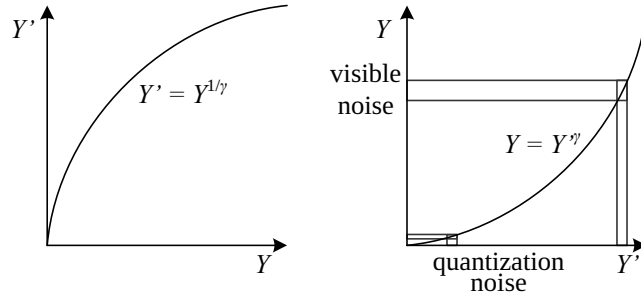


Figure 2.32 *Gamma compression: (a) The relationship between the input signal luminance Y and the transmitted signal Y' is given by $Y' = Y^{1/\gamma}$. (b) At the receiver, the signal Y' is exponentiated by the factor γ , $\hat{Y} = Y'^{\gamma}$. Noise introduced during transmission is squashed in the dark regions, which corresponds to the more noise-sensitive region of the visual system.*

with a γ of about 2.2. To compensate for this effect, the electronics in the TV camera would pre-map the sensed luminance Y through an inverse gamma,

$$Y' = Y^{\frac{1}{\gamma}}, \quad (2.112)$$

with a typical value of $\frac{1}{\gamma} = 0.45$.

The mapping of the signal through this non-linearity before transmission had a beneficial side effect: noise added during transmission (remember, these were analog days!) would be reduced (after applying the gamma at the receiver) in the darker regions of the signal where it was more visible (Figure 2.32).²⁹ (Remember that our visual system is roughly sensitive to relative differences in luminance.)

When color television was invented, it was decided to separately pass the red, green, and blue signals through the same gamma non-linearity before combining them for encoding. Today, even though we no longer have analog noise in our transmission systems, signals are still quantized during compression (see Section 2.3.3), so applying inverse gamma to sensed values remains useful.

Unfortunately, for both computer vision and computer graphics, the presence of gamma in images is often problematic. For example, the proper simulation of radiometric phenomena such as shading (see Section 2.2 and Equation (2.88)) occurs in a linear radiance space. Once all of the computations have been performed, the appropriate gamma should be applied before display. Unfortunately, many computer graphics systems (such as shading models) operate

²⁹ A related technique called *companding* was the basis of the Dolby noise reduction systems used with audio tapes.

directly on RGB values and display these values directly. (Fortunately, newer color imaging standards such as the 16-bit sRGB use a linear space, which makes this less of a problem (Glassner 1995).)

In computer vision, the situation can be even more daunting. The accurate determination of surface normals, using a technique such as photometric stereo (Section 13.1.1) or even a simpler operation such as accurate image deblurring, require that the measurements be in a linear space of intensities. Therefore, it is imperative when performing detailed quantitative computations such as these to first undo the gamma and the per-image color re-balancing in the sensed color values. Chakrabarti, Scharstein, and Zickler (2009) develop a sophisticated 24-parameter model that is a good match to the processing performed by today's digital cameras; they also provide a database of color images you can use for your own testing.

For other vision applications, however, such as feature detection or the matching of signals in stereo and motion estimation, this linearization step is often not necessary. In fact, determining whether it is necessary to undo gamma can take some careful thinking, e.g., in the case of compensating for exposure variations in image stitching (see Exercise 2.7).

If all of these processing steps sound confusing to model, they are. Exercise 2.9 has you try to tease apart some of these phenomena using empirical investigation, i.e., taking pictures of color charts and comparing the RAW and JPEG compressed color values.

Other color spaces

While RGB and XYZ are the primary color spaces used to describe the spectral content (and hence tri-stimulus response) of color signals, a variety of other representations have been developed both in video and still image coding and in computer graphics.

The earliest color representation developed for video transmission was the YIQ standard developed for NTSC video in North America and the closely related YUV standard developed for PAL in Europe. In both of these cases, it was desired to have a *luma* channel Y (so called since it only roughly mimics true luminance) that would be comparable to the regular black-and-white TV signal, along with two lower frequency *chroma* channels.

In both systems, the Y signal (or more appropriately, the Y' luma signal since it is gamma compressed) is obtained from

$$Y'_{601} = 0.299R' + 0.587G' + 0.114B', \quad (2.113)$$

where R'G'B' is the triplet of gamma-compressed color components. When using the newer color definitions for HDTV in BT.709, the formula is

$$Y'_{709} = 0.2125R' + 0.7154G' + 0.0721B'. \quad (2.114)$$

The UV components are derived from scaled versions of $(B' - Y')$ and $(R' - Y')$, namely,

$$U = 0.492111(B' - Y') \quad \text{and} \quad V = 0.877283(R' - Y'), \quad (2.115)$$

whereas the IQ components are the UV components rotated through an angle of 33° . In composite (NTSC and PAL) video, the chroma signals were then low-pass filtered horizontally before being modulated and superimposed on top of the Y' luma signal. Backward compatibility was achieved by having older black-and-white TV sets effectively ignore the high-frequency chroma signal (because of slow electronics) or, at worst, superimposing it as a high-frequency pattern on top of the main signal.

While these conversions were important in the early days of computer vision, when frame grabbers would directly digitize the composite TV signal, today all digital video and still image compression standards are based on the newer YCbCr conversion. YCbCr is closely related to YUV (the C_b and C_r signals carry the blue and red color difference signals and have more useful mnemonics than UV) but uses different scale factors to fit within the eight-bit range available with digital signals.

For video, the Y' signal is re-scaled to fit within the $[16 \dots 235]$ range of values, while the Cb and Cr signals are scaled to fit within $[16 \dots 240]$ (Gomes and Velho 1997; Fairchild 2013). For still images, the JPEG standard uses the full eight-bit range with no reserved values,

$$\begin{bmatrix} Y' \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}, \quad (2.116)$$

where the $R'G'B'$ values are the eight-bit gamma-compressed color components (i.e., the actual RGB values we obtain when we open up or display a JPEG image). For most applications, this formula is not that important, since your image reading software will directly provide you with the eight-bit gamma-compressed $R'G'B'$ values. However, if you are trying to do careful image deblocking (Exercise 4.3), this information may be useful.

Another color space you may come across is *hue, saturation, value* (HSV), which is a projection of the RGB color cube onto a non-linear chroma angle, a radial saturation percentage, and a luminance-inspired value. In more detail, value is defined as either the mean or maximum color value, saturation is defined as scaled distance from the diagonal, and hue is defined as the direction around a color wheel (the exact formulas are described by Hall (1989), Hughes, van Dam *et al.* (2013), and Brown (2019)). Such a decomposition is quite natural in graphics applications such as color picking (it approximates the Munsell chart for color description). Figure 2.33l–n shows an HSV representation of a sample color image,

where saturation is encoded using a gray scale (saturated = darker) and hue is depicted as a color.

If you want your computer vision algorithm to only affect the value (luminance) of an image and not its saturation or hue, a simpler solution is to use either the Yxy (luminance + chromaticity) coordinates defined in (2.105) or the even simpler *color ratios*,

$$r = \frac{R}{R+G+B}, \quad g = \frac{G}{R+G+B}, \quad b = \frac{B}{R+G+B} \quad (2.117)$$

(Figure 2.33e–h). After manipulating the luma (2.113), e.g., through the process of histogram equalization (Section 3.1.4), you can multiply each color ratio by the ratio of the new to old luma to obtain an adjusted RGB triplet.

While all of these color systems may sound confusing, in the end, it often may not matter that much which one you use. Poynton, in his *Color FAQ*, <https://www.poynton.com/ColorFAQ.html>, notes that the perceptually motivated $L^*a^*b^*$ system is qualitatively similar to the gamma-compressed $R'G'B'$ system we mostly deal with, since both have a fractional power scaling (which approximates a logarithmic response) between the actual intensity values and the numbers being manipulated. As in all cases, think carefully about what you are trying to accomplish before deciding on a technique to use.

2.3.3 Compression

The last stage in a camera's processing pipeline is usually some form of image compression (unless you are using a lossless compression scheme such as camera RAW or PNG).

All color video and image compression algorithms start by converting the signal into YCbCr (or some closely related variant), so that they can compress the luminance signal with higher fidelity than the chrominance signal. (Recall that the human visual system has poorer frequency response to color than to luminance changes.) In video, it is common to subsample Cb and Cr by a factor of two horizontally; with still images (JPEG), the subsampling (averaging) occurs both horizontally and vertically.

Once the luminance and chrominance images have been appropriately subsampled and separated into individual images, they are then passed to a *block transform* stage. The most common technique used here is the *discrete cosine transform* (DCT), which is a real-valued variant of the discrete Fourier transform (DFT) (see Section 3.4.1). The DCT is a reasonable approximation to the Karhunen–Loève or eigenvalue decomposition of natural image patches, i.e., the decomposition that simultaneously packs the most energy into the first coefficients and diagonalizes the joint covariance matrix among the pixels (makes transform coefficients statistically independent). Both MPEG and JPEG use 8×8 DCT transforms (Wallace 1991;

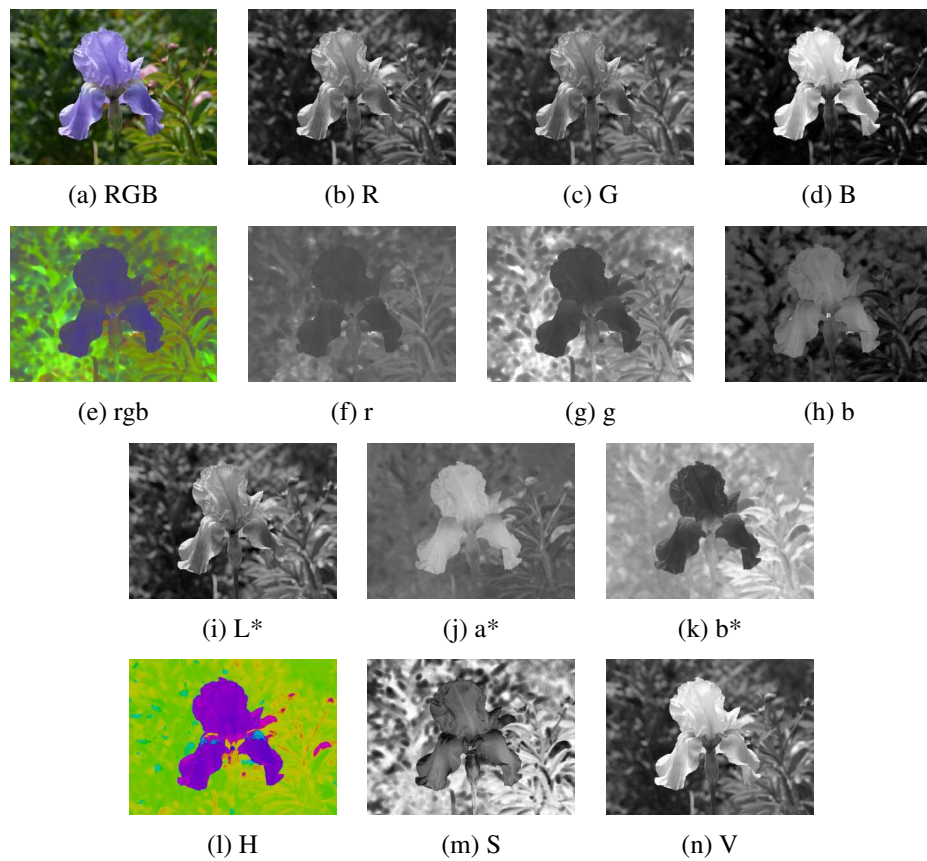


Figure 2.33 Color space transformations: (a–d) RGB; (e–h) rgb. (i–k) $L^*a^*b^*$; (l–n) HSV. Note that the rgb, $L^*a^*b^*$, and HSV values are all re-scaled to fit the dynamic range of the printed page.



Figure 2.34 Image compressed with JPEG at three quality settings. Note how the amount of block artifact and high-frequency aliasing (“mosquito noise”) increases from left to right.

Le Gall 1991), although newer variants, including the new AV1 open standard,³⁰ use smaller 4×4 or even 2×2 blocks. Alternative transformations, such as wavelets (Taubman and Marcellin 2002) and lapped transforms (Malvar 1990, 1998, 2000) are used in compression standards such as JPEG 2000 and JPEG XR.

After transform coding, the coefficient values are quantized into a set of small integer values that can be coded using a variable bit length scheme such as a Huffman code or an arithmetic code (Wallace 1991; Marpe, Schwarz, and Wiegand 2003). (The DC (lowest frequency) coefficients are also adaptively predicted from the previous block’s DC values. The term “DC” comes from “direct current”, i.e., the non-sinusoidal or non-alternating part of a signal.) The step size in the quantization is the main variable controlled by the *quality* setting on the JPEG file (Figure 2.34).

With video, it is also usual to perform block-based *motion compensation*, i.e., to encode the difference between each block and a *predicted* set of pixel values obtained from a shifted block in the previous frame. (The exception is the *motion-JPEG* scheme used in older DV camcorders, which is nothing more than a series of individually JPEG compressed image frames.) While basic MPEG uses 16×16 motion compensation blocks with integer motion values (Le Gall 1991), newer standards use adaptively sized blocks, sub-pixel motions, and the ability to reference blocks from older frames (Sullivan, Ohm *et al.* 2012). In order to recover more gracefully from failures and to allow for random access to the video stream, predicted P frames are interleaved among independently coded I frames. (Bi-directional B frames are also sometimes used.)

The quality of a compression algorithm is usually reported using its *peak signal-to-noise ratio* (PSNR), which is derived from the average *mean square error*,

$$MSE = \frac{1}{n} \sum_{\mathbf{x}} \left[I(\mathbf{x}) - \hat{I}(\mathbf{x}) \right]^2, \quad (2.118)$$

³⁰<https://aomedia.org>

where $I(\mathbf{x})$ is the original uncompressed image and $\hat{I}(\mathbf{x})$ is its compressed counterpart, or equivalently, the *root mean square error* (RMS error), which is defined as

$$RMS = \sqrt{MSE}. \quad (2.119)$$

The PSNR is defined as

$$PSNR = 10 \log_{10} \frac{I_{\max}^2}{MSE} = 20 \log_{10} \frac{I_{\max}}{RMS}, \quad (2.120)$$

where $I_{\max}$ is the maximum signal extent, e.g., 255 for eight-bit images.

While this is just a high-level sketch of how image compression works, it is useful to understand so that the artifacts introduced by such techniques can be compensated for in various computer vision applications. Note also that researchers are currently developing novel image and video compression algorithms based on deep neural networks, e.g., (Rippel and Bourdev 2017; Mentzer, Agustsson *et al.* 2019; Rippel, Nair *et al.* 2019) and <https://www.compression.cc>. It will be interesting to see what kinds of different artifacts these techniques produce.

2.4 Additional reading

As we mentioned at the beginning of this chapter, this book provides but a brief summary of a very rich and deep set of topics, traditionally covered in a number of separate fields.

A more thorough introduction to the geometry of points, lines, planes, and projections can be found in textbooks on multi-view geometry (Faugeras and Luong 2001; Hartley and Zisserman 2004) and computer graphics (Watt 1995; OpenGL-ARB 1997; Hughes, van Dam *et al.* 2013; Marschner and Shirley 2015). Topics covered in more depth include higher-order primitives such as quadrics, conics, and cubics, as well as three-view and multi-view geometry.

The image formation (synthesis) process is traditionally taught as part of a computer graphics curriculum (Glassner 1995; Watt 1995; Hughes, van Dam *et al.* 2013; Marschner and Shirley 2015) but it is also studied in physics-based computer vision (Wolff, Shafer, and Healey 1992a). The behavior of camera lens systems is studied in optics (Möller 1988; Ray 2002; Hecht 2015).

Some good books on color theory have been written by Healey and Shafer (1992), Wandell (1995), Wyszecki and Stiles (2000), and Fairchild (2013), with Livingstone (2008) providing a more fun and informal introduction to the topic of color perception. Mark Fairchild's page of color books and links³¹ lists many other sources.

³¹http://markfairchild.org/WhyIsColor/books_links.html.

Topics relating to sampling and aliasing are covered in textbooks on signal and image processing (Crane 1997; Jähne 1997; Oppenheim and Schaffer 1996; Oppenheim, Schaffer, and Buck 1999; Pratt 2007; Russ 2007; Burger and Burge 2008; Gonzalez and Woods 2017).

Two courses that cover many of the above topics (image formation, lenses, color and sampling theory) in wonderful detail are Marc Levoy’s Digital Photography course at Stanford (Levoy 2010) and Michael Brown’s tutorial on the image processing pipeline at ICCV 2019 (Brown 2019). The recent book by Ikeuchi, Matsushita *et al.* (2020) also covers 3D geometry, photometry, and sensor models, but with an emphasis on *active illumination* systems.

2.5 Exercises

A note to students: This chapter is relatively light on exercises since it contains mostly background material and not that many usable techniques. If you really want to understand multi-view geometry in a thorough way, I encourage you to read and do the exercises provided by Hartley and Zisserman (2004). Similarly, if you want some exercises related to the image formation process, Glassner’s (1995) book is full of challenging problems.

Ex 2.1: Least squares intersection point and line fitting—advanced. Equation (2.4) shows how the intersection of two 2D lines can be expressed as their cross product, assuming the lines are expressed as homogeneous coordinates.

1. If you are given more than two lines and want to find a point $\tilde{\mathbf{x}}$ that minimizes the sum of squared distances to each line,

$$D = \sum_i (\tilde{\mathbf{x}} \cdot \tilde{\mathbf{l}}_i)^2, \quad (2.121)$$

how can you compute this quantity? (Hint: Write the dot product as $\tilde{\mathbf{x}}^T \tilde{\mathbf{l}}_i$ and turn the squared quantity into a *quadratic form*, $\tilde{\mathbf{x}}^T \mathbf{A} \tilde{\mathbf{x}}$.)

2. To fit a line to a bunch of points, you can compute the *centroid* (mean) of the points as well as the *covariance matrix* of the points around this mean. Show that the line passing through the centroid along the major axis of the covariance ellipsoid (largest eigenvector) minimizes the sum of squared distances to the points.
3. These two approaches are fundamentally different, even though projective duality tells us that points and lines are interchangeable. Why are these two algorithms so apparently different? Are they actually minimizing different objectives?

Ex 2.2: 2D transform editor. Write a program that lets you interactively create a set of rectangles and then modify their “pose” (2D transform). You should implement the following steps:

1. Open an empty window (“canvas”).
2. Shift drag (rubber-band) to create a new rectangle.
3. Select the deformation mode (motion model): translation, rigid, similarity, affine, or perspective.
4. Drag any corner of the outline to change its transformation.

This exercise should be built on a set of pixel coordinate and transformation classes, either implemented by yourself or from a software library. Persistence of the created representation (save and load) should also be supported (for each rectangle, save its transformation).

Ex 2.3: 3D viewer. Write a simple viewer for 3D points, lines, and polygons. Import a set of point and line commands (primitives) as well as a viewing transform. Interactively modify the object or camera transform. This viewer can be an extension of the one you created in Exercise 2.2. Simply replace the viewing transformations with their 3D equivalents.

(Optional) Add a z-buffer to do hidden surface removal for polygons.

(Optional) Use a 3D drawing package and just write the viewer control.

Ex 2.4: Focus distance and depth of field. Figure out how the focus distance and depth of field indicators on a lens are determined.

1. Compute and plot the focus distance z_o as a function of the distance traveled from the focal length $\Delta z_i = f - z_i$ for a lens of focal length f (say, 100mm). Does this explain the hyperbolic progression of focus distances you see on a typical lens (Figure 2.20)?
2. Compute the depth of field (minimum and maximum focus distances) for a given focus setting z_o as a function of the circle of confusion diameter c (make it a fraction of the sensor width), the focal length f , and the f-stop number N (which relates to the aperture diameter d). Does this explain the usual depth of field markings on a lens that bracket the in-focus marker, as in Figure 2.20a?
3. Now consider a zoom lens with a varying focal length f . Assume that as you zoom, the lens stays in focus, i.e., the distance from the rear nodal point to the sensor plane z_i adjusts itself automatically for a fixed focus distance z_o . How do the depth of field indicators vary as a function of focal length? Can you reproduce a two-dimensional plot that mimics the curved depth of field lines seen on the lens in Figure 2.20b?

Ex 2.5: F-numbers and shutter speeds. List the common f-numbers and shutter speeds that your camera provides. On older model SLRs, they are visible on the lens and shutter speed dials. On newer cameras, you have to look at the electronic viewfinder (or LCD screen/indicator) as you manually adjust exposures.

1. Do these form geometric progressions; if so, what are the ratios? How do these relate to exposure values (EVs)?
2. If your camera has shutter speeds of $\frac{1}{60}$ and $\frac{1}{125}$, do you think that these two speeds are exactly a factor of two apart or a factor of $125/60 = 2.083$ apart?
3. How accurate do you think these numbers are? Can you devise some way to measure exactly how the aperture affects how much light reaches the sensor and what the exact exposure times actually are?

Ex 2.6: Noise level calibration. Estimate the amount of noise in your camera by taking repeated shots of a scene with the camera mounted on a tripod. (Purchasing a remote shutter release is a good investment if you own a DSLR.) Alternatively, take a scene with constant color regions (such as a color checker chart) and estimate the variance by fitting a smooth function to each color region and then taking differences from the predicted function.

1. Plot your estimated variance as a function of level for each of your color channels separately.
2. Change the ISO setting on your camera; if you cannot do that, reduce the overall light in your scene (turn off lights, draw the curtains, wait until dusk). Does the amount of noise vary a lot with ISO/gain?
3. Compare your camera to another one at a different price point or year of make. Is there evidence to suggest that “you get what you pay for”? Does the quality of digital cameras seem to be improving over time?

Ex 2.7: Gamma correction in image stitching. Here’s a relatively simple puzzle. Assume you are given two images that are part of a panorama that you want to stitch (see Section 8.2). The two images were taken with different exposures, so you want to adjust the RGB values so that they match along the seam line. Is it necessary to undo the gamma in the color values in order to achieve this?

Ex 2.8: White point balancing—tricky. A common (in-camera or post-processing) technique for performing white point adjustment is to take a picture of a white piece of paper and to adjust the RGB values of an image to make this a neutral color.

1. Describe how you would adjust the RGB values in an image given a sample “white color” of (R_w, G_w, B_w) to make this color neutral (without changing the exposure too much).
2. Does your transformation involve a simple (per-channel) scaling of the RGB values or do you need a full 3×3 color twist matrix (or something else)?
3. Convert your RGB values to XYZ. Does the appropriate correction now only depend on the XY (or xy) values? If so, when you convert back to RGB space, do you need a full 3×3 color twist matrix to achieve the same effect?
4. If you used pure diagonal scaling in the direct RGB mode but end up with a twist if you work in XYZ space, how do you explain this apparent dichotomy? Which approach is correct? (Or is it possible that neither approach is actually correct?)

If you want to find out what your camera *actually* does, continue on to the next exercise.

Ex 2.9: In-camera color processing—challenging. If your camera supports a RAW pixel mode, take a pair of RAW and JPEG images, and see if you can infer what the camera is doing when it converts the RAW pixel values to the final color-corrected and gamma-compressed eight-bit JPEG pixel values.

1. Deduce the pattern in your color filter array from the correspondence between co-located RAW and color-mapped pixel values. Use a color checker chart at this stage if it makes your life easier. You may find it helpful to split the RAW image into four separate images (subsampling even and odd columns and rows) and to treat each of these new images as a “virtual” sensor.
2. Evaluate the quality of the demosaicing algorithm by taking pictures of challenging scenes which contain strong color edges (such as those shown in in Section 10.3.1).
3. If you can take the same exact picture after changing the color balance values in your camera, compare how these settings affect this processing.
4. Compare your results against those presented in (Chakrabarti, Scharstein, and Zickler 2009), Kim, Lin *et al.* (2012), Hasinoff, Sharlet *et al.* (2016), Karaimer and Brown (2016), and Brooks, Mildenhall *et al.* (2019) or use the data available in their database of color images.

Chapter 3

Image processing

3.1	Point operators	109
3.1.1	Pixel transforms	111
3.1.2	Color transforms	112
3.1.3	Compositing and matting	113
3.1.4	Histogram equalization	115
3.1.5	<i>Application: Tonal adjustment</i>	119
3.2	Linear filtering	119
3.2.1	Separable filtering	124
3.2.2	Examples of linear filtering	125
3.2.3	Band-pass and steerable filters	127
3.3	More neighborhood operators	131
3.3.1	Non-linear filtering	132
3.3.2	Bilateral filtering	133
3.3.3	Binary image processing	138
3.4	Fourier transforms	142
3.4.1	Two-dimensional Fourier transforms	146
3.4.2	<i>Application: Sharpening, blur, and noise removal</i>	148
3.5	Pyramids and wavelets	149
3.5.1	Interpolation	150
3.5.2	Decimation	153
3.5.3	Multi-resolution representations	154
3.5.4	Wavelets	159
3.5.5	<i>Application: Image blending</i>	165
3.6	Geometric transformations	168
3.6.1	Parametric transformations	168
3.6.2	Mesh-based warping	175
3.6.3	<i>Application: Feature-based morphing</i>	177
3.7	Additional reading	178
3.8	Exercises	180

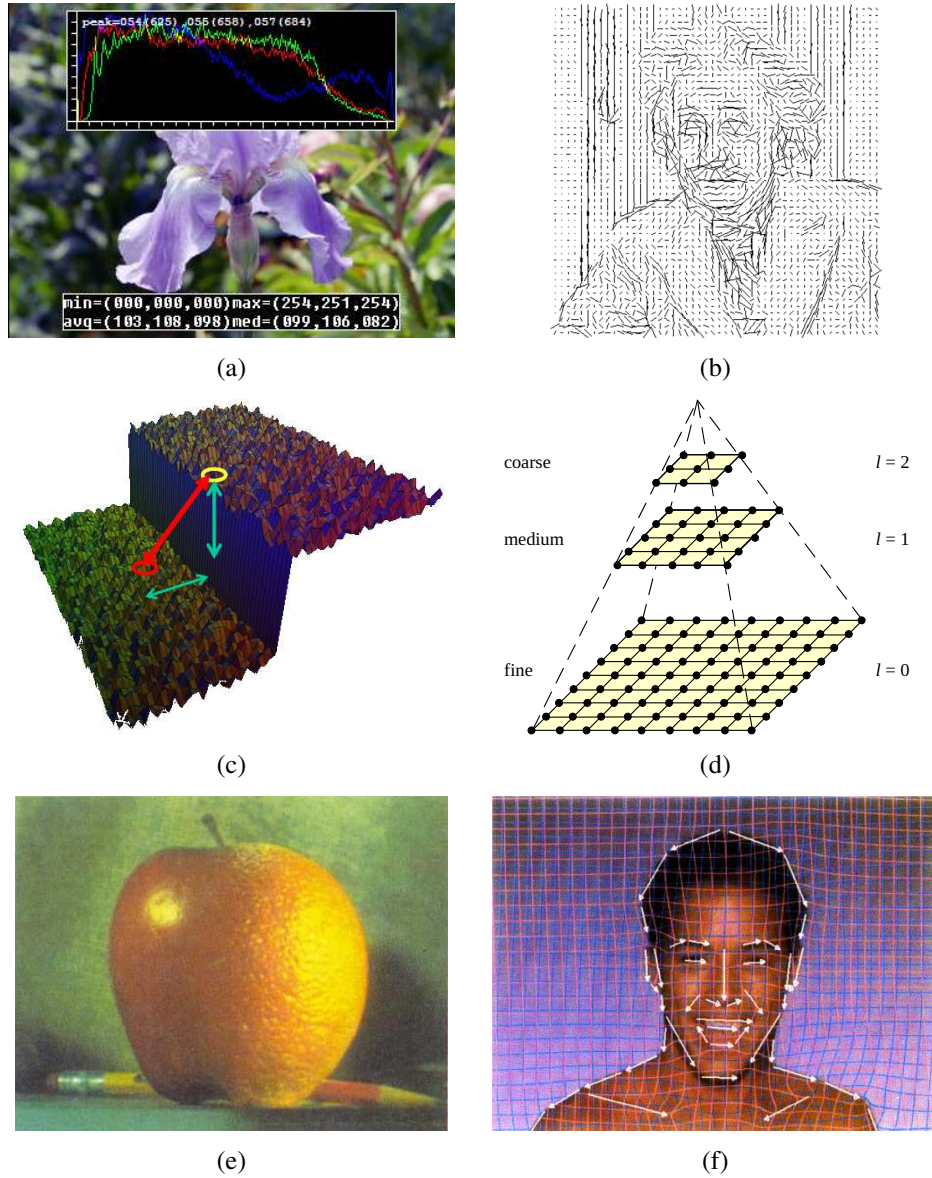


Figure 3.1 Some common image processing operations: (a) partial histogram equalization; (b) orientation map computed from the second-order steerable filter (Freeman 1992) © 1992 IEEE; (c) bilateral filter (Durand and Dorsey 2002) © 2002 ACM; (d) image pyramid; (e) Laplacian pyramid blending (Burt and Adelson 1983b) © 1983 ACM; (f) line-based image warping (Beier and Neely 1992) © 1992 ACM.

Now that we have seen how images are formed through the interaction of 3D scene elements, lighting, and camera optics and sensors, let us look at the first stage in most computer vision algorithms, namely the use of image processing to preprocess the image and convert it into a form suitable for further analysis. Examples of such operations include exposure correction and color balancing, reducing image noise, increasing sharpness, or straightening the image by rotating it. Additional examples include image warping and image blending, which are often used for visual effects (Figures 3.1 and Section 3.6.3). While some may consider image processing to be outside the purview of computer vision, most computer vision applications, such as computational photography and even recognition, require care in designing the image processing stages to achieve acceptable results.

In this chapter, we review standard image processing operators that map pixel values from one image to another. Image processing is often taught in electrical engineering departments as a follow-on course to an introductory course in signal processing (Oppenheim and Schaffer 1996; Oppenheim, Schaffer, and Buck 1999). There are several popular textbooks for image processing, including Gomes and Velho (1997), Jähne (1997), Pratt (2007), Burger and Burge (2009), and Gonzalez and Woods (2017).

We begin this chapter with the simplest kind of image transforms, namely those that manipulate each pixel independently of its neighbors (Section 3.1). Such transforms are often called *point operators* or *point processes*. Next, we examine *neighborhood* (area-based) operators, where each new pixel's value depends on a small number of neighboring input values (Sections 3.2 and 3.3). A convenient tool to analyze (and sometimes accelerate) such neighborhood operations is the *Fourier Transform*, which we cover in Section 3.4. Neighborhood operators can be cascaded to form *image pyramids* and *wavelets*, which are useful for analyzing images at a variety of resolutions (scales) and for accelerating certain operations (Section 3.5). Another important class of global operators are *geometric transformations*, such as rotations, shears, and perspective deformations (Section 3.6).

While this chapter covers *classical* image processing techniques that consist mostly of linear and non-linear filtering operations, the next two chapters introduce energy-based and Bayesian graphical models, i.e., *Markov random fields* (Chapter 4), and then deep convolutional networks (Chapter 5), both of which are now widely used in image processing applications.

3.1 Point operators

The simplest kinds of image processing transforms are *point operators*, where each output pixel's value depends on only the corresponding input pixel value (plus, potentially, some

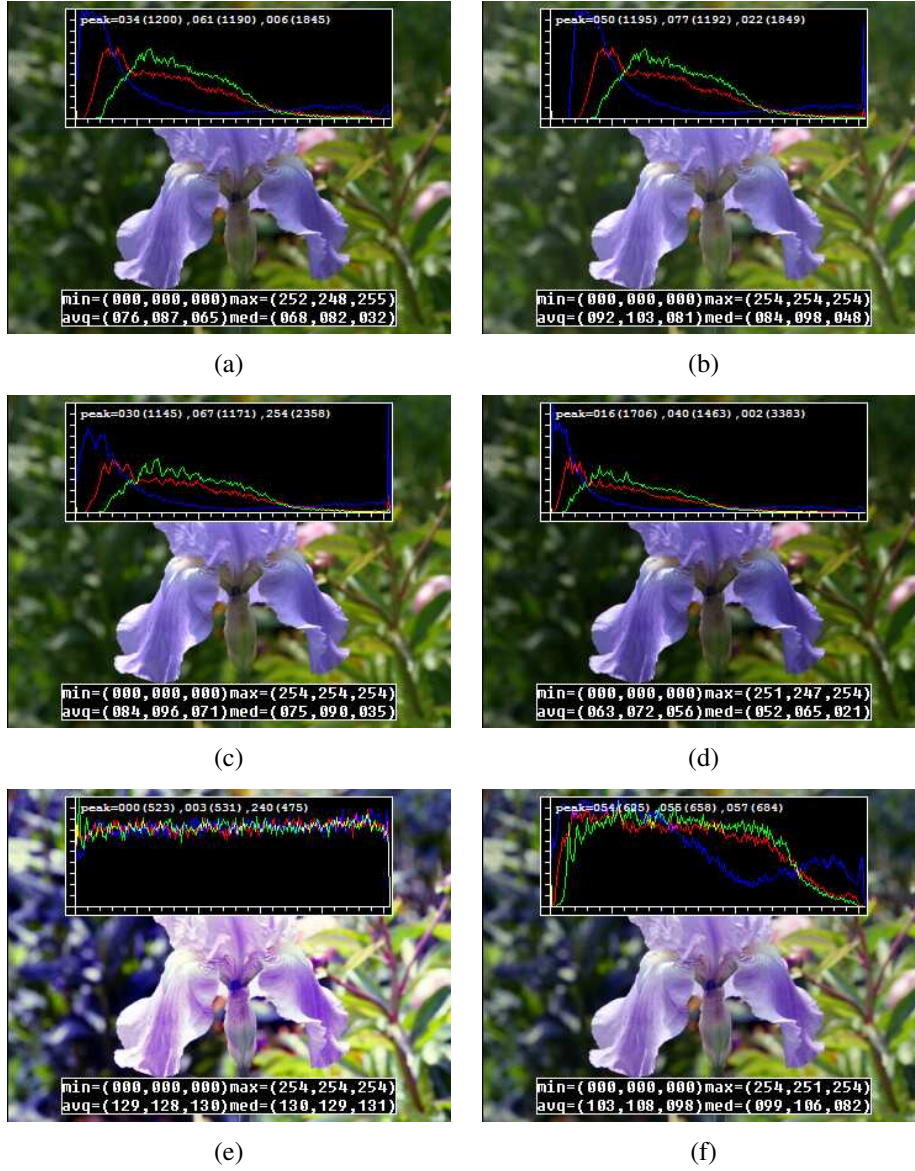


Figure 3.2 Some local image processing operations: (a) original image along with its three color (per-channel) histograms; (b) brightness increased (additive offset, $b = 16$); (c) contrast increased (multiplicative gain, $a = 1.1$); (d) gamma (partially) linearized ($\gamma = 1.2$); (e) full histogram equalization; (f) partial histogram equalization.

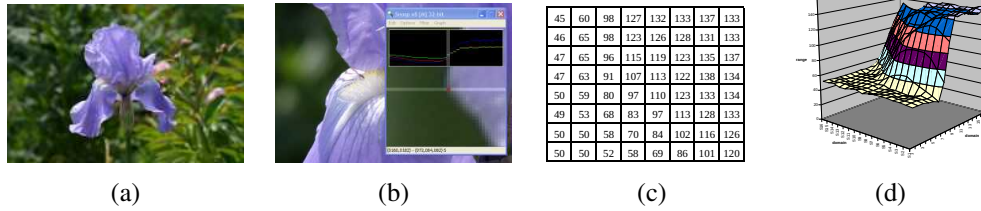


Figure 3.3 Visualizing image data: (a) original image; (b) cropped portion and scanline plot using an image inspection tool; (c) grid of numbers; (d) surface plot. For figures (c)–(d), the image was first converted to grayscale.

globally collected information or parameters). Examples of such operators include brightness and contrast adjustments (Figure 3.2) as well as color correction and transformations. In the image processing literature, such operations are also known as *point processes* (Crane 1997).¹

We begin this section with a quick review of simple point operators, such as brightness scaling and image addition. Next, we discuss how colors in images can be manipulated. We then present *image compositing* and *matting* operations, which play an important role in computational photography (Chapter 10) and computer graphics applications. Finally, we describe the more global process of *histogram equalization*. We close with an example application that manipulates *tonal values* (exposure and contrast) to improve image appearance.

3.1.1 Pixel transforms

A general image processing *operator* is a function that takes one or more input images and produces an output image. In the continuous domain, this can be denoted as

$$g(\mathbf{x}) = h(f(\mathbf{x})) \quad \text{or} \quad g(\mathbf{x}) = h(f_0(\mathbf{x}), \dots, f_n(\mathbf{x})), \quad (3.1)$$

where $\mathbf{x}$ is in the D -dimensional (usually $D = 2$ for images) *domain* of the input and output functions f and g , which operate over some *range*, which can either be scalar or vector-valued, e.g., for color images or 2D motion. For discrete (sampled) images, the domain consists of a finite number of *pixel locations*, $\mathbf{x} = (i, j)$, and we can write

$$g(i, j) = h(f(i, j)). \quad (3.2)$$

Figure 3.3 shows how an image can be represented either by its color (appearance), as a grid of numbers, or as a two-dimensional function (surface plot).

¹In convolutional neural networks (Section 5.4), such operations are sometimes called 1×1 convolutions.

Two commonly used point processes are multiplication and addition with a constant,

$$g(\mathbf{x}) = af(\mathbf{x}) + b. \quad (3.3)$$

The parameters $a > 0$ and b are often called the *gain* and *bias* parameters; sometimes these parameters are said to control *contrast* and *brightness*, respectively (Figures 3.2b–c).² The bias and gain parameters can also be spatially varying,

$$g(\mathbf{x}) = a(\mathbf{x})f(\mathbf{x}) + b(\mathbf{x}), \quad (3.4)$$

e.g., when simulating the *graded density filter* used by photographers to selectively darken the sky or when modeling vignetting in an optical system.

Multiplicative gain (both global and spatially varying) is a *linear* operation, as it obeys the *superposition principle*,

$$h(f_0 + f_1) = h(f_0) + h(f_1). \quad (3.5)$$

(We will have more to say about linear shift invariant operators in Section 3.2.) Operators such as image squaring (which is often used to get a local estimate of the *energy* in a band-pass filtered signal, see Section 3.5) are not linear.

Another commonly used *dyadic* (two-input) operator is the *linear blend* operator,

$$g(\mathbf{x}) = (1 - \alpha)f_0(\mathbf{x}) + \alpha f_1(\mathbf{x}). \quad (3.6)$$

By varying α from $0 \rightarrow 1$, this operator can be used to perform a temporal *cross-dissolve* between two images or videos, as seen in slide shows and film production, or as a component of image *morphing* algorithms (Section 3.6.3).

One highly used non-linear transform that is often applied to images before further processing is *gamma correction*, which is used to remove the non-linear mapping between input radiance and quantized pixel values (Section 2.3.2). To invert the gamma mapping applied by the sensor, we can use

$$g(\mathbf{x}) = [f(\mathbf{x})]^{1/\gamma}, \quad (3.7)$$

where a gamma value of $\gamma \approx 2.2$ is a reasonable fit for most digital cameras.

3.1.2 Color transforms

While color images can be treated as arbitrary vector-valued functions or collections of independent bands, it usually makes sense to think about them as highly correlated signals with

²An image's luminance characteristics can also be summarized by its *key* (average luminance) and *range* (Kopf, Uyttendaele *et al.* 2007).



Figure 3.4 Image matting and compositing (Chuang, Curless et al. 2001) © 2001 IEEE: (a) source image; (b) extracted foreground object F ; (c) alpha matte α shown in grayscale; (d) new composite C .

strong connections to the image formation process (Section 2.2), sensor design (Section 2.3), and human perception (Section 2.3.2). Consider, for example, brightening a picture by adding a constant value to all three channels, as shown in Figure 3.2b. Can you tell if this achieves the desired effect of making the image look brighter? Can you see any undesirable side-effects or artifacts?

In fact, adding the same value to each color channel not only increases the apparent *intensity* of each pixel, it can also affect the pixel's *hue* and *saturation*. How can we define and manipulate such quantities in order to achieve the desired perceptual effects?

As discussed in Section 2.3.2, chromaticity coordinates (2.105) or even simpler color ratios (2.117) can first be computed and then used after manipulating (e.g., brightening) the luminance Y to re-compute a valid RGB image with the same hue and saturation. Figures 2.33f–h show some color ratio images multiplied by the middle gray value for better visualization.

Similarly, color balancing (e.g., to compensate for incandescent lighting) can be performed either by multiplying each channel with a different scale factor or by the more complex process of mapping to XYZ color space, changing the nominal white point, and mapping back to RGB, which can be written down using a linear 3×3 *color twist* transform matrix. Exercises 2.8 and 3.1 have you explore some of these issues.

Another fun project, best attempted after you have mastered the rest of the material in this chapter, is to take a picture with a rainbow in it and enhance the strength of the rainbow (Exercise 3.29).

3.1.3 Compositing and matting

In many photo editing and visual effects applications, it is often desirable to cut a *foreground* object out of one scene and put it on top of a different *background* (Figure 3.4). The process of extracting the object from the original image is often called *matting* (Smith and Blinn

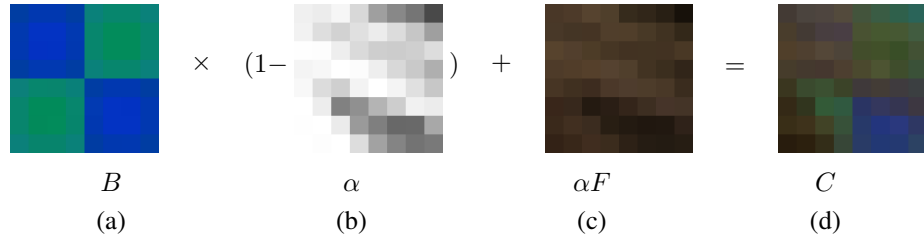


Figure 3.5 Compositing equation $C = (1 - \alpha)B + \alpha F$. The images are taken from a close-up of the region of the hair in the upper right part of the lion in Figure 3.4.

1996), while the process of inserting it into another image (without visible artifacts) is called *compositing* (Porter and Duff 1984; Blinn 1994a).

The intermediate representation used for the foreground object between these two stages is called an *alpha-matted color image* (Figure 3.4b–c). In addition to the three color RGB channels, an alpha-matted image contains a fourth *alpha* channel α (or A) that describes the relative amount of *opacity* or *fractional coverage* at each pixel (Figures 3.4c and 3.5b). The opacity is the opposite of the *transparency*. Pixels within the object are fully opaque ($\alpha = 1$), while pixels fully outside the object are transparent ($\alpha = 0$). Pixels on the boundary of the object vary smoothly between these two extremes, which hides the perceptual visible *jaggies* that occur if only binary opacities are used.

To composite a new (or foreground) image on top of an old (background) image, the *over operator*, first proposed by Porter and Duff (1984) and then studied extensively by Blinn (1994a; 1994b), is used:

$$C = (1 - \alpha)B + \alpha F. \quad (3.8)$$

This operator *attenuates* the influence of the background image B by a factor $(1 - \alpha)$ and then adds in the color (and opacity) values corresponding to the foreground layer F , as shown in Figure 3.5.

In many situations, it is convenient to represent the foreground colors in *pre-multiplied* form, i.e., to store (and manipulate) the αF values directly. As Blinn (1994b) shows, the pre-multiplied RGBA representation is preferred for several reasons, including the ability to blur or resample (e.g., rotate) alpha-matted images without any additional complications (just treating each RGBA band independently). However, when matting using local color consistency (Ruzon and Tomasi 2000; Chuang, Curless *et al.* 2001), the pure un-multiplied foreground colors F are used, since these remain constant (or vary slowly) in the vicinity of the object edge.

The over operation is not the only kind of compositing operation that can be used. Porter



Figure 3.6 An example of light reflecting off the transparent glass of a picture frame (Black and Anandan 1996) © 1996 Elsevier. You can clearly see the woman's portrait inside the picture frame superimposed with the reflection of a man's face off the glass.

and Duff (1984) describe a number of additional operations that can be useful in photo editing and visual effects applications. In this book, we concern ourselves with only one additional commonly occurring case (but see Exercise 3.3).

When light reflects off clean transparent glass, the light passing through the glass and the light reflecting off the glass are simply added together (Figure 3.6). This model is useful in the analysis of *transparent motion* (Black and Anandan 1996; Szeliski, Avidan, and Anandan 2000), which occurs when such scenes are observed from a moving camera (see Section 9.4.2).

The actual process of *matting*, i.e., recovering the foreground, background, and alpha matte values from one or more images, has a rich history, which we study in Section 10.4. Smith and Blinn (1996) have a nice survey of traditional *blue-screen matting* techniques, while Toyama, Krumm *et al.* (1999) review *difference matting*. Since then, there has been a lot of activity in computational photography relating to *natural image matting* (Ruzon and Tomasi 2000; Chuang, Curless *et al.* 2001; Wang and Cohen 2009; Xu, Price *et al.* 2017), which attempts to extract the mattes from a single natural image (Figure 3.4a) or from extended video sequences (Chuang, Agarwala *et al.* 2002). All of these techniques are described in more detail in Section 10.4.

3.1.4 Histogram equalization

While the brightness and gain controls described in Section 3.1.1 can improve the appearance of an image, how can we automatically determine their best values? One approach might be to look at the darkest and brightest pixel values in an image and map them to pure black and pure white. Another approach might be to find the *average* value in the image, push it

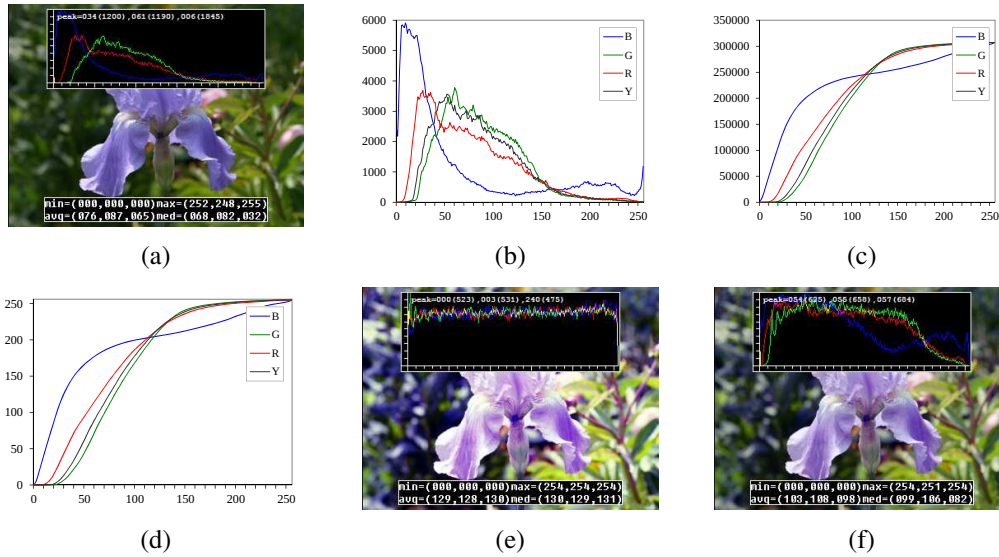


Figure 3.7 Histogram analysis and equalization: (a) original image; (b) color channel and intensity (luminance) histograms; (c) cumulative distribution functions; (d) equalization (transfer) functions; (e) full histogram equalization; (f) partial histogram equalization.

towards middle gray, and expand the *range* so that it more closely fills the displayable values (Kopf, Uyttendaele *et al.* 2007).

How can we visualize the set of lightness values in an image to test some of these heuristics? The answer is to plot the *histogram* of the individual color channels and luminance values, as shown in Figure 3.7b.³ From this distribution, we can compute relevant statistics such as the minimum, maximum, and average intensity values. Notice that the image in Figure 3.7a has both an excess of dark values and light values, but that the mid-range values are largely under-populated. Would it not be better if we could simultaneously brighten some dark values and darken some light values, while still using the full extent of the available dynamic range? Can you think of a mapping that might do this?

One popular answer to this question is to perform *histogram equalization*, i.e., to find an intensity mapping function $f(I)$ such that the resulting histogram is flat. The trick to finding such a mapping is the same one that people use to generate random samples from a *probability density function*, which is to first compute the *cumulative distribution function*

³The histogram is simply the *count* of the number of pixels at each gray level value. For an eight-bit image, an accumulation table with 256 entries is needed. For higher bit depths, a table with the appropriate number of entries (probably fewer than the full number of gray levels) should be used.

shown in Figure 3.7c.

Think of the original histogram $h(I)$ as the distribution of grades in a class after some exam. How can we map a particular grade to its corresponding *percentile*, so that students at the 75% percentile range scored better than $3/4$ of their classmates? The answer is to integrate the distribution $h(I)$ to obtain the cumulative distribution $c(I)$,

$$c(I) = \frac{1}{N} \sum_{i=0}^I h(i) = c(I-1) + \frac{1}{N} h(I), \quad (3.9)$$

where N is the number of pixels in the image or students in the class. For any given grade or intensity, we can look up its corresponding percentile $c(I)$ and determine the final value that the pixel should take. When working with eight-bit pixel values, the I and c axes are rescaled from $[0, 255]$.

Figure 3.7e shows the result of applying $f(I) = c(I)$ to the original image. As we can see, the resulting histogram is flat; so is the resulting image (it is “flat” in the sense of a lack of contrast and being muddy looking). One way to compensate for this is to only *partially* compensate for the histogram unevenness, e.g., by using a mapping function $f(I) = \alpha c(I) + (1 - \alpha)I$, which is a linear blend between the cumulative distribution function and the identity transform (a straight line). As you can see in Figure 3.7f, the resulting image maintains more of its original grayscale distribution while having a more appealing balance.

Another potential problem with histogram equalization (or, in general, image brightening) is that noise in dark regions can be amplified and become more visible. Exercise 3.7 suggests some possible ways to mitigate this, as well as alternative techniques to maintain contrast and “punch” in the original images (Larson, Rushmeier, and Piatko 1997; Stark 2000).

Locally adaptive histogram equalization

While global histogram equalization can be useful, for some images it might be preferable to apply different kinds of equalization in different regions. Consider for example the image in Figure 3.8a, which has a wide range of luminance values. Instead of computing a single curve, what if we were to subdivide the image into $M \times M$ pixel blocks and perform separate histogram equalization in each sub-block? As you can see in Figure 3.8b, the resulting image exhibits a lot of blocking artifacts, i.e., intensity discontinuities at block boundaries.

One way to eliminate blocking artifacts is to use a *moving window*, i.e., to recompute the histogram for every $M \times M$ block centered at each pixel. This process can be quite slow (M^2 operations per pixel), although with clever programming only the histogram entries corresponding to the pixels entering and leaving the block (in a raster scan across the image)

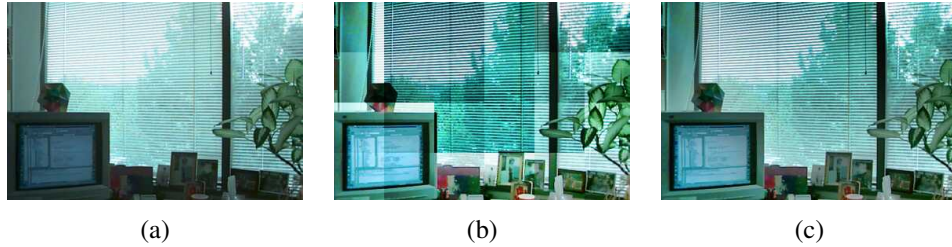


Figure 3.8 *Locally adaptive histogram equalization: (a) original image; (b) block histogram equalization; (c) full locally adaptive equalization.*

need to be updated (M operations per pixel). Note that this operation is an example of the *non-linear neighborhood operations* we study in more detail in Section 3.3.1.

A more efficient approach is to compute non-overlapped block-based equalization functions as before, but to then smoothly interpolate the transfer functions as we move between blocks. This technique is known as *adaptive histogram equalization* (AHE) and its contrast-limited (gain-limited) version is known as CLAHE (Pizer, Amburn *et al.* 1987).⁴ The weighting function for a given pixel (i, j) can be computed as a function of its horizontal and vertical position (s, t) within a block, as shown in Figure 3.9a. To blend the four lookup functions $\{f_{00}, \dots, f_{11}\}$, a *bilinear* blending function,

$$f_{s,t}(I) = (1-s)(1-t)f_{00}(I) + s(1-t)f_{10}(I) + (1-s)t f_{01}(I) + st f_{11}(I) \quad (3.10)$$

can be used. (See Section 3.5.2 for higher-order generalizations of such *spline* functions.) Note that instead of blending the four lookup tables for each output pixel (which would be quite slow), we can instead blend the results of mapping a given pixel through the four neighboring lookups.

A variant on this algorithm is to place the lookup tables at the *corners* of each $M \times M$ block (see Figure 3.9b and Exercise 3.8). In addition to blending four lookups to compute the final value, we can also *distribute* each input pixel into four adjacent lookup tables during the histogram accumulation phase (notice that the gray arrows in Figure 3.9b point both ways), i.e.,

$$h_{k,l}(I(i, j)) += w(i, j, k, l), \quad (3.11)$$

where $w(i, j, k, l)$ is the bilinear weighting function between pixel (i, j) and lookup table (k, l) . This is an example of *soft histogramming*, which is used in a variety of other applications, including the construction of SIFT feature descriptors (Section 7.1.3) and vocabulary trees (Section 7.1.4).

⁴The CLAHE algorithm is part of OpenCV.

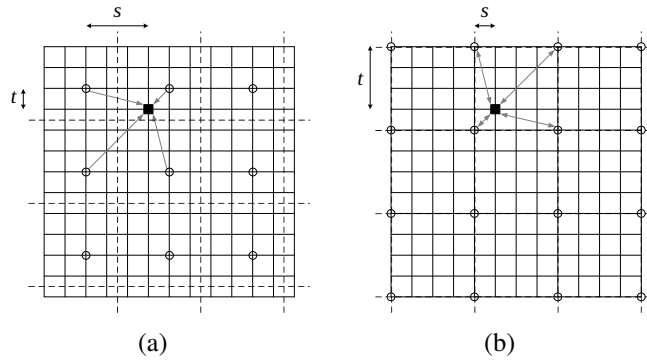


Figure 3.9 Local histogram interpolation using relative (s, t) coordinates: (a) block-based histograms, with block centers shown as circles; (b) corner-based “spline” histograms. Pixels are located on grid intersections. The black square pixel’s transfer function is interpolated from the four adjacent lookup tables (gray arrows) using the computed (s, t) values. Block boundaries are shown as dashed lines.

3.1.5 Application: Tonal adjustment

One of the most widely used applications of point-wise image processing operators is the manipulation of contrast or *tone* in photographs, to make them look either more attractive or more interpretable. You can get a good sense of the range of operations possible by opening up any photo manipulation tool and trying out a variety of contrast, brightness, and color manipulation options, as shown in Figures 3.2 and 3.7.

Exercises 3.1, 3.6, and 3.7 have you implement some of these operations, to become familiar with basic image processing operators. More sophisticated techniques for tonal adjustment (Bae, Paris, and Durand 2006; Reinhard, Heidrich *et al.* 2010) are described in the section on high dynamic range tone mapping (Section 10.2.1).

3.2 Linear filtering

Locally adaptive histogram equalization is an example of a *neighborhood operator* or *local operator*, which uses a collection of pixel values in the vicinity of a given pixel to determine its final output value (Figure 3.10). In addition to performing local tone adjustment, neighborhood operators can be used to *filter* images to add soft blur, sharpen details, accentuate edges, or remove noise (Figure 3.11b–d). In this section, we look at *linear* filtering operators, which involve fixed weighted combinations of pixels in small neighborhoods. In Section 3.3,

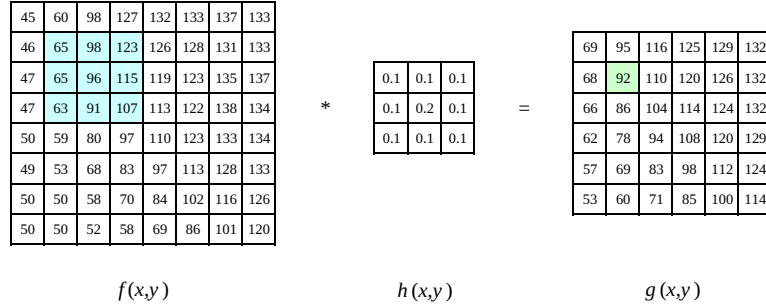


Figure 3.10 *Neighborhood filtering (convolution): The image on the left is convolved with the filter in the middle to yield the image on the right. The light blue pixels indicate the source neighborhood for the light green destination pixel.*

we look at non-linear operators such as morphological filters and distance transforms.

The most widely used type of neighborhood operator is a *linear filter*, where an output pixel's value is a weighted sum of pixel values within a small neighborhood $\mathcal{N}$ (Figure 3.10),

$$g(i, j) = \sum_{k, l} f(i + k, j + l) h(k, l). \quad (3.12)$$

The entries in the weight *kernel* or *mask* $h(k, l)$ are often called the *filter coefficients*. The above *correlation* operator can be more compactly notated as

$$g = f \otimes h. \quad (3.13)$$

A common variant on this formula is

$$g(i, j) = \sum_{k, l} f(i - k, j - l) h(k, l) = \sum_{k, l} f(k, l) h(i - k, j - l), \quad (3.14)$$

where the sign of the offsets in f has been reversed. This is called the *convolution* operator,

$$g = f * h, \quad (3.15)$$

and h is then called the *impulse response function*.⁵ The reason for this name is that the kernel function, h , convolved with an impulse signal, $\delta(i, j)$ (an image that is 0 everywhere except at the origin) reproduces itself, $h * \delta = h$, whereas correlation produces the reflected signal. (Try this yourself to verify that it is so.)

⁵The continuous version of convolution can be written as $g(\mathbf{x}) = \int f(\mathbf{x} - \mathbf{u}) h(\mathbf{u}) d\mathbf{u}$.

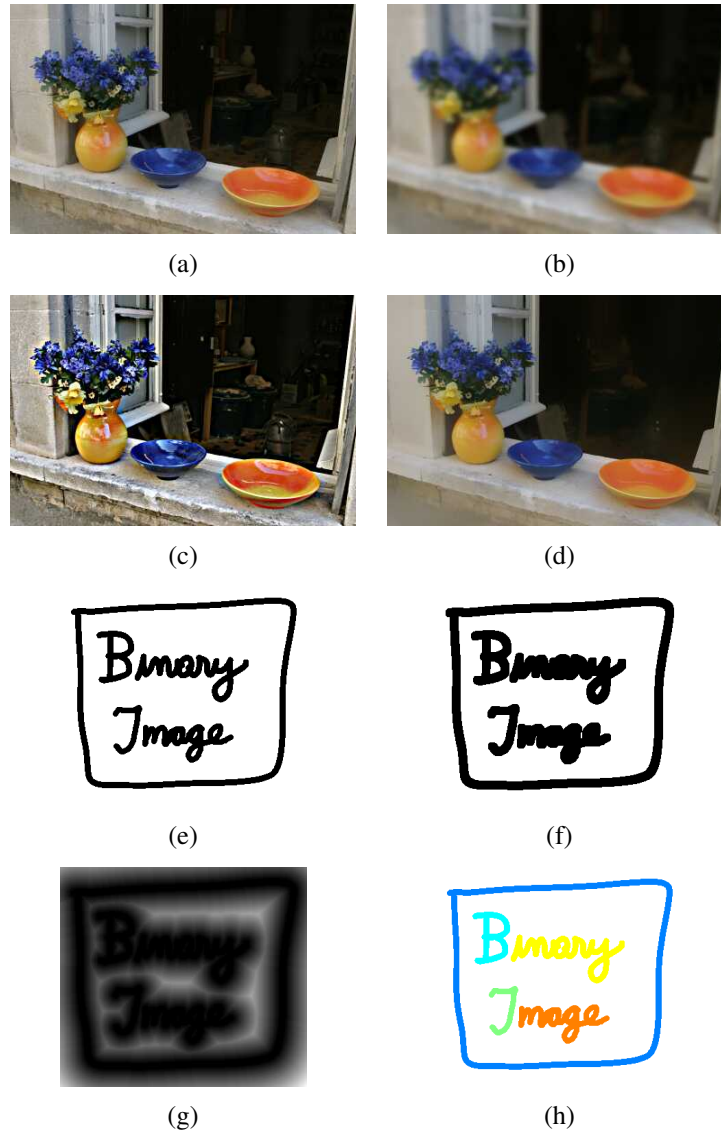


Figure 3.11 Some neighborhood operations: (a) original image; (b) blurred; (c) sharpened; (d) smoothed with edge-preserving filter; (e) binary image; (f) dilated; (g) distance transform; (h) connected components. For the dilation and connected components, black (ink) pixels are assumed to be active, i.e., to have a value of 1 in Equations (3.44–3.48).

$$\begin{bmatrix} 72 & 88 & 62 & 52 & 37 \end{bmatrix} * \begin{bmatrix} 1/4 & 1/2 & 1/4 \end{bmatrix} \Leftrightarrow \frac{1}{4} \begin{bmatrix} 2 & 1 & . & . & . \\ 1 & 2 & 1 & . & . \\ . & 1 & 2 & 1 & . \\ . & . & 1 & 2 & 1 \\ . & . & . & 1 & 2 \end{bmatrix} \begin{bmatrix} 72 \\ 88 \\ 62 \\ 52 \\ 37 \end{bmatrix}$$

Figure 3.12 One-dimensional signal convolution as a sparse matrix-vector multiplication, $\mathbf{g} = \mathbf{H}\mathbf{f}$.

In fact, Equation (3.14) can be interpreted as the superposition (addition) of shifted impulse response functions $h(i-k, j-l)$ multiplied by the input pixel values $f(k, l)$. Convolution has additional nice properties, e.g., it is both commutative and associative. As well, the Fourier transform of two convolved images is the product of their individual Fourier transforms (Section 3.4).

Both correlation and convolution are *linear shift-invariant* (LSI) operators, which obey both the superposition principle (3.5),

$$h \circ (f_0 + f_1) = h \circ f_0 + h \circ f_1, \quad (3.16)$$

and the *shift invariance* principle,

$$g(i, j) = f(i+k, j+l) \Leftrightarrow (h \circ g)(i, j) = (h \circ f)(i+k, j+l), \quad (3.17)$$

which means that shifting a signal commutes with applying the operator ($\circ$ stands for the LSI operator). Another way to think of shift invariance is that the operator “behaves the same everywhere”.

Occasionally, a shift-variant version of correlation or convolution may be used, e.g.,

$$g(i, j) = \sum_{k,l} f(i-k, j-l) h(k, l; i, j), \quad (3.18)$$

where $h(k, l; i, j)$ is the convolution kernel at pixel (i, j) . For example, such a spatially varying kernel can be used to model blur in an image due to variable depth-dependent defocus.

Correlation and convolution can both be written as a matrix-vector multiplication, if we first convert the two-dimensional images $f(i, j)$ and $g(i, j)$ into raster-ordered vectors $\mathbf{f}$ and $\mathbf{g}$,

$$\mathbf{g} = \mathbf{H}\mathbf{f}, \quad (3.19)$$

where the (sparse) $\mathbf{H}$ matrix contains the convolution kernels. Figure 3.12 shows how a one-dimensional convolution can be represented in matrix-vector form.

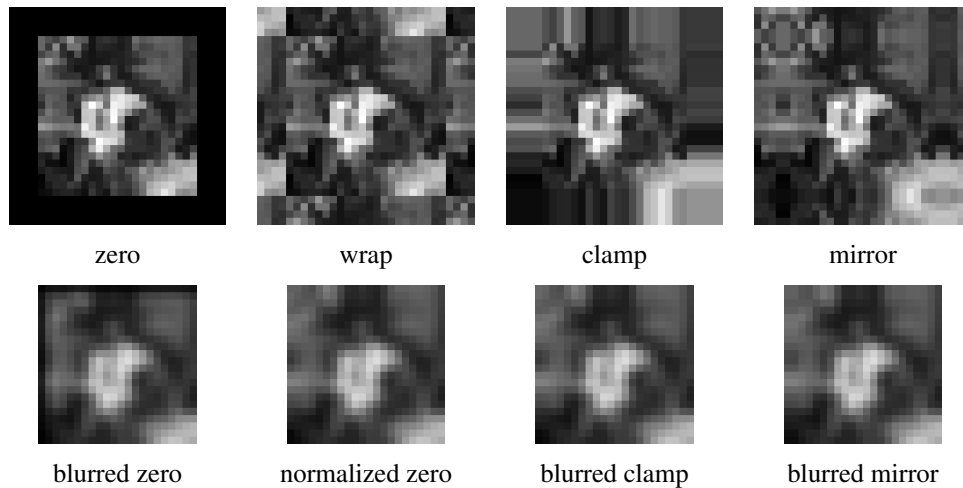


Figure 3.13 *Border padding (top row) and the results of blurring the padded image (bottom row). The normalized zero image is the result of dividing (normalizing) the blurred zero-padded RGBA image by its corresponding soft alpha value.*

Padding (border effects)

The astute reader will notice that the correlation shown in Figure 3.10 produces a result that is smaller than the original image, which may not be desirable in many applications.⁶ This is because the neighborhoods of typical correlation and convolution operations extend beyond the image boundaries near the edges, and so the filtered images suffer from *boundary effects*.

To deal with this, a number of different *padding* or extension modes have been developed for neighborhood operations (Figure 3.13):

- *zero*: set all pixels outside the source image to 0 (a good choice for alpha-matted cutout images);
- *constant (border color)*: set all pixels outside the source image to a specified *border* value;
- *clamp (replicate or clamp to edge)*: repeat edge pixels indefinitely;
- *(cyclic) wrap (repeat or tile)*: loop “around” the image in a “toroidal” configuration;
- *mirror*: reflect pixels across the image edge;

⁶Note, however, that early convolutional networks such as LeNet (LeCun, Bottou *et al.* 1998) adopted this structure.

- *extend*: extend the signal by subtracting the mirrored version of the signal from the edge pixel value.

In the computer graphics literature (Akenine-Möller and Haines 2002, p. 124), these mechanisms are known as the *wrapping mode* (OpenGL) or *texture addressing mode* (Direct3D). The formulas for these modes are left to the reader (Exercise 3.9).

Figure 3.13 shows the effects of padding an image with each of the above mechanisms and then blurring the resulting padded image. As you can see, zero padding darkens the edges, clamp (replication) padding propagates border values inward, mirror (reflection) padding preserves colors near the borders. Extension padding (not shown) keeps the border pixels fixed (during blur).

An alternative to padding is to blur the zero-padded RGBA image and to then divide the resulting image by its alpha value to remove the darkening effect. The results can be quite good, as seen in the normalized zero image in Figure 3.13.

3.2.1 Separable filtering

The process of performing a convolution requires K^2 (multiply-add) operations per pixel, where K is the size (width or height) of the convolution kernel, e.g., the box filter in Figure 3.14a. In many cases, this operation can be significantly sped up by first performing a one-dimensional horizontal convolution followed by a one-dimensional vertical convolution, which requires a total of $2K$ operations per pixel. A convolution kernel for which this is possible is said to be *separable*.

It is easy to show that the two-dimensional kernel $\mathbf{K}$ corresponding to successive convolution with a horizontal kernel $\mathbf{h}$ and a vertical kernel $\mathbf{v}$ is the *outer product* of the two kernels,

$$\mathbf{K} = \mathbf{v}\mathbf{h}^T \quad (3.20)$$

(see Figure 3.14 for some examples). Because of the increased efficiency, the design of convolution kernels for computer vision applications is often influenced by their separability.

How can we tell if a given kernel $\mathbf{K}$ is indeed separable? This can often be done by inspection or by looking at the analytic form of the kernel (Freeman and Adelson 1991). A more direct method is to treat the 2D kernel as a 2D matrix $\mathbf{K}$ and to take its singular value decomposition (SVD),

$$\mathbf{K} = \sum_i \sigma_i \mathbf{u}_i \mathbf{v}_i^T \quad (3.21)$$

(see Appendix A.1.1 for the definition of the SVD). If only the first singular value σ_0 is non-zero, the kernel is separable and $\sqrt{\sigma_0} \mathbf{u}_0$ and $\sqrt{\sigma_0} \mathbf{v}_0^T$ provide the vertical and horizontal

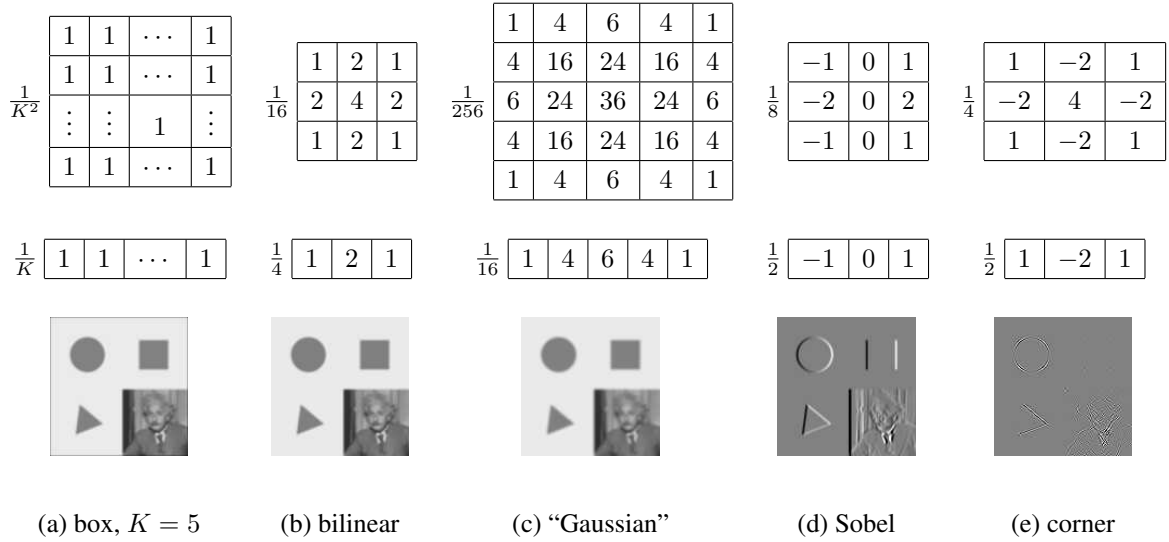


Figure 3.14 *Separable linear filters: For each image (a)–(e), we show the 2D filter kernel (top), the corresponding horizontal 1D kernel (middle), and the filtered image (bottom). The filtered Sobel and corner images are signed, scaled up by $2\times$ and $4\times$, respectively, and added to a gray offset before display.*

kernels (Perona 1995). For example, the Laplacian of Gaussian kernel (3.26 and 7.23) can be implemented as the sum of two separable filters (7.24) (Wiejak, Buxton, and Buxton 1985).

What if your kernel is not separable and yet you still want a faster way to implement it? Perona (1995), who first made the link between kernel separability and SVD, suggests using more terms in the (3.21) series, i.e., summing up a number of separable convolutions. Whether this is worth doing or not depends on the relative sizes of K and the number of significant singular values, as well as other considerations, such as cache coherency and memory locality.

3.2.2 Examples of linear filtering

Now that we have described the process for performing linear filtering, let us examine a number of frequently used filters.

The simplest filter to implement is the *moving average* or *box* filter, which simply averages the pixel values in a $K \times K$ window. This is equivalent to convolving the image with a kernel of all ones and then scaling (Figure 3.14a). For large kernels, a more efficient implementation is to slide a moving window across each scanline (in a separable filter) while adding the

newest pixel and subtracting the oldest pixel from the running sum. This is related to the concept of *summed area tables*, which we describe shortly.

A smoother image can be obtained by separably convolving the image with a piecewise linear “tent” function (also known as a *Bartlett* filter). Figure 3.14b shows a 3×3 version of this filter, which is called the *bilinear* kernel, since it is the outer product of two linear (first-order) splines (see Section 3.5.2).

Convolving the linear tent function with itself yields the cubic approximating spline, which is called the “Gaussian” kernel (Figure 3.14c) in Burt and Adelson’s (1983a) *Laplacian pyramid* representation (Section 3.5). Note that approximate Gaussian kernels can also be obtained by iterated convolution with box filters (Wells 1986). In applications where the filters really need to be rotationally symmetric, carefully tuned versions of sampled Gaussians should be used (Freeman and Adelson 1991) (Exercise 3.11).

The kernels we just discussed are all examples of blurring (smoothing) or *low-pass* kernels, since they pass through the lower frequencies while attenuating higher frequencies. How good are they at doing this? In Section 3.4, we use frequency-space Fourier analysis to examine the exact frequency response of these filters. We also introduce the *sinc* $((\sin x)/x)$ filter, which performs *ideal* low-pass filtering.

In practice, smoothing kernels are often used to reduce high-frequency noise. We have much more to say about using variants of smoothing to remove noise later (see Sections 3.3.1, 3.4, and as well as Chapters 4 and 5).

Surprisingly, smoothing kernels can also be used to *sharpen* images using a process called *unsharp masking*. Since blurring the image reduces high frequencies, adding some of the difference between the original and the blurred image makes it sharper,

$$g_{\text{sharp}} = f + \gamma(f - h_{\text{blur}} * f). \quad (3.22)$$

In fact, before the advent of digital photography, this was the standard way to sharpen images in the darkroom: create a blurred (“positive”) negative from the original negative by misfocusing, then overlay the two negatives before printing the final image, which corresponds to

$$g_{\text{unsharp}} = f(1 - \gamma h_{\text{blur}} * f). \quad (3.23)$$

This is no longer a linear filter but it still works well.

Linear filtering can also be used as a pre-processing stage to edge extraction (Section 7.2) and interest point detection (Section 7.1) algorithms. Figure 3.14d shows a simple 3×3 edge extractor called the Sobel operator, which is a separable combination of a horizontal *central difference* (so called because the horizontal derivative is centered on the pixel) and a vertical

tent filter (to smooth the results). As you can see in the image below the kernel, this filter effectively emphasizes vertical edges.

The simple corner detector (Figure 3.14e) looks for simultaneous horizontal and vertical second derivatives. As you can see, however, it responds not only to the corners of the square, but also along diagonal edges. Better corner detectors, or at least interest point detectors that are more rotationally invariant, are described in Section 7.1.

3.2.3 Band-pass and steerable filters

The Sobel and corner operators are simple examples of band-pass and oriented filters. More sophisticated kernels can be created by first smoothing the image with a (unit area) Gaussian filter,

$$G(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \quad (3.24)$$

and then taking the first or second derivatives (Marr 1982; Witkin 1983; Freeman and Adelson 1991). Such filters are known collectively as *band-pass filters*, since they filter out both low and high frequencies.

The (undirected) second derivative of a two-dimensional image,

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}, \quad (3.25)$$

is known as the *Laplacian* operator. Blurring an image with a Gaussian and then taking its Laplacian is equivalent to convolving directly with the *Laplacian of Gaussian* (LoG) filter,

$$\nabla^2 G(x, y; \sigma) = \left(\frac{x^2 + y^2}{\sigma^4} - \frac{2}{\sigma^2} \right) G(x, y; \sigma), \quad (3.26)$$

which has certain nice *scale-space properties* (Witkin 1983; Witkin, Terzopoulos, and Kass 1986). The five-point Laplacian is just a compact approximation to this more sophisticated filter.

Likewise, the Sobel operator is a simple approximation to a *directional* or *oriented* filter, which can be obtained by smoothing with a Gaussian (or some other filter) and then taking a *directional derivative* $\nabla_{\hat{\mathbf{u}}} = \frac{\partial}{\partial \hat{\mathbf{u}}}$, which is obtained by taking the dot product between the gradient field ∇ and a unit direction $\hat{\mathbf{u}} = (\cos \theta, \sin \theta)$,

$$\hat{\mathbf{u}} \cdot \nabla(G * f) = \nabla_{\hat{\mathbf{u}}}(G * f) = (\nabla_{\hat{\mathbf{u}}}G) * f. \quad (3.27)$$

The smoothed directional derivative filter,

$$G_{\hat{\mathbf{u}}} = uG_x + vG_y = u \frac{\partial G}{\partial x} + v \frac{\partial G}{\partial y}, \quad (3.28)$$

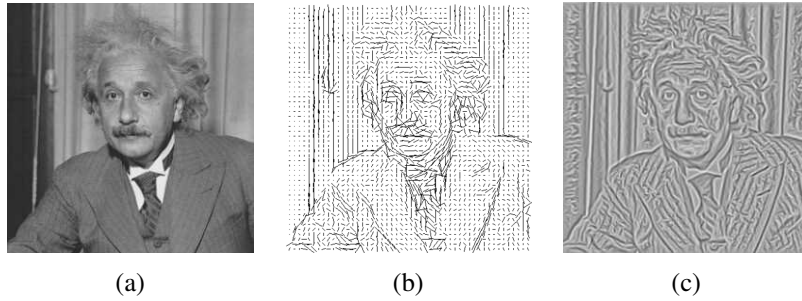


Figure 3.15 *Second-order steerable filter (Freeman 1992) © 1992 IEEE: (a) original image of Einstein; (b) orientation map computed from the second-order oriented energy; (c) original image with oriented structures enhanced.*

where $\hat{\mathbf{u}} = (u, v)$, is an example of a *steerable* filter, since the value of an image convolved with $G_{\hat{\mathbf{u}}}$ can be computed by first convolving with the pair of filters (G_x, G_y) and then *steering* the filter (potentially locally) by multiplying this gradient field with a unit vector $\hat{\mathbf{u}}$ (Freeman and Adelson 1991). The advantage of this approach is that a whole *family* of filters can be evaluated with very little cost.

How about steering a directional second derivative filter $\nabla_{\hat{\mathbf{u}}} \cdot \nabla_{\hat{\mathbf{u}}} G$, which is the result of taking a (smoothed) directional derivative and then taking the directional derivative again? For example, G_{xx} is the second directional derivative in the x direction.

At first glance, it would appear that the steering trick will not work, since for every direction $\hat{\mathbf{u}}$, we need to compute a different first directional derivative. Somewhat surprisingly, Freeman and Adelson (1991) showed that, for directional Gaussian derivatives, it is possible to steer *any* order of derivative with a relatively small number of basis functions. For example, only three basis functions are required for the second-order directional derivative,

$$G_{\hat{\mathbf{u}}\hat{\mathbf{u}}} = u^2 G_{xx} + 2uv G_{xy} + v^2 G_{yy}. \quad (3.29)$$

Furthermore, each of the basis filters, while not itself necessarily separable, can be computed using a linear combination of a small number of separable filters (Freeman and Adelson 1991).

This remarkable result makes it possible to construct directional derivative filters of increasingly greater *directional selectivity*, i.e., filters that only respond to edges that have strong local consistency in orientation (Figure 3.15). Furthermore, higher order steerable filters can respond to potentially more than a single edge orientation at a given location, and they can respond to both *bar* edges (thin lines) and the classic step edges (Figure 3.16). In order to do this, however, full *Hilbert transform pairs* need to be used for second-order and

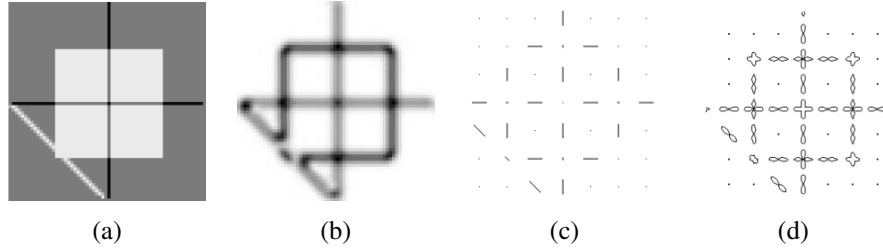


Figure 3.16 Fourth-order steerable filter (Freeman and Adelson 1991) © 1991 IEEE: (a) test image containing bars (lines) and step edges at different orientations; (b) average oriented energy; (c) dominant orientation; (d) oriented energy as a function of angle (polar plot).

higher filters, as described in (Freeman and Adelson 1991).

Steerable filters are often used to construct both feature descriptors (Section 7.1.3) and edge detectors (Section 7.2). While the filters developed by Freeman and Adelson (1991) are best suited for detecting linear (edge-like) structures, more recent work by Koethe (2003) shows how a combined 2×2 boundary tensor can be used to encode both edge and junction (“corner”) features. Exercise 3.13 has you implement such steerable filters and apply them to finding both edge and corner features.

Summed area table (integral image)

If an image is going to be repeatedly convolved with different box filters (and especially filters of different sizes at different locations), you can precompute the *summed area table* (Crow 1984), which is just the running sum of all the pixel values from the origin,

$$s(i, j) = \sum_{k=0}^i \sum_{l=0}^j f(k, l). \quad (3.30)$$

This can be efficiently computed using a recursive (raster-scan) algorithm,

$$s(i, j) = s(i-1, j) + s(i, j-1) - s(i-1, j-1) + f(i, j). \quad (3.31)$$

The image $s(i, j)$ is also often called an *integral image* (see Figure 3.17) and can actually be computed using only two additions per pixel if separate row sums are used (Viola and Jones 2004). To find the summed area (integral) inside a rectangle $[i_0, i_1] \times [j_0, j_1]$, we simply combine four samples from the summed area table,

$$S(i_0 \dots i_1, j_0 \dots j_1) = s(i_1, j_1) - s(i_1, j_0 - 1) - s(i_0 - 1, j_1) + s(i_0 - 1, j_0 - 1). \quad (3.32)$$

3	2	7	2	3
1	5	1	3	4
5	1	3	5	1
4	3	2	1	6
2	4	1	4	8

(a) $S = 24$

3	5	12	14	17
4	11	19	24	31
9	17	28	38	46
13	24	37	48	62
15	30	44	59	81

(b) $s = 28$

3	5	12	14	17
4	11	19	24	31
9	17	28	38	46
13	24	37	48	62
15	30	44	59	81

(c) $S = 24$

Figure 3.17 Summed area tables: (a) original image; (b) summed area table; (c) computation of area sum. Each value in the summed area table $s(i, j)$ (red) is computed recursively from its three adjacent (blue) neighbors (3.31). Area sums S (green) are computed by combining the four values at the rectangle corners (purple) (3.32). Positive values are shown in **bold** and negative values in *italics*.

A potential disadvantage of summed area tables is that they require $\log M + \log N$ extra bits in the accumulation image compared to the original image, where M and N are the image width and height. Extensions of summed area tables can also be used to approximate other convolution kernels (Wolberg (1990, Section 6.5.2) contains a review).

In computer vision, summed area tables have been used in face detection (Viola and Jones 2004) to compute simple multi-scale low-level features. Such features, which consist of adjacent rectangles of positive and negative values, are also known as *boxlets* (Simard, Bottou *et al.* 1998). In principle, summed area tables could also be used to compute the sums in the sum of squared differences (SSD) stereo and motion algorithms (Section 12.4). In practice, separable moving average filters are usually preferred (Kanade, Yoshida *et al.* 1996), unless many different window shapes and sizes are being considered (Veksler 2003).

Recursive filtering

The incremental formula (3.31) for the summed area is an example of a *recursive filter*, i.e., one whose values depends on previous filter outputs. In the signal processing literature, such filters are known as *infinite impulse response* (IIR), since the output of the filter to an impulse (single non-zero value) goes on forever. For example, for a summed area table, an impulse generates an infinite rectangle of 1s below and to the right of the impulse. The filters we have previously studied in this chapter, which involve the image with a finite extent kernel, are known as *finite impulse response* (FIR).

Two-dimensional IIR filters and recursive formulas are sometimes used to compute quan-

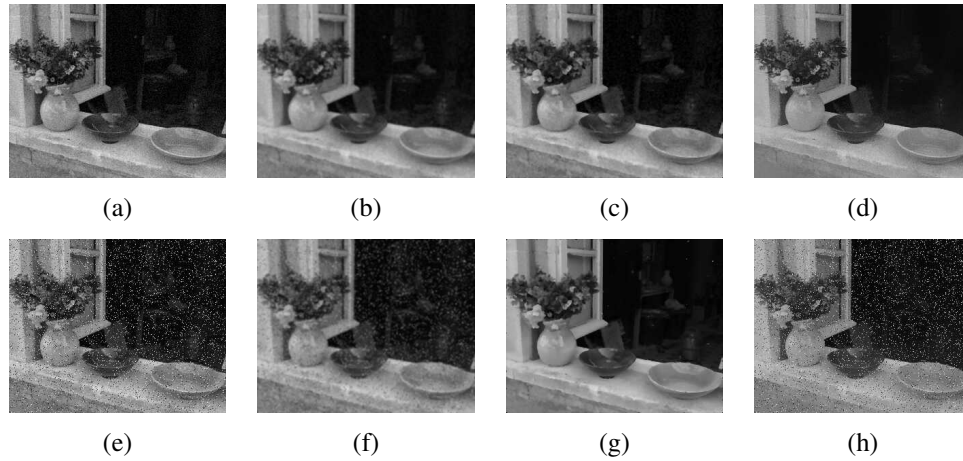


Figure 3.18 *Median and bilateral filtering: (a) original image with Gaussian noise; (b) Gaussian filtered; (c) median filtered; (d) bilaterally filtered; (e) original image with shot noise; (f) Gaussian filtered; (g) median filtered; (h) bilaterally filtered. Note that the bilateral filter fails to remove the shot noise because the noisy pixels are too different from their neighbors.*

ties that involve large area interactions, such as two-dimensional distance functions (Section 3.3.3) and connected components (Section 3.3.3).

More commonly, however, IIR filters are used inside one-dimensional separable filtering stages to compute large-extent smoothing kernels, such as efficient approximations to Gaussians and edge filters (Deriche 1990; Nielsen, Florack, and Deriche 1997). Pyramid-based algorithms (Section 3.5) can also be used to perform such large-area smoothing computations.

3.3 More neighborhood operators

As we have just seen, linear filters can perform a wide variety of image transformations. However non-linear filters, such as edge-preserving median or bilateral filters, can sometimes perform even better. Other examples of neighborhood operators include *morphological* operators that operate on binary images, as well as *semi-global* operators that compute *distance transforms* and find *connected components* in binary images (Figure 3.11f–h).

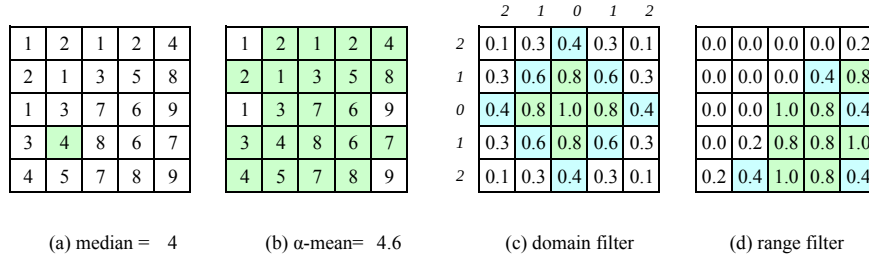


Figure 3.19 Median and bilateral filtering: (a) median pixel (green); (b) selected α -trimmed mean pixels; (c) domain filter (numbers along edge are pixel distances); (d) range filter.

3.3.1 Non-linear filtering

The filters we have looked at so far have all been *linear*, i.e., their response to a sum of two signals is the same as the sum of the individual responses. This is equivalent to saying that each output pixel is a weighted summation of some number of input pixels (3.19). Linear filters are easier to compose and are amenable to frequency response analysis (Section 3.4).

In many cases, however, better performance can be obtained by using a *non-linear* combination of neighboring pixels. Consider for example the image in Figure 3.18e, where the noise, rather than being Gaussian, is *shot noise*, i.e., it occasionally has very large values. In this case, regular blurring with a Gaussian filter fails to remove the noisy pixels and instead turns them into softer (but still visible) spots (Figure 3.18f).

Median filtering

A better filter to use in this case is the *median* filter, which selects the median value from each pixel's neighborhood (Figure 3.19a). Median values can be computed in expected linear time using a randomized select algorithm (Cormen 2001) and incremental variants have also been developed (Tomasi and Manduchi 1998; Bovik 2000, Section 3.2), as well as a constant time algorithm that is independent of window size (Perreault and Hébert 2007). Since the shot noise value usually lies well outside the true values in the neighborhood, the median filter is able to filter away such bad pixels (Figure 3.18g).

One downside of the median filter, in addition to its moderate computational cost, is that because it selects only one input pixel value to replace each output pixel, it is not as *efficient* at averaging away regular Gaussian noise (Huber 1981; Hampel, Ronchetti *et al.* 1986; Stewart 1999). A better choice may be the α -trimmed mean (Lee and Redner 1990; Crane 1997,

p. 109), which averages together all of the pixels except for the α fraction that are the smallest and the largest (Figure 3.19b).

Another possibility is to compute a *weighted median*, in which each pixel is used a number of times depending on its distance from the center. This turns out to be equivalent to minimizing the weighted objective function

$$\sum_{k,l} w(k,l) |f(i+k, j+l) - g(i,j)|^p, \quad (3.33)$$

where $g(i,j)$ is the desired output value and $p = 1$ for the weighted median. The value $p = 2$ is the usual *weighted mean*, which is equivalent to correlation (3.12) after normalizing by the sum of the weights (Haralick and Shapiro 1992, Section 7.2.6; Bovik 2000, Section 3.2). The weighted mean also has deep connections to other methods in robust statistics (see Appendix B.3), such as influence functions (Huber 1981; Hampel, Ronchetti *et al.* 1986).

Non-linear smoothing has another, perhaps even more important property, especially as shot noise is rare in today's cameras. Such filtering is more *edge preserving*, i.e., it has less tendency to soften edges while filtering away high-frequency noise.

Consider the noisy image in Figure 3.18a. In order to remove most of the noise, the Gaussian filter is forced to smooth away high-frequency detail, which is most noticeable near strong edges. Median filtering does better but, as mentioned before, does not do as well at smoothing away from discontinuities. See Tomasi and Manduchi (1998) for some additional references to edge-preserving smoothing techniques.

While we could try to use the α -trimmed mean or weighted median, these techniques still have a tendency to round sharp corners, since the majority of pixels in the smoothing area come from the background distribution.

3.3.2 Bilateral filtering

What if we were to combine the idea of a weighted filter kernel with a better version of outlier rejection? What if instead of rejecting a fixed percentage α , we simply reject (in a soft way) pixels whose *values* differ too much from the central pixel value? This is the essential idea in *bilateral filtering*, which was first popularized in the computer vision community by Tomasi and Manduchi (1998), although it had been proposed earlier by Aurich and Weule (1995) and Smith and Brady (1997). Paris, Kornprobst *et al.* (2008) provide a nice review of work in this area as well as myriad applications in computer vision, graphics, and computational photography.

In the bilateral filter, the output pixel value depends on a weighted combination of neigh-

boring pixel values

$$\mathbf{g}(i, j) = \frac{\sum_{k,l} \mathbf{f}(k, l) w(i, j, k, l)}{\sum_{k,l} w(i, j, k, l)}. \quad (3.34)$$

The weighting coefficient $w(i, j, k, l)$ depends on the product of a *domain kernel*, (Figure 3.19c),

$$d(i, j, k, l) = \exp \left(-\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2} \right), \quad (3.35)$$

and a data-dependent *range kernel* (Figure 3.19d),

$$r(i, j, k, l) = \exp \left(-\frac{\|\mathbf{f}(i, j) - \mathbf{f}(k, l)\|^2}{2\sigma_r^2} \right). \quad (3.36)$$

When multiplied together, these yield the data-dependent *bilateral weight function*

$$w(i, j, k, l) = \exp \left(-\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2} - \frac{\|\mathbf{f}(i, j) - \mathbf{f}(k, l)\|^2}{2\sigma_r^2} \right). \quad (3.37)$$

Figure 3.20 shows an example of the bilateral filtering of a noisy step edge. Note how the domain kernel is the usual Gaussian, the range kernel measures appearance (intensity) similarity to the center pixel, and the bilateral filter kernel is a product of these two.

Notice that for color images, the range filter (3.36) uses the *vector distance* between the center and the neighboring pixel. This is important in color images, since an edge in any *one* of the color bands signals a change in material and hence the need to downweight a pixel's influence.⁷

Since bilateral filtering is quite slow compared to regular separable filtering, a number of acceleration techniques have been developed, as discussed in Durand and Dorsey (2002), Paris and Durand (2009), Chen, Paris, and Durand (2007), and Paris, Kornprobst *et al.* (2008). In particular, the *bilateral grid* (Chen, Paris, and Durand 2007), which subsamples the higher-dimensional color/position space on a uniform grid, continues to be widely used, including the application of the *bilateral solver* (Section 4.2.3 and Barron and Poole (2016)). An even faster implementation of bilateral filtering can be obtained using the *permutohedral lattice* approach developed by Adams, Baek, and Davis (2010).

Iterated adaptive smoothing and anisotropic diffusion

Bilateral (and other) filters can also be applied in an iterative fashion, especially if an appearance more like a “cartoon” is desired (Tomasi and Manduchi 1998). When iterated filtering is applied, a much smaller neighborhood can often be used.

⁷Tomasi and Manduchi (1998) show that using the vector distance (as opposed to filtering each color band separately) reduces color fringing effects. They also recommend taking the color difference in the more perceptually uniform CIELAB color space (see Section 2.3.2).

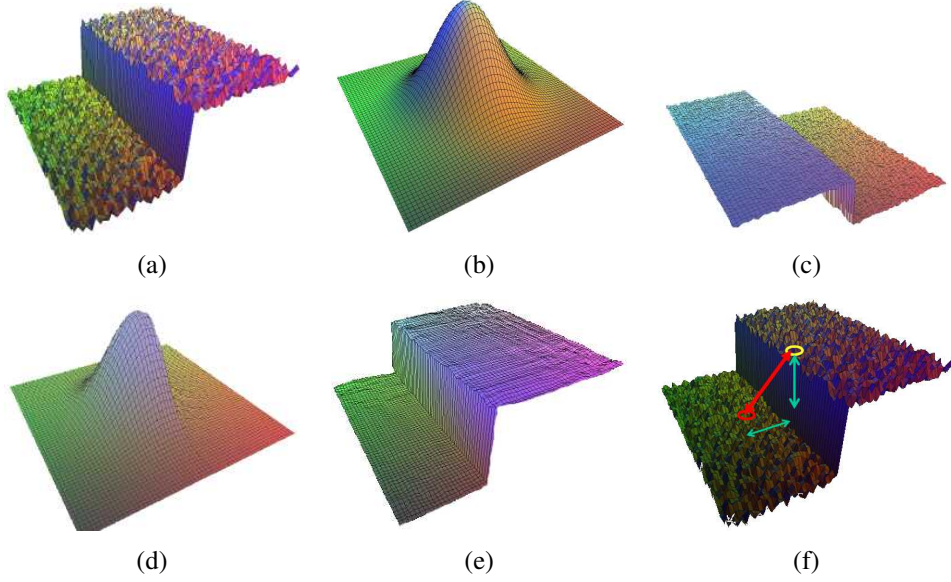


Figure 3.20 Bilateral filtering (Durand and Dorsey 2002) © 2002 ACM: (a) noisy step edge input; (b) domain filter (Gaussian); (c) range filter (similarity to center pixel value); (d) bilateral filter; (e) filtered step edge output; (f) 3D distance between pixels.

Consider, for example, using only the four nearest neighbors, i.e., restricting $|k - i| + |l - j| \leq 1$ in (3.34). Observe that

$$d(i, j, k, l) = \exp \left(-\frac{(i - k)^2 + (j - l)^2}{2\sigma_d^2} \right) \quad (3.38)$$

$$= \begin{cases} 1, & |k - i| + |l - j| = 0, \\ e^{-1/2\sigma_d^2}, & |k - i| + |l - j| = 1. \end{cases} \quad (3.39)$$

We can thus re-write (3.34) as

$$\begin{aligned} f^{(t+1)}(i, j) &= \frac{f^{(t)}(i, j) + \eta \sum_{k,l} f^{(t)}(k, l) r(i, j, k, l)}{1 + \eta \sum_{k,l} r(i, j, k, l)} \\ &= f^{(t)}(i, j) + \frac{\eta}{1 + \eta R} \sum_{k,l} r(i, j, k, l) [f^{(t)}(k, l) - f^{(t)}(i, j)], \end{aligned} \quad (3.40)$$

where $R = \sum_{(k,l)} r(i, j, k, l)$, (k, l) are the $\mathcal{N}_4$ (nearest four) neighbors of (i, j) , and we have made the iterative nature of the filtering explicit.

As Barash (2002) notes, (3.40) is the same as the discrete *anisotropic diffusion* equation

first proposed by Perona and Malik (1990b).⁸ Since its original introduction, anisotropic diffusion has been extended and applied to a wide range of problems (Nielsen, Florack, and Deriche 1997; Black, Sapiro *et al.* 1998; Weickert, ter Haar Romeny, and Viergever 1998; Weickert 1998). It has also been shown to be closely related to other *adaptive smoothing* techniques (Saint-Marc, Chen, and Medioni 1991; Barash 2002; Barash and Comaniciu 2004) as well as Bayesian regularization with a non-linear smoothness term that can be derived from image statistics (Schar, Black, and Haussecker 2003).

In its general form, the range kernel $r(i, j, k, l) = r(\|f(i, j) - f(k, l)\|)$, which is usually called the *gain* or *edge-stopping* function, or diffusion coefficient, can be any monotonically increasing function with $r'(x) \rightarrow 0$ as $x \rightarrow \infty$. Black, Sapiro *et al.* (1998) show how anisotropic diffusion is equivalent to minimizing a robust penalty function on the image gradients, which we discuss in Sections 4.2 and 4.3. Schar, Black, and Haussecker (2003) show how the edge-stopping function can be derived in a principled manner from local image statistics. They also extend the diffusion neighborhood from $\mathcal{N}_4$ to $\mathcal{N}_8$, which allows them to create a diffusion operator that is both rotationally invariant and incorporates information about the eigenvalues of the local structure tensor.

Note that, without a bias term towards the original image, anisotropic diffusion and iterative adaptive smoothing converge to a constant image. Unless a small number of iterations is used (e.g., for speed), it is usually preferable to formulate the smoothing problem as a joint minimization of a smoothness term and a data fidelity term, as discussed in Sections 4.2 and 4.3 and by Schar, Black, and Haussecker (2003), which introduce such a bias in a principled manner.

Guided image filtering

While so far we have discussed techniques for filtering an image to obtain an improved version, e.g., one with less noise or sharper edges, it is also possible to use a different *guide* image to adaptively filter a noisy input (Eisemann and Durand 2004; Petschnigg, Agrawala *et al.* 2004; He, Sun, and Tang 2013). An example of this is using a *flash* image, which has strong edges but poor color, to adaptively filter a low-light non-flash color image, which has large amounts of noise, as described in Section 10.2.2. In their papers, where they apply the range filter (3.36) to a different guide image $\mathbf{h}()$, Eisemann and Durand (2004) call their approach a *cross-bilateral filter*, while Petschnigg, Agrawala *et al.* (2004) call it *joint bilateral filtering*.

He, Sun, and Tang (2013) point out that these papers are just two examples of the more general concept of *guided image filtering*, where the guide image $\mathbf{h}()$ is used to compute the

⁸The $1/(1 + \eta R)$ factor is not present in anisotropic diffusion but becomes negligible as $\eta \rightarrow 0$.

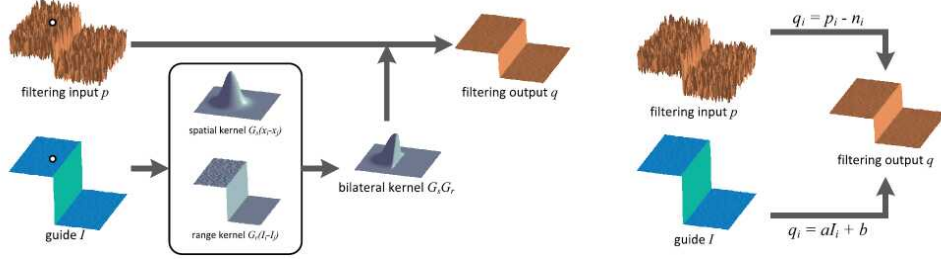


Figure 3.21 Guided image filtering (He, Sun, and Tang 2013) © 2013 IEEE. Unlike joint bilateral filtering, shown on the left, which computes a per pixel weight mask from the guide image (shown as I in the figure, but $\mathbf{h}$ in the text), the guided image filter models the output value (shown as q_i in the figure, but denoted as $\mathbf{g}(i, j)$ in the text) as a local affine transformation of the guide pixels.

locally adapted inter-pixel weights $w(i, j, k, l)$, i.e.,

$$\mathbf{g}(i, j) = \sum_{k, l} w(\mathbf{h}; i, j, k, l) \mathbf{f}(k, l). \quad (3.41)$$

In their paper, the authors suggest modeling the relationship between the guide and input images using a local affine transformation,

$$\mathbf{g}(i, j) = \mathbf{A}_{k, l} \mathbf{h}(i, j) + \mathbf{b}_{k, l}, \quad (3.42)$$

where the estimates for $\mathbf{A}_{k, l}$ and $\mathbf{b}_{k, l}$ are obtained from a regularized least squares fit over a square neighborhood centered around pixel (k, l) , i.e., minimizing

$$\sum_{(i, j) \in \mathcal{N}_{k, l}} \|\mathbf{A}_{k, l} \mathbf{h}(i, j) + \mathbf{b}_{k, l} - \mathbf{f}(i, j)\|^2 + \lambda \|\mathbf{A}\|^2. \quad (3.43)$$

These kinds of regularized least squares problems are called *ridge regression* (Section 4.1). The concept behind this algorithm is illustrated in Figure 3.21.

Instead of just taking the predicted value of the filtered pixel $\mathbf{g}(i, j)$ from the window centered on that pixel, an average across all windows that cover the pixel is used. The resulting algorithm (He, Sun, and Tang 2013, Algorithm 1) consists of a series of local mean image and image moment filters, a per-pixel linear system solve (which reduces to a division if the guide image is scalar), and another set of filtering steps. The authors describe how this fast and simple process has been applied to a wide variety of computer vision problems, including image matting (Section 10.4.3), high dynamic range image tone mapping (Section 10.2.1), stereo matching (Hosni, Rhemann *et al.* 2013), and image denoising.

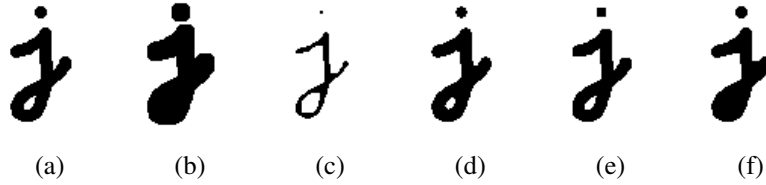


Figure 3.22 *Binary image morphology: (a) original image; (b) dilation; (c) erosion; (d) majority; (e) opening; (f) closing. The structuring element for all examples is a 5×5 square. The effects of majority are a subtle rounding of sharp corners. Opening fails to eliminate the dot, as it is not wide enough.*

3.3.3 Binary image processing

While non-linear filters are often used to enhance grayscale and color images, they are also used extensively to process binary images. Such images often occur after a *thresholding* operation,

$$\theta(f, t) = \begin{cases} 1 & \text{if } f \geq t, \\ 0 & \text{else,} \end{cases} \quad (3.44)$$

e.g., converting a scanned grayscale document into a binary image for further processing, such as *optical character recognition*.

Morphology

The most common binary image operations are called *morphological operations*, because they change the *shape* of the underlying binary objects (Ritter and Wilson 2000, Chapter 7). To perform such an operation, we first convolve the binary image with a binary *structuring element* and then select a binary output value depending on the thresholded result of the convolution. (This is not the usual way in which these operations are described, but I find it a nice simple way to unify the processes.) The structuring element can be any shape, from a simple 3×3 box filter, to more complicated disc structures. It can even correspond to a particular shape that is being sought for in the image.

Figure 3.22 shows a close-up of the convolution of a binary image f with a 3×3 structuring element s and the resulting images for the operations described below. Let

$$c = f \otimes s \quad (3.45)$$

be the integer-valued *count* of the number of 1s inside each structuring element as it is scanned over the image and S be the size of the structuring element (number of pixels). The standard operations used in binary morphology include:

- **dilation**: $\text{dilate}(f, s) = \theta(c, 1)$;
- **erosion**: $\text{erode}(f, s) = \theta(c, S)$;
- **majority**: $\text{maj}(f, s) = \theta(c, S/2)$;
- **opening**: $\text{open}(f, s) = \text{dilate}(\text{erode}(f, s), s)$;
- **closing**: $\text{close}(f, s) = \text{erode}(\text{dilate}(f, s), s)$.

As we can see from Figure 3.22, dilation grows (thickens) objects consisting of 1s, while erosion shrinks (thins) them. The opening and closing operations tend to leave large regions and smooth boundaries unaffected, while removing small objects or holes and smoothing boundaries.

While we will not use mathematical morphology much in the rest of this book, it is a handy tool to have around whenever you need to clean up some thresholded images. You can find additional details on morphology in other textbooks on computer vision and image processing (Haralick and Shapiro 1992, Section 5.2; Bovik 2000, Section 2.2; Ritter and Wilson 2000, Section 7) as well as articles and books specifically on this topic (Serra 1982; Serra and Vincent 1992; Yuille, Vincent, and Geiger 1992; Soille 2006).

Distance transforms

The distance transform is useful in quickly precomputing the distance to a curve or set of points using a two-pass raster algorithm (Rosenfeld and Pfaltz 1966; Danielsson 1980; Borgefors 1986; Paglieroni 1992; Breu, Gil *et al.* 1995; Felzenszwalb and Huttenlocher 2012; Fabbri, Costa *et al.* 2008). It has many applications, including level sets (Section 7.3.2), fast *chamfer matching* (binary image alignment) (Huttenlocher, Klanderman, and Rucklidge 1993), feathering in image stitching and blending (Section 8.4.2), and nearest point alignment (Section 13.2.1).

The distance transform $D(i, j)$ of a binary image $b(i, j)$ is defined as follows. Let $d(k, l)$ be some *distance metric* between pixel offsets. Two commonly used metrics include the *city block* or *Manhattan* distance

$$d_1(k, l) = |k| + |l| \quad (3.46)$$

and the *Euclidean* distance

$$d_2(k, l) = \sqrt{k^2 + l^2}. \quad (3.47)$$

The distance transform is then defined as

$$D(i, j) = \min_{k, l: b(k, l)=0} d(i - k, j - l), \quad (3.48)$$

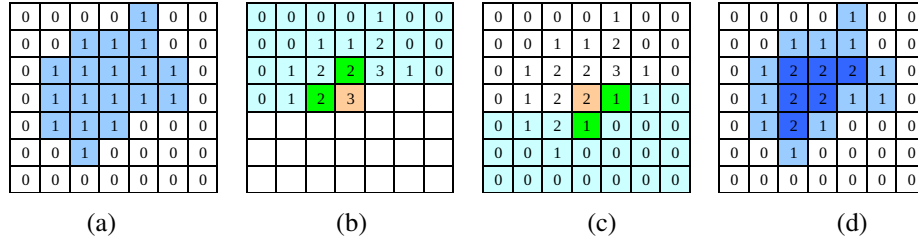


Figure 3.23 City block distance transform: (a) original binary image; (b) top to bottom (forward) raster sweep: green values are used to compute the orange value; (c) bottom to top (backward) raster sweep: green values are merged with old orange value; (d) final distance transform.

i.e., it is the distance to the *nearest* background pixel whose value is 0.

The D_1 city block distance transform can be efficiently computed using a forward and backward pass of a simple raster-scan algorithm, as shown in Figure 3.23. During the forward pass, each non-zero pixel in b is replaced by the minimum of 1 + the distance of its north or west neighbor. During the backward pass, the same occurs, except that the minimum is both over the current value D and 1 + the distance of the south and east neighbors (Figure 3.23).

Efficiently computing the Euclidean distance transform is more complicated (Danielsson 1980; Borgefors 1986). Here, just keeping the minimum scalar distance to the boundary during the two passes is not sufficient. Instead, a *vector-valued* distance consisting of both the x and y coordinates of the distance to the boundary must be kept and compared using the squared distance (hypotenuse) rule. As well, larger search regions need to be used to obtain reasonable results.

Figure 3.11g shows a distance transform computed from a binary image. Notice how the values grow away from the black (ink) regions and form ridges in the white area of the original image. Because of this linear growth from the starting boundary pixels, the distance transform is also sometimes known as the *grassfire transform*, since it describes the time at which a fire starting inside the black region would consume any given pixel, or a *chamfer*, because it resembles similar shapes used in woodworking and industrial design. The ridges in the distance transform become the *skeleton* (or *medial axis transform (MAT)*) of the region where the transform is computed, and consist of pixels that are of equal distance to two (or more) boundaries (Tek and Kimia 2003; Sebastian and Kimia 2005).

A useful extension of the basic distance transform is the *signed distance transform*, which computes distances to boundary pixels for *all* the pixels (Lavallée and Szeliski 1995). The simplest way to create this is to compute the distance transforms for both the original binary

image and its complement and to negate one of them before combining. Because such distance fields tend to be smooth, it is possible to store them more compactly (with minimal loss in *relative* accuracy) using a spline defined over a quadtree or octree data structure (Lavallée and Szeliski 1995; Szeliski and Lavallée 1996; Frisken, Perry *et al.* 2000). Such precomputed signed distance transforms can be extremely useful in efficiently aligning and merging 2D curves and 3D surfaces (Huttenlocher, Klanderman, and Rucklidge 1993; Szeliski and Lavallée 1996; Curless and Levoy 1996), especially if the *vectorial* version of the distance transform, i.e., a pointer from each pixel or voxel to the nearest boundary or surface element, is stored and interpolated. Signed distance fields are also an essential component of level set evolution (Section 7.3.2), where they are called *characteristic functions*.

Connected components

Another useful semi-global image operation is finding *connected components*, which are defined as regions of adjacent pixels that have the same input value or label. Pixels are said to be $\mathcal{N}_4$ adjacent if they are immediately horizontally or vertically adjacent, and $\mathcal{N}_8$ if they can also be diagonally adjacent. Both variants of connected components are widely used in a variety of applications, such as finding individual letters in a scanned document or finding objects (say, cells) in a thresholded image and computing their area statistics. Over the years, a wide variety of efficient algorithms have been developed to find such components, including the ones described in Haralick and Shapiro (1992, Section 2.3) and He, Ren *et al.* (2017). Such algorithms are usually included in image processing libraries such as OpenCV.

Once a binary or multi-valued image has been segmented into its connected components, it is often useful to compute the area statistics for each individual region $\mathcal{R}$. Such statistics include:

- the area (number of pixels);
- the perimeter (number of boundary pixels);
- the centroid (average x and y values);
- the second moments,

$$\mathbf{M} = \sum_{(x,y) \in \mathcal{R}} \begin{bmatrix} x - \bar{x} \\ y - \bar{y} \end{bmatrix} \begin{bmatrix} x - \bar{x} & y - \bar{y} \end{bmatrix}, \quad (3.49)$$

from which the major and minor axis orientation and lengths can be computed using eigenvalue analysis.

These statistics can then be used for further processing, e.g., for sorting the regions by the area size (to consider the largest regions first) or for preliminary matching of regions in different images.

3.4 Fourier transforms

In Section 3.2, we mentioned that Fourier analysis could be used to analyze the frequency characteristics of various filters. In this section, we explain both how Fourier analysis lets us determine these characteristics (i.e., the frequency *content* of an image) and how using the Fast Fourier Transform (FFT) lets us perform large-kernel convolutions in time that is independent of the kernel's size. More comprehensive introductions to Fourier transforms are provided by Bracewell (1986), Glassner (1995), Oppenheim and Schaffer (1996), and Oppenheim, Schaffer, and Buck (1999).

How can we analyze what a given filter does to high, medium, and low frequencies? The answer is to simply pass a sinusoid of known frequency through the filter and to observe by how much it is attenuated. Let

$$s(x) = \sin(2\pi f x + \phi_i) = \sin(\omega x + \phi_i) \quad (3.50)$$

be the input sinusoid whose *frequency* is f , *angular frequency* is $\omega = 2\pi f$, and *phase* is ϕ_i . Note that in this section, we use the variables x and y to denote the spatial coordinates of an image, rather than i and j as in the previous sections. This is both because the letters i and j are used for the *imaginary* number (the usage depends on whether you are reading complex variables or electrical engineering literature) and because it is clearer how to distinguish the horizontal (x) and vertical (y) components in frequency space. In this section, we use the letter j for the imaginary number, since that is the form more commonly found in the signal processing literature (Bracewell 1986; Oppenheim and Schaffer 1996; Oppenheim, Schaffer, and Buck 1999).

If we convolve the sinusoidal signal $s(x)$ with a filter whose impulse response is $h(x)$, we get another sinusoid of the same frequency but different magnitude A and phase ϕ_o ,

$$o(x) = h(x) * s(x) = A \sin(\omega x + \phi_o), \quad (3.51)$$

as shown in Figure 3.24. To see that this is the case, remember that a convolution can be expressed as a weighted summation of shifted input signals (3.14) and that the summation of a bunch of shifted sinusoids of the same frequency is just a single sinusoid at that frequency.⁹

⁹If h is a general (non-linear) transform, additional *harmonic* frequencies are introduced. This was traditionally

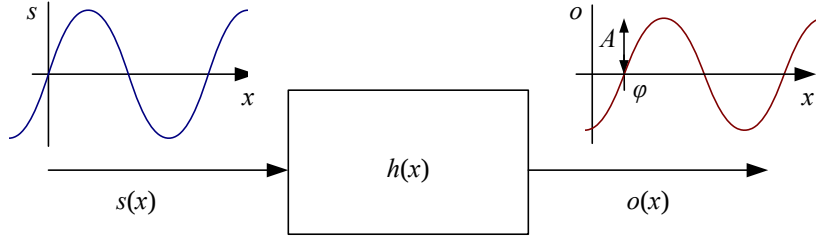


Figure 3.24 The Fourier Transform as the response of a filter $h(x)$ to an input sinusoid $s(x) = e^{j\omega x}$ yielding an output sinusoid $o(x) = h(x) * s(x) = Ae^{j(\omega x + \phi)}$.

The new magnitude A is called the *gain* or *magnitude* of the filter, while the phase difference $\Delta\phi = \phi_o - \phi_i$ is called the *shift* or *phase*.

In fact, a more compact notation is to use the complex-valued sinusoid

$$s(x) = e^{j\omega x} = \cos \omega x + j \sin \omega x. \quad (3.52)$$

In that case, we can simply write,

$$o(x) = h(x) * s(x) = Ae^{j(\omega x + \phi)}. \quad (3.53)$$

The *Fourier transform* is simply a tabulation of the magnitude and phase response at each frequency,

$$H(\omega) = \mathcal{F}\{h(x)\} = Ae^{j\phi}, \quad (3.54)$$

i.e., it is the response to a complex sinusoid of frequency ω passed through the filter $h(x)$. The Fourier transform pair is also often written as

$$h(x) \xleftrightarrow{\mathcal{F}} H(\omega). \quad (3.55)$$

Unfortunately, (3.54) does not give an actual *formula* for computing the Fourier transform. Instead, it gives a *recipe*, i.e., convolve the filter with a sinusoid, observe the magnitude and phase shift, repeat. Fortunately, closed form equations for the Fourier transform exist both in the continuous domain,

$$H(\omega) = \int_{-\infty}^{\infty} h(x)e^{-j\omega x} dx, \quad (3.56)$$

the bane of audiophiles, who insisted on equipment with no *harmonic distortion*. Now that digital audio has introduced pure distortion-free sound, some audiophiles are buying retro tube amplifiers or digital signal processors that simulate such distortions because of their “warmer sound”.

and in the discrete domain,

$$H(k) = \frac{1}{N} \sum_{x=0}^{N-1} h(x) e^{-j \frac{2\pi kx}{N}}, \quad (3.57)$$

where N is the length of the signal or region of analysis. These formulas apply both to filters, such as $h(x)$, and to signals or images, such as $s(x)$ or $g(x)$.

The discrete form of the Fourier transform (3.57) is known as the *Discrete Fourier Transform* (DFT). Note that while (3.57) can be evaluated for any value of k , it only makes sense for values in the range $k \in [-\frac{N}{2}, \frac{N}{2}]$. This is because larger values of k alias with lower frequencies and hence provide no additional information, as explained in the discussion on aliasing in Section 2.3.1.

At face value, the DFT takes $O(N^2)$ operations (multiply-adds) to evaluate. Fortunately, there exists a faster algorithm called the *Fast Fourier Transform* (FFT), which requires only $O(N \log_2 N)$ operations (Bracewell 1986; Oppenheim, Schafer, and Buck 1999). We do not explain the details of the algorithm here, except to say that it involves a series of $\log_2 N$ stages, where each stage performs small 2×2 transforms (matrix multiplications with known coefficients) followed by some semi-global permutations. (You will often see the term *butterfly* applied to these stages because of the pictorial shape of the signal processing graphs involved.) Implementations for the FFT can be found in most numerical and signal processing libraries.

The Fourier transform comes with a set of extremely useful properties relating original signals and their Fourier transforms, including superposition, shifting, reversal, convolution, correlation, multiplication, differentiation, domain scaling (stretching), and energy preservation (Parseval's Theorem). To make room for all of the new material in this second edition, I have removed all of these details, as well as a discussion of commonly used Fourier transform pairs. Interested readers should refer to (Szeliski 2010, Section 3.1, Tables 3.1–3.3) or standard textbooks on signal processing and Fourier transforms (Bracewell 1986; Glassner 1995; Oppenheim and Schafer 1996; Oppenheim, Schafer, and Buck 1999).

We can also compute the Fourier transforms for the small discrete kernels shown in Figure 3.14 (see Table 3.1). Notice how the moving average filters do not uniformly dampen higher frequencies and hence can lead to ringing artifacts. The binomial filter (Gomes and Velho 1997) used as the “Gaussian” in Burt and Adelson's (1983a) Laplacian pyramid (see Section 3.5), does a decent job of separating the high and low frequencies, but still leaves a fair amount of high-frequency detail, which can lead to aliasing after downsampling. The Sobel edge detector at first linearly accentuates frequencies, but then decays at higher frequencies, and hence has trouble detecting fine-scale edges, e.g., adjacent black and white

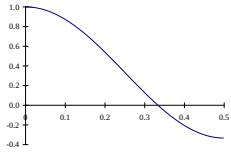
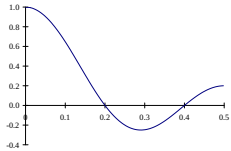
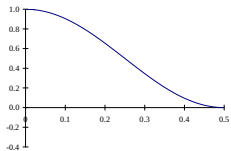
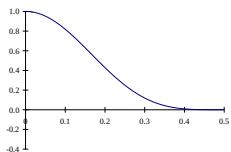
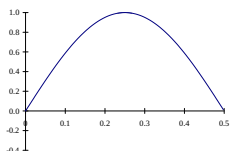
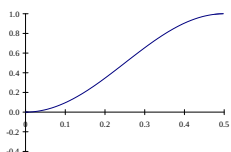
Name	Kernel	Transform	Plot
box-3	$\frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$	$\frac{1}{3}(1 + 2 \cos \omega)$	
box-5	$\frac{1}{5} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}$	$\frac{1}{5}(1 + 2 \cos \omega + 2 \cos 2\omega)$	
linear	$\frac{1}{4} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$	$\frac{1}{2}(1 + \cos \omega)$	
binomial	$\frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	$\frac{1}{4}(1 + \cos \omega)^2$	
Sobel	$\frac{1}{2} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$	$\sin \omega$	
corner	$\frac{1}{2} \begin{bmatrix} -1 & 2 & -1 \end{bmatrix}$	$\frac{1}{2}(1 - \cos \omega)$	

Table 3.1 *Fourier transforms of the separable kernels shown in Figure 3.14, obtained by evaluating $\sum_k h(k)e^{-jk\omega}$.*

columns. We look at additional examples of small kernel Fourier transforms in Section 3.5.2, where we study better kernels for prefiltering before decimation (size reduction).

3.4.1 Two-dimensional Fourier transforms

The formulas and insights we have developed for one-dimensional signals and their transforms translate directly to two-dimensional images. Here, instead of just specifying a horizontal or vertical frequency ω_x or ω_y , we can create an oriented sinusoid of frequency (ω_x, ω_y) ,

$$s(x, y) = \sin(\omega_x x + \omega_y y). \quad (3.58)$$

The corresponding two-dimensional Fourier transforms are then

$$H(\omega_x, \omega_y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(x, y) e^{-j(\omega_x x + \omega_y y)} dx dy, \quad (3.59)$$

and in the discrete domain,

$$H(k_x, k_y) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h(x, y) e^{-j2\pi(k_x x/M + k_y y/N)} \quad (3.60)$$

where M and N are the width and height of the image.

All of the Fourier transform properties from 1D carry over to two dimensions if we replace the scalar variables x , ω , x_0 and a , with their 2D vector counterparts $\mathbf{x} = (x, y)$, $\boldsymbol{\omega} = (\omega_x, \omega_y)$, $\mathbf{x}_0 = (x_0, y_0)$, and $\mathbf{a} = (a_x, a_y)$, and use vector inner products instead of multiplications.

Wiener filtering

While the Fourier transform is a useful tool for analyzing the frequency characteristics of a filter kernel or image, it can also be used to analyze the frequency spectrum of a whole *class* of images.

A simple model for images is to assume that they are random noise fields whose expected magnitude at each frequency is given by this *power spectrum* $P_s(\omega_x, \omega_y)$, i.e.,

$$\langle [S(\omega_x, \omega_y)]^2 \rangle = P_s(\omega_x, \omega_y), \quad (3.61)$$

where the angle brackets $\langle \cdot \rangle$ denote the expected (mean) value of a random variable.¹⁰ To generate such an image, we simply create a random Gaussian noise image $S(\omega_x, \omega_y)$ where each “pixel” is a zero-mean Gaussian of variance $P_s(\omega_x, \omega_y)$ and then take its inverse FFT.

¹⁰The notation $E[\cdot]$ is also commonly used.

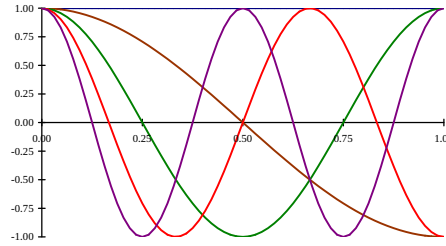


Figure 3.25 *Discrete cosine transform (DCT) basis functions: The first DC (i.e., constant) basis is the horizontal blue line, the second is the brown half-cycle waveform, etc. These bases are widely used in image and video compression standards such as JPEG.*

The observation that signal spectra capture a first-order description of spatial statistics is widely used in signal and image processing. In particular, assuming that an image is a sample from a correlated Gaussian random noise field combined with a statistical model of the measurement process yields an optimum restoration filter known as the *Wiener filter*.

The first edition of this book contains a derivation of the Wiener filter (Szeliski 2010, Section 3.4.3), but I’ve decided to remove this from the current edition, since it is almost never used in practice any more, having been replaced with better-performing non-linear filters.

Discrete cosine transform

The *discrete cosine transform* (DCT) is a variant of the Fourier transform particularly well-suited to compressing images in a block-wise fashion. The one-dimensional DCT is computed by taking the dot product of each N -wide block of pixels with a set of cosines of different frequencies,

$$F(k) = \sum_{i=0}^{N-1} \cos\left(\frac{\pi}{N}\left(i + \frac{1}{2}\right)k\right) f(i), \quad (3.62)$$

where k is the coefficient (frequency) index and the $1/2$ -pixel offset is used to make the basis coefficients symmetric (Wallace 1991). Some of the discrete cosine basis functions are shown in Figure 3.25. As you can see, the first basis function (the straight blue line) encodes the average DC value in the block of pixels, while the second encodes a slightly curvy version of the slope.

It turns out that the DCT is a good approximation to the optimal Karhunen–Loève decomposition of natural image statistics over small patches, which can be obtained by performing a principal component analysis (PCA) of images, as described in Section 5.2.3. The KL-

transform decorrelates the signal optimally (assuming the signal is described by its spectrum) and thus, theoretically, leads to optimal compression.

The two-dimensional version of the DCT is defined similarly,

$$F(k, l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \cos\left(\frac{\pi}{N}\left(i + \frac{1}{2}\right)k\right) \cos\left(\frac{\pi}{N}\left(j + \frac{1}{2}\right)l\right) f(i, j). \quad (3.63)$$

Like the 2D Fast Fourier Transform, the 2D DCT can be implemented separably, i.e., first computing the DCT of each line in the block and then computing the DCT of each resulting column. Like the FFT, each of the DCTs can also be computed in $O(N \log N)$ time.

As we mentioned in Section 2.3.3, the DCT is widely used in today's image and video compression algorithms, although alternatives such as wavelet transforms (Simoncelli and Adelson 1990b; Taubman and Marcellin 2002), discussed in Section 3.5.4, and overlapped variants of the DCT (Malvar 1990, 1998, 2000), are used in the JPEG2000 and JPEG XR standards. These newer algorithms suffer less from the *blocking artifacts* (visible edge-aligned discontinuities) that result from the pixels in each block (typically 8×8) being transformed and quantized independently. See Exercise 4.3 for ideas on how to remove blocking artifacts from compressed JPEG images.

3.4.2 Application: Sharpening, blur, and noise removal

Another common application of image processing is the enhancement of images through the use of sharpening and noise removal operations, which require some kind of neighborhood processing. Traditionally, these kinds of operations were performed using linear filtering (see Sections 3.2 and Section 3.4.1). Today, it is more common to use non-linear filters (Section 3.3.1), such as the weighted median or bilateral filter (3.34–3.37), anisotropic diffusion (3.39–3.40), or non-local means (Buades, Coll, and Morel 2008). Variational methods (Section 4.2), especially those using non-quadratic (robust) norms such as the L_1 norm (which is called *total variation*), are also often used. Most recently, deep neural networks have taken over the denoising community (Section 10.3). Figure 3.19 shows some examples of linear and non-linear filters being used to remove noise.

When measuring the effectiveness of image denoising algorithms, it is common to report the results as a *peak signal-to-noise ratio* (PSNR) measurement (2.120), where $I(\mathbf{x})$ is the original (noise-free) image and $\hat{I}(\mathbf{x})$ is the image after denoising; this is for the case where the noisy image has been synthetically generated, so that the clean image is known. A better way to measure the quality is to use a perceptually based similarity metric, such as the structural similarity (SSIM) index (Wang, Bovik *et al.* 2004; Wang, Bovik, and Simoncelli 2005) or FLIP image difference evaluator (Andersson, Nilsson *et al.* 2020). More recently,

people have started measuring similarity using neural “perceptual” similarity metrics (Johnson, Alahi, and Fei-Fei 2016; Dosovitskiy and Brox 2016; Zhang, Isola *et al.* 2018; Tariq, Tursun *et al.* 2020; Czolbe, Krause *et al.* 2020), which, unlike L_2 (PSNR) or L_1 metrics, which encourage smooth or flat average results, prefer images with similar amounts of texture (Cho, Joshi *et al.* 2012). When the clean image is not available, it is also possible to assess the quality of an image using *no-reference image quality assessment* (Mittal, Moorthy, and Bovik 2012; Talebi and Milanfar 2018).

Exercises 3.12, 3.21, and 3.28 have you implement some of these operations and compare their effectiveness. More sophisticated techniques for blur removal and the related task of super-resolution are discussed in Section 10.3.

3.5 Pyramids and wavelets

So far in this chapter, all of the image transformations we have studied produce output images of the same size as the inputs. Often, however, we may wish to change the resolution of an image before proceeding further. For example, we may need to interpolate a small image to make its resolution match that of the output printer or computer screen. Alternatively, we may want to reduce the size of an image to speed up the execution of an algorithm or to save on storage space or transmission time.

Sometimes, we do not even know what the appropriate resolution for the image should be. Consider, for example, the task of finding a face in an image (Section 6.3.1). Since we do not know the scale at which the face will appear, we need to generate a whole *pyramid* of differently sized images and scan each one for possible faces. (Biological visual systems also operate on a hierarchy of scales (Marr 1982).) Such a pyramid can also be very helpful in accelerating the search for an object by first finding a smaller instance of that object at a coarser level of the pyramid and then looking for the full resolution object only in the vicinity of coarse-level detections (Section 9.1.1). Finally, image pyramids are extremely useful for performing multi-scale editing operations such as blending images while maintaining details.

In this section, we first discuss good filters for changing image resolution, i.e., upsampling (*interpolation*, Section 3.5.1) and downsampling (*decimation*, Section 3.5.2). We then present the concept of multi-resolution pyramids, which can be used to create a complete hierarchy of differently sized images and to enable a variety of applications (Section 3.5.3). A closely related concept is that of *wavelets*, which are a special kind of pyramid with higher frequency selectivity and other useful properties (Section 3.5.4). Finally, we present a useful application of pyramids, namely the blending of different images in a way that hides the seams between the image boundaries (Section 3.5.5).

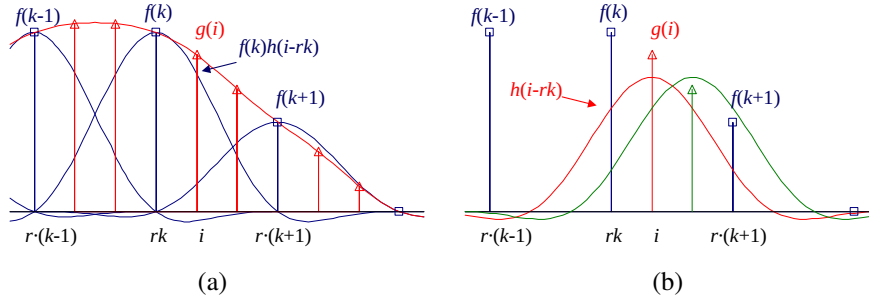


Figure 3.26 Signal interpolation, $g(i) = \sum_k f(k)h(i - rk)$: (a) weighted summation of input values; (b) polyphase filter interpretation.

3.5.1 Interpolation

In order to *interpolate* (or *upsample*) an image to a higher resolution, we need to select some interpolation kernel with which to convolve the image,

$$g(i, j) = \sum_{k, l} f(k, l)h(i - rk, j - rl). \quad (3.64)$$

This formula is related to the discrete convolution formula (3.14), except that we replace k and l in $h()$ with rk and rl , where r is the upsampling rate. Figure 3.26a shows how to think of this process as the superposition of sample weighted interpolation kernels, one centered at each input sample k . An alternative mental model is shown in Figure 3.26b, where the kernel is centered at the output pixel value i (the two forms are equivalent). The latter form is sometimes called the *polyphase filter* form, since the kernel values $h(i)$ can be stored as r separate kernels, each of which is selected for convolution with the input samples depending on the *phase* of i relative to the upsampled grid.

What kinds of kernel make good interpolators? The answer depends on the application and the computation time involved. Any of the smoothing kernels shown in Table 3.1 can be used after appropriate re-scaling.¹¹ The *linear* interpolator (corresponding to the tent kernel) produces interpolating piecewise linear curves, which result in unappealing *creases* when applied to images (Figure 3.27a). The cubic B-spline, whose discrete $1/2$ -pixel sampling appears as the *binomial kernel* in Table 3.1, is an *approximating* kernel (the interpolated image does not pass through the input data points) that produces soft images with reduced high-frequency detail. The equation for the cubic B-spline is easiest to derive by convolving the tent function (linear B-spline) with itself.

¹¹The smoothing kernels in Table 3.1 have a unit area. To turn them into interpolating kernels, we simply scale

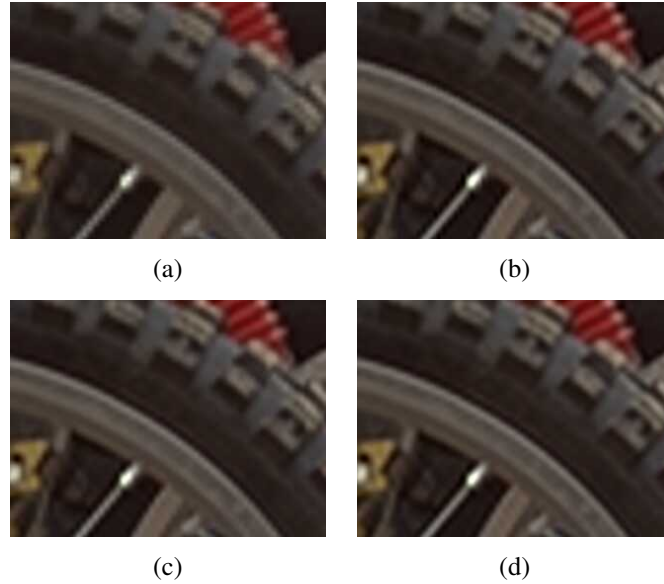


Figure 3.27 Two-dimensional image interpolation: (a) bilinear; (b) bicubic ($a = -1$); (c) bicubic ($a = -0.5$); (d) windowed sinc (nine taps).

While most graphics cards use the bilinear kernel (optionally combined with a MIP-map—see Section 3.5.3), most photo editing packages use *bicubic* interpolation. The cubic interpolant is a C^1 (derivative-continuous) piecewise-cubic *spline* (the term “spline” is synonymous with “piecewise-polynomial”)¹² whose equation is

$$h(x) = \begin{cases} 1 - (a + 3)x^2 + (a + 2)|x|^3 & \text{if } |x| < 1 \\ a(|x| - 1)(|x| - 2)^2 & \text{if } 1 \leq |x| < 2 \\ 0 & \text{otherwise,} \end{cases} \quad (3.65)$$

where a specifies the derivative at $x = 1$ (Parker, Kenyon, and Troxel 1983). The value of a is often set to -1 , since this best matches the frequency characteristics of a sinc function (Figure 3.28). It also introduces a small amount of sharpening, which can be visually appealing. Unfortunately, this choice does not linearly interpolate straight lines (intensity ramps), so some visible ringing may occur. A better choice for large amounts of interpolation is probably $a = -0.5$, which produces a *quadratic reproducing* spline; it interpolates linear and quadratic functions exactly (Wolberg 1990, Section 5.4.3). Figure 3.28 shows the $a = -1$

them up by the interpolation rate r .

¹²The term “spline” comes from the draughtsman’s workshop, where it was the name of a flexible piece of wood or metal used to draw smooth curves.

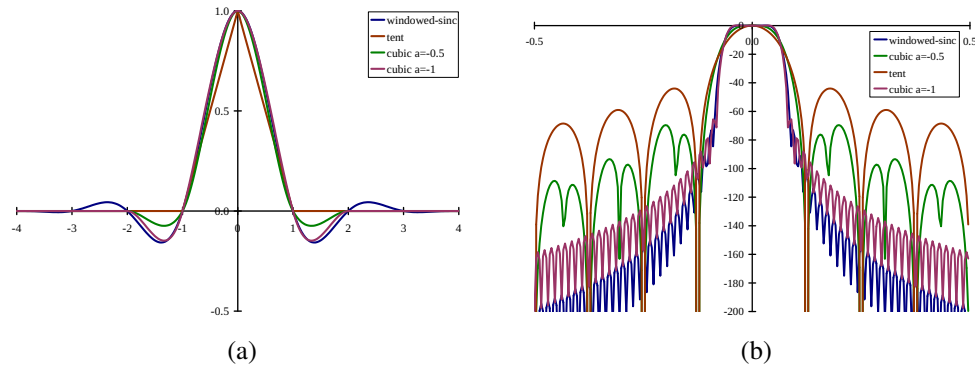


Figure 3.28 (a) Some windowed sinc functions and (b) their log Fourier transforms: raised-cosine windowed sinc in blue, cubic interpolators ($a = -1$ and $a = -0.5$) in green and purple, and tent function in brown. They are often used to perform high-accuracy low-pass filtering operations.

and $a = -0.5$ cubic interpolating kernel along with their Fourier transforms; Figure 3.27b and c shows them being applied to two-dimensional interpolation.

Splines have long been used for function and data value interpolation because of the ability to precisely specify derivatives at control points and efficient *incremental* algorithms for their evaluation (Bartels, Beatty, and Barsky 1987; Farin 1992, 2002). Splines are widely used in geometric modeling and computer-aided design (CAD) applications, although they have started being displaced by subdivision surfaces (Zorin, Schröder, and Sweldens 1996; Peters and Reif 2008). In computer vision, splines are often used for elastic image deformations (Section 3.6.2), scattered data interpolation (Section 4.1), motion estimation (Section 9.2.2), and surface interpolation (Section 13.3). In fact, it is possible to carry out most image processing operations by representing images as splines and manipulating them in a multi-resolution framework (Unser 1999; Nehab and Hoppe 2014).

The highest quality interpolator is generally believed to be the windowed sinc function because it both preserves details in the lower resolution image and avoids aliasing. (It is also possible to construct a C^1 piecewise-cubic approximation to the windowed sinc by matching its derivatives at zero crossing (Szeliski and Ito 1986).) However, some people object to the excessive *ringing* that can be introduced by the windowed sinc and to the repetitive nature of the ringing frequencies (see Figure 3.27d). For this reason, some photographers prefer to repeatedly interpolate images by a small fractional amount (this tends to decorrelate the original pixel grid with the final image). Additional possibilities include using the bilateral filter as an interpolator (Kopf, Cohen *et al.* 2007), using global optimization (Section 3.6) or

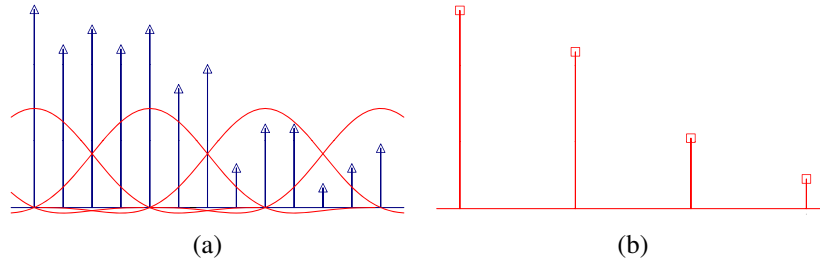


Figure 3.29 *Signal decimation: (a) the original samples are (b) convolved with a low-pass filter before being downsampled.*

hallucinating details (Section 10.3).

3.5.2 Decimation

While interpolation can be used to increase the resolution of an image, decimation (downsampling) is required to reduce the resolution.¹³ To perform decimation, we first (conceptually) convolve the image with a low-pass filter (to avoid aliasing) and then keep every r th sample. In practice, we usually only evaluate the convolution at every r th sample,

$$g(i, j) = \sum_{k, l} f(k, l) h(ri - k, rj - l), \quad (3.66)$$

as shown in Figure 3.29. Note that the smoothing kernel $h(k, l)$, in this case, is often a stretched and re-scaled version of an interpolation kernel. Alternatively, we can write

$$g(i, j) = \frac{1}{r} \sum_{k, l} f(k, l) h(i - k/r, j - l/r) \quad (3.67)$$

and keep the same kernel $h(k, l)$ for both interpolation and decimation.

One commonly used ($r = 2$) decimation filter is the *binomial* filter introduced by Burt and Adelson (1983a). As shown in Table 3.1, this kernel does a decent job of separating the high and low frequencies, but still leaves a fair amount of high-frequency detail, which can lead to aliasing after downsampling. However, for applications such as image blending (discussed later in this section), this aliasing is of little concern.

If, however, the downsampled images will be displayed directly to the user or, perhaps, blended with other resolutions (as in MIP-mapping, Section 3.5.3), a higher-quality filter is

¹³The term “decimation” has a gruesome etymology relating to the practice of killing every tenth soldier in a Roman unit guilty of cowardice. It is generally used in signal processing to mean any downsampling or rate reduction operation.

desired. For high downsampling rates, the windowed sinc prefilter is a good choice (Figure 3.28). However, for small downsampling rates, e.g., $r = 2$, more careful filter design is required.

Table 3.2 shows a number of commonly used $r = 2$ downsampling filters, while Figure 3.30 shows their corresponding frequency responses. These filters include:

- the linear $[1, 2, 1]$ filter gives a relatively poor response;
- the binomial $[1, 4, 6, 4, 1]$ filter cuts off a lot of frequencies but is useful for computer vision analysis pyramids;
- the cubic filters from (3.65); the $a = -1$ filter has a sharper fall-off than the $a = -0.5$ filter (Figure 3.30);
- a cosine-windowed sinc function;
- the QMF-9 filter of Simoncelli and Adelson (1990b) is used for wavelet denoising and aliases a fair amount (note that the original filter coefficients are normalized to $\sqrt{2}$ gain so they can be “self-inverting”);
- the 9/7 analysis filter from JPEG 2000 (Taubman and Marcellin 2002).

Please see the original papers for the full-precision values of some of these coefficients.

3.5.3 Multi-resolution representations

Now that we have described interpolation and decimation algorithms, we can build a complete image pyramid (Figure 3.31). As we mentioned before, pyramids can be used to accelerate

$ n $	Linear	Binomial	Cubic $a = -1$	Cubic $a = -0.5$	Windowed sinc	QMF-9	JPEG 2000
0	0.50	0.3750	0.5000	0.50000	0.4939	0.5638	0.6029
1	0.25	0.2500	0.3125	0.28125	0.2684	0.2932	0.2669
2		0.0625	0.0000	0.00000	0.0000	-0.0519	-0.0782
3			-0.0625	-0.03125	-0.0153	-0.0431	-0.0169
4					0.0000	0.0198	0.0267

Table 3.2 Filter coefficients for $2 \times$ decimation. These filters are of odd length, are symmetric, and are normalized to have unit DC gain (sum up to 1). See Figure 3.30 for their associated frequency responses.

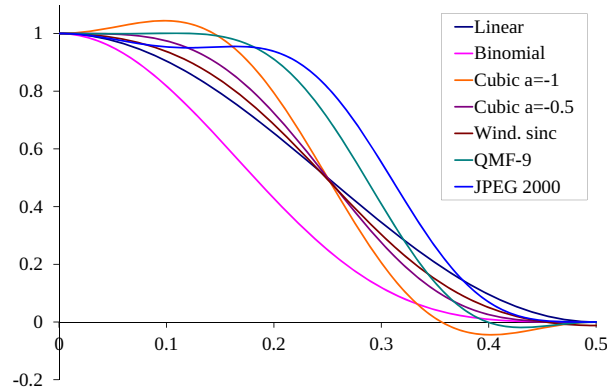


Figure 3.30 Frequency response for some $2 \times$ decimation filters. The cubic $a = -1$ filter has the sharpest fall-off but also a bit of ringing; the wavelet analysis filters (QMF-9 and JPEG 2000), while useful for compression, have more aliasing.

coarse-to-fine search algorithms, to look for objects or patterns at different scales, and to perform multi-resolution blending operations. They are also widely used in computer graphics hardware and software to perform fractional-level decimation using the MIP-map, which we discuss in Section 3.6.

The best known (and probably most widely used) pyramid in computer vision is Burt and Adelson's (1983a) Laplacian pyramid. To construct the pyramid, we first blur and subsample the original image by a factor of two and store this in the next level of the pyramid (Figures 3.31 and 3.32). Because adjacent levels in the pyramid are related by a sampling rate $r = 2$, this kind of pyramid is known as an *octave pyramid*. Burt and Adelson originally proposed a five-tap kernel of the form

$$\begin{bmatrix} c & b & a & b & c \end{bmatrix}, \quad (3.68)$$

with $b = 1/4$ and $c = 1/4 - a/2$. In practice, they and everyone else uses $a = 3/8$, which results in the familiar binomial kernel,

$$\frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix}, \quad (3.69)$$

which is particularly easy to implement using shifts and adds. (This was important in the days when multipliers were expensive.) The reason they call their resulting pyramid a *Gaussian* pyramid is that repeated convolutions of the binomial kernel converge to a Gaussian.¹⁴

¹⁴Then again, this is true for any smoothing kernel (Wells 1986).

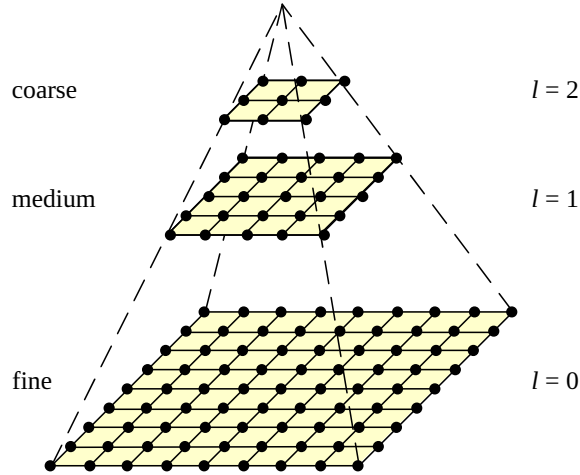


Figure 3.31 A traditional image pyramid: each level has half the resolution (width and height), and hence a quarter of the pixels, of its parent level.

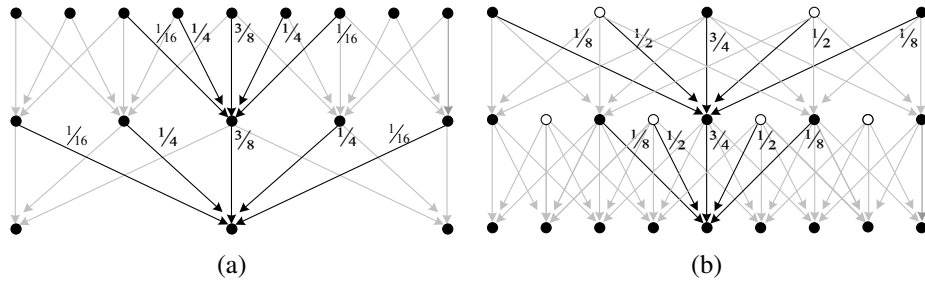


Figure 3.32 The Gaussian pyramid shown as a signal processing diagram: The (a) analysis and (b) re-synthesis stages are shown as using similar computations. The white circles indicate zero values inserted by the $\uparrow 2$ upsampling operation. Notice how the reconstruction filter coefficients are twice the analysis coefficients. The computation is shown as flowing down the page, regardless of whether we are going from coarse to fine or vice versa.

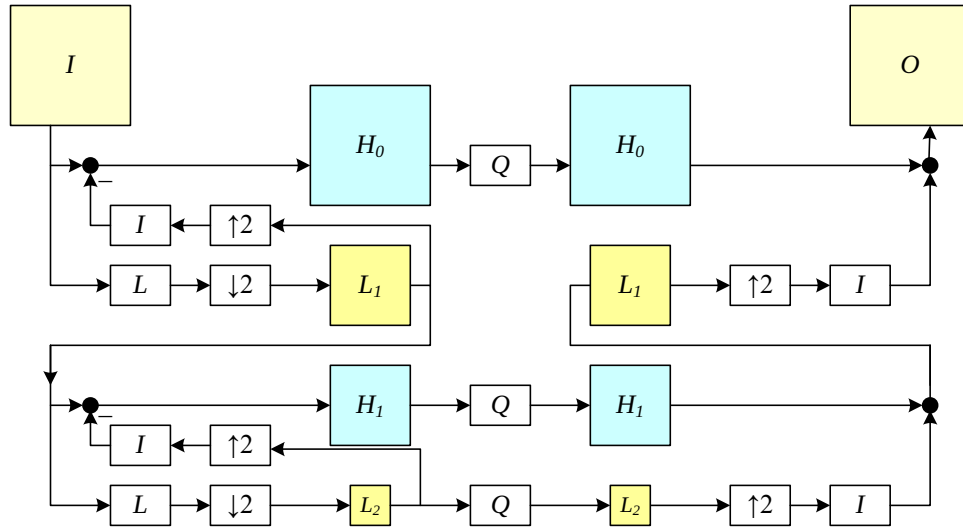


Figure 3.33 The Laplacian pyramid. The yellow images form the Gaussian pyramid, which is obtained by successively low-pass filtering and downsampling the input image. The blue images, together with the smallest low-pass image, which is needed for reconstruction, form the Laplacian pyramid. Each band-pass (blue) image is computed by upsampling and interpolating the lower-resolution Gaussian pyramid image, resulting in a blurred version of that level's low-pass image, which is subtracted from the low-pass to yield the blue band-pass image. During reconstruction, the interpolated images and the (optionally filtered) high-pass images are added back together starting with the coarsest level. The Q box indicates quantization or some other pyramid processing, e.g., noise removal by coring (setting small wavelet values to 0).

To compute the *Laplacian* pyramid, Burt and Adelson first interpolate a lower resolution image to obtain a *reconstructed* low-pass version of the original image (Figure 3.33). They then subtract this low-pass version from the original to yield the band-pass “Laplacian” image, which can be stored away for further processing. The resulting pyramid has *perfect reconstruction*, i.e., the Laplacian images plus the base-level Gaussian (L_2 in Figure 3.33) are sufficient to exactly reconstruct the original image. Figure 3.32 shows the same computation in one dimension as a signal processing diagram, which completely captures the computations being performed during the analysis and re-synthesis stages.

Burt and Adelson also describe a variant of the Laplacian pyramid, where the low-pass image is taken from the original blurred image rather than the reconstructed pyramid (piping

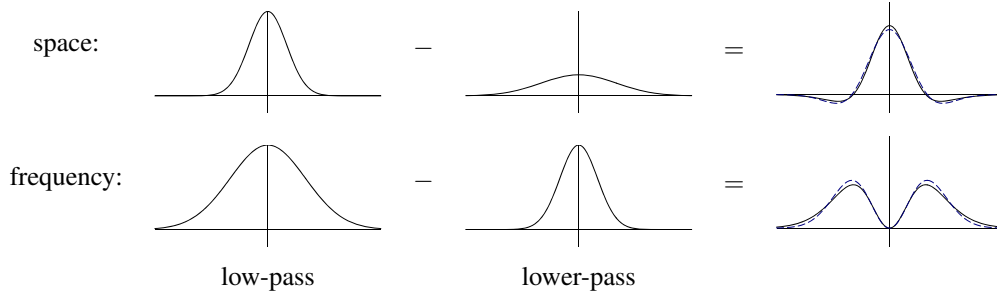


Figure 3.34 The difference of two low-pass filters results in a band-pass filter. The dashed blue lines show the close fit to a half-octave Laplacian of Gaussian.

the output of the L box directly to the subtraction in Figure 3.33). This variant has less aliasing, since it avoids one downsampling and upsampling round-trip, but it is not self-inverting, since the Laplacian images are no longer adequate to reproduce the original image.

As with the Gaussian pyramid, the term Laplacian is a bit of a misnomer, since their band-pass images are really differences of (approximate) Gaussians, or DoGs,

$$\text{DoG}\{I; \sigma_1, \sigma_2\} = G_{\sigma_1} * I - G_{\sigma_2} * I = (G_{\sigma_1} - G_{\sigma_2}) * I. \quad (3.70)$$

A Laplacian of Gaussian (which we saw in (3.26)) is actually its second derivative,

$$\text{LoG}\{I; \sigma\} = \nabla^2(G_\sigma * I) = (\nabla^2 G_\sigma) * I, \quad (3.71)$$

where

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \quad (3.72)$$

is the Laplacian (operator) of a function. Figure 3.34 shows how the Differences of Gaussian and Laplacians of Gaussian look in both space and frequency.

Laplacians of Gaussian have elegant mathematical properties, which have been widely studied in the *scale-space* community (Witkin 1983; Witkin, Terzopoulos, and Kass 1986; Lindeberg 1990; Nielsen, Florack, and Deriche 1997) and can be used for a variety of applications including edge detection (Marr and Hildreth 1980; Perona and Malik 1990b), stereo matching (Witkin, Terzopoulos, and Kass 1987), and image enhancement (Nielsen, Florack, and Deriche 1997).

One particularly useful application of the Laplacian pyramid is in the manipulation of local contrast as well as the tone mapping of high dynamic range images (Section 10.2.1). Paris, Hasinoff, and Kautz (2011) present a technique they call *local Laplacian filters*, which uses local range clipping in the construction of a modified Laplacian pyramid, as well as

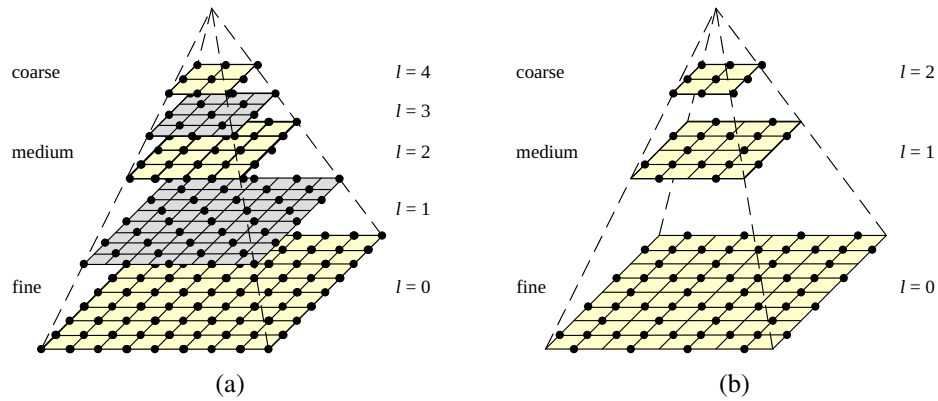


Figure 3.35 Multiresolution pyramids: (a) pyramid with half-octave (quincunx) sampling (odd levels are colored gray for clarity). (b) wavelet pyramid—each wavelet level stores $3/4$ of the original pixels (usually the horizontal, vertical, and mixed gradients), so that the total number of wavelet coefficients and original pixels is the same.

different accentuation and attenuation curves for small and large details, to implement edge-preserving filtering and tone mapping. Aubry, Paris *et al.* (2014) discuss how to accelerate this processing for monotone (single channel) images and also show style transfer applications.

A less widely used variant is *half-octave pyramids*, shown in Figure 3.35a. These were first introduced to the vision community by Crowley and Stern (1984), who call them *Difference of Low-Pass* (DOLP) transforms. Because of the small scale change between adjacent levels, the authors claim that coarse-to-fine algorithms perform better. In the image-processing community, half-octave pyramids combined with checkerboard sampling grids are known as *quincunx* sampling (Feilner, Van De Ville, and Unser 2005). In detecting multi-scale features (Section 7.1.1), it is often common to use half-octave or even quarter-octave pyramids (Lowe 2004; Triggs 2004). However, in this case, the subsampling only occurs at every octave level, i.e., the image is repeatedly blurred with wider Gaussians until a full octave of resolution change has been achieved (Figure 7.11).

3.5.4 Wavelets

While pyramids are used extensively in computer vision applications, some people use *wavelet* decompositions as an alternative. Wavelets are filters that localize a signal in both space and frequency (like the Gabor filter) and are defined over a hierarchy of scales. Wavelets provide a smooth way to decompose a signal into frequency components without blocking and are closely related to pyramids.

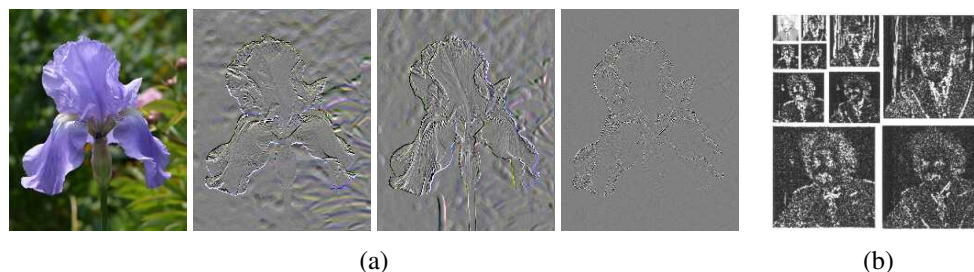


Figure 3.36 A wavelet decomposition of an image: (a) single level decomposition with horizontal, vertical, and diagonal detail wavelets constructed using PyWavelet code (<https://pywavelets.readthedocs.io>); (b) coefficient magnitudes of a multi-level decomposition, with the high-high components in the lower right corner and the base in the upper left (Buccigrossi and Simoncelli 1999) © 1999 IEEE. Notice how the low-high and high-low components accentuate horizontal and vertical edges and gradients, while the high-high components store the less frequent mixed derivatives.

Wavelets were originally developed in the applied math and signal processing communities and were introduced to the computer vision community by Mallat (1989). Strang (1989), Simoncelli and Adelson (1990b), Rioul and Vetterli (1991), Chui (1992), and Meyer (1993) all provide nice introductions to the subject along with historical reviews, while Chui (1992) provides a more comprehensive review and survey of applications. Sweldens (1997) describes the *lifting* approach to wavelets that we discuss shortly.

Wavelets are widely used in the computer graphics community to perform multi-resolution geometric processing (Stollnitz, DeRose, and Salesin 1996) and have also been used in computer vision for similar applications (Szeliski 1990b; Pentland 1994; Gortler and Cohen 1995; Yaou and Chang 1994; Lai and Vemuri 1997; Szeliski 2006b; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013), as well as for multi-scale oriented filtering (Simoncelli, Freeman *et al.* 1992) and denoising (Portilla, Strela *et al.* 2003).

As both image pyramids and wavelets decompose an image into multi-resolution descriptions that are localized in both space and frequency, how do they differ? The usual answer is that traditional pyramids are *overcomplete*, i.e., they use more pixels than the original image to represent the decomposition, whereas wavelets provide a *tight frame*, i.e., they keep the size of the decomposition the same as the image (Figure 3.35b). However, some wavelet families *are*, in fact, overcomplete in order to provide better shiftability or steering in orientation (Simoncelli, Freeman *et al.* 1992). A better distinction, therefore, might be that wavelets are more orientation selective than regular band-pass pyramids.

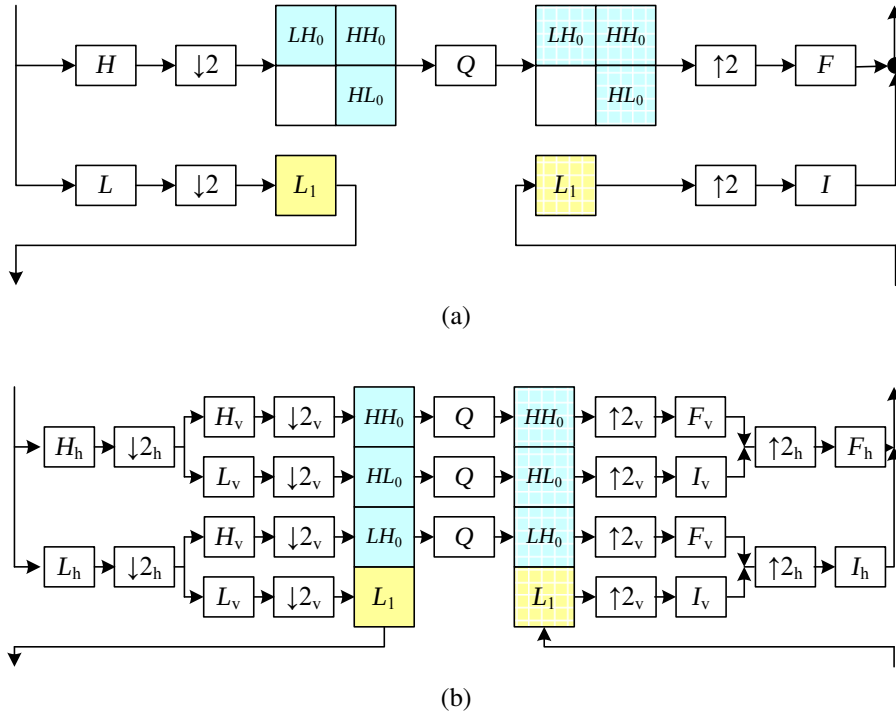


Figure 3.37 Two-dimensional wavelet decomposition: (a) high-level diagram showing the low-pass and high-pass transforms as single boxes; (b) separable implementation, which involves first performing the wavelet transform horizontally and then vertically. The I and F boxes are the interpolation and filtering boxes required to re-synthesize the image from its wavelet components.

How are two-dimensional wavelets constructed? Figure 3.37a shows a high-level diagram of one stage of the (recursive) coarse-to-fine construction (analysis) pipeline alongside the complementary re-construction (synthesis) stage. In this diagram, the high-pass filter followed by decimation keeps $3/4$ of the original pixels, while $1/4$ of the low-frequency coefficients are passed on to the next stage for further analysis. In practice, the filtering is usually broken down into two separable sub-stages, as shown in Figure 3.37b. The resulting three wavelet images are sometimes called the high-high (HH), high-low (HL), and low-high (LH) images. The high-low and low-high images accentuate the horizontal and vertical edges and gradients, while the high-high image contains the less frequently occurring mixed derivatives (Figure 3.36).

How are the high-pass H and low-pass L filters shown in Figure 3.37b chosen and how

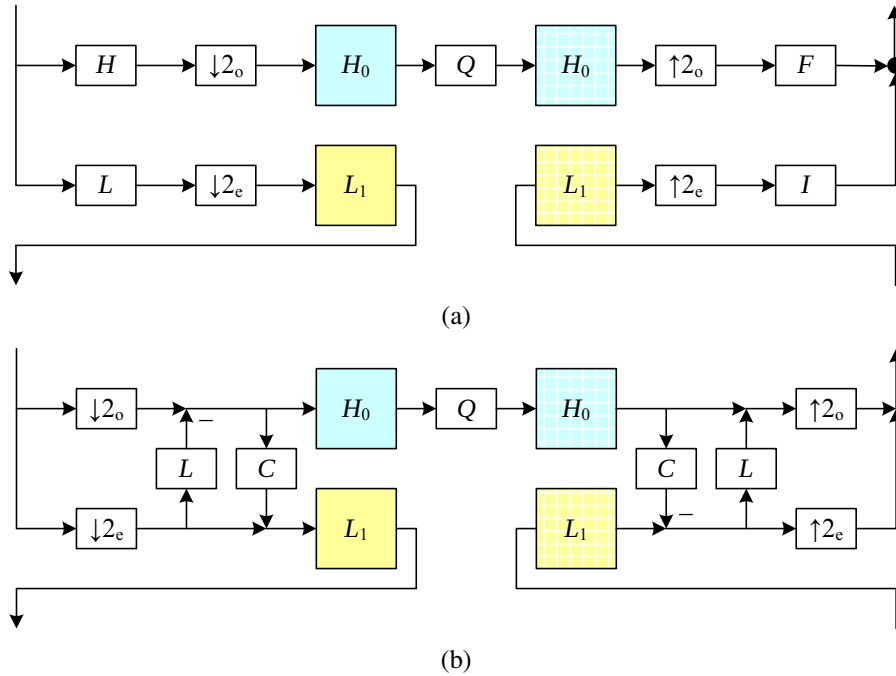


Figure 3.38 One-dimensional wavelet transform: (a) usual high-pass + low-pass filters followed by odd ($\downarrow 2_o$) and even ($\downarrow 2_e$) downsampling; (b) lifted version, which first selects the odd and even subsequences and then applies a low-pass prediction stage L and a high-pass correction stage C in an easily reversible manner.

can the corresponding reconstruction filters I and F be computed? Can filters be designed that all have finite impulse responses? This topic has been the main subject of study in the wavelet community for over two decades. The answer depends largely on the intended application, e.g., whether the wavelets are being used for compression, image analysis (feature finding), or denoising. Simoncelli and Adelson (1990b) show (in Table 4.1) some good odd-length quadrature mirror filter (QMF) coefficients that seem to work well in practice.

Since the design of wavelet filters is such a tricky art, is there perhaps a better way? Indeed, a simpler procedure is to split the signal into its even and odd components and then perform trivially reversible filtering operations on each sequence to produce what are called *lifted wavelets* (Figures 3.38 and 3.39). Sweldens (1996) gives a wonderfully understandable introduction to the *lifting scheme* for *second-generation wavelets*, followed by a comprehensive review (Sweldens 1997).

As Figure 3.38 demonstrates, rather than first filtering the whole input sequence (image)

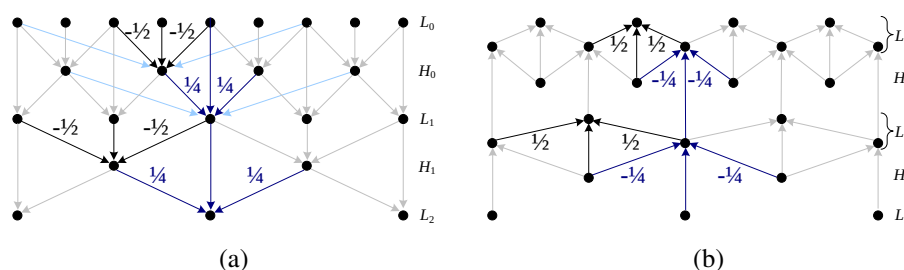


Figure 3.39 *Lifted transform shown as a signal processing diagram: (a) The analysis stage first predicts the odd value from its even neighbors, stores the difference wavelet, and then compensates the coarser even value by adding in a fraction of the wavelet. (b) The synthesis stage simply reverses the flow of computation and the signs of some of the filters and operations. The light blue lines show what happens if we use four taps for the prediction and correction instead of just two.*

with high-pass and low-pass filters and then keeping the odd and even sub-sequences, the lifting scheme first splits the sequence into its even and odd sub-components. Filtering the even sequence with a low-pass filter L and subtracting the result from the odd sequence is trivially reversible: simply perform the same filtering and then add the result back in. Furthermore, this operation can be performed in place, resulting in significant space savings. The same applies to filtering the difference signal with the correction filter C , which is used to ensure that the even sequence is low-pass. A series of such *lifting* steps can be used to create more complex filter responses with low computational cost and guaranteed reversibility.

This process can be more easily understood by considering the signal processing diagram in Figure 3.39. During analysis, the average of the even values is subtracted from the odd value to obtain a high-pass wavelet coefficient. However, the even samples still contain an aliased sample of the low-frequency signal. To compensate for this, a small amount of the high-pass wavelet is added back to the even sequence so that it is properly low-pass filtered. (It is easy to show that the effective low-pass filter is $[-1/8, 1/4, 3/4, 1/4, -1/8]$, which is indeed a low-pass filter.) During synthesis, the same operations are reversed with a judicious change in sign.

Of course, we need not restrict ourselves to two-tap filters. Figure 3.39 shows as light blue arrows additional filter coefficients that could optionally be added to the lifting scheme without affecting its reversibility. In fact, the low-pass and high-pass filtering operations can be interchanged, e.g., we could use a five-tap cubic low-pass filter on the odd sequence (plus center value) first, followed by a four-tap cubic low-pass predictor to estimate the wavelet,

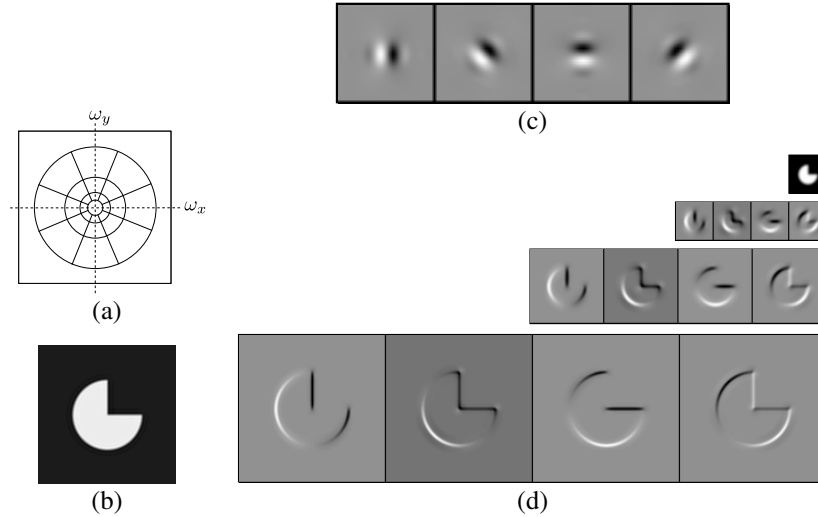


Figure 3.40 *Steerable shiftable multiscale transforms (Simoncelli, Freeman et al. 1992) © 1992 IEEE: (a) radial multi-scale frequency domain decomposition; (b) original image; (c) a set of four steerable filters; (d) the radial multi-scale wavelet decomposition.*

although I have not seen this scheme written down.

Lifted wavelets are called *second-generation wavelets* because they can easily adapt to non-regular sampling topologies, e.g., those that arise in computer graphics applications such as multi-resolution surface manipulation (Schröder and Sweldens 1995). It also turns out that lifted *weighted wavelets*, i.e., wavelets whose coefficients adapt to the underlying problem being solved (Fattal 2009), can be extremely effective for low-level image manipulation tasks and also for preconditioning the kinds of sparse linear systems that arise in the optimization-based approaches to vision algorithms that we discuss in Chapter 4 (Szeliski 2006b; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013).

An alternative to the widely used “separable” approach to wavelet construction, which decomposes each level into horizontal, vertical, and “cross” sub-bands, is to use a representation that is more rotationally symmetric and orientationally selective and also avoids the aliasing inherent in sampling signals below their Nyquist frequency.¹⁵ Simoncelli, Freeman *et al.* (1992) introduce such a representation, which they call a *pyramidal radial frequency implementation of shiftable multi-scale transforms* or, more succinctly, *steerable pyramids*. Their representation is not only overcomplete (which eliminates the aliasing problem) but is

¹⁵Such aliasing can often be seen as the signal content moving between bands as the original signal is slowly shifted.

also orientationally selective and has identical analysis and synthesis basis functions, i.e., it is *self-inverting*, just like “regular” wavelets. As a result, this makes steerable pyramids a much more useful basis for the structural analysis and matching tasks commonly used in computer vision.

Figure 3.40a shows how such a decomposition looks in frequency space. Instead of recursively dividing the frequency domain into 2×2 squares, which results in checkerboard high frequencies, radial arcs are used instead. Figure 3.40d illustrates the resulting pyramid sub-bands. Even though the representation is *overcomplete*, i.e., there are more wavelet coefficients than input pixels, the additional frequency and orientation selectivity makes this representation preferable for tasks such as texture analysis and synthesis (Portilla and Simoncelli 2000) and image denoising (Portilla, Strela *et al.* 2003; Lyu and Simoncelli 2009).

3.5.5 Application: Image blending

One of the most engaging and fun applications of the Laplacian pyramid presented in Section 3.5.3 is the creation of blended composite images, as shown in Figure 3.41 (Burt and Adelson 1983b). While splicing the apple and orange images together along the midline produces a noticeable cut, *splining* them together (as Burt and Adelson (1983b) called their procedure) creates a beautiful illusion of a truly hybrid fruit. The key to their approach is that the low-frequency color variations between the red apple and the orange are smoothly blended, while the higher-frequency textures on each fruit are blended more quickly to avoid “ghosting” effects when two textures are overlaid.

To create the blended image, each source image is first decomposed into its own Laplacian pyramid (Figure 3.42, left and middle columns). Each band is then multiplied by a smooth weighting function whose extent is proportional to the pyramid level. The simplest and most general way to create these weights is to take a binary mask image (Figure 3.41g) and to construct a *Gaussian* pyramid from this mask. Each Laplacian pyramid image is then multiplied by its corresponding Gaussian mask and the sum of these two weighted pyramids is then used to construct the final image (Figure 3.42, right column).

Figure 3.41e–h shows that this process can be applied to arbitrary mask images with surprising results. It is also straightforward to extend the pyramid blend to an arbitrary number of images whose pixel provenance is indicated by an integer-valued label image (see Exercise 3.18). This is particularly useful in image stitching and compositing applications, where the exposures may vary between different images, as described in Section 8.4.4, where we also present more recent variants such as Poisson and gradient-domain blending (Pérez, Gangnet, and Blake 2003; Levin, Zomet *et al.* 2004).

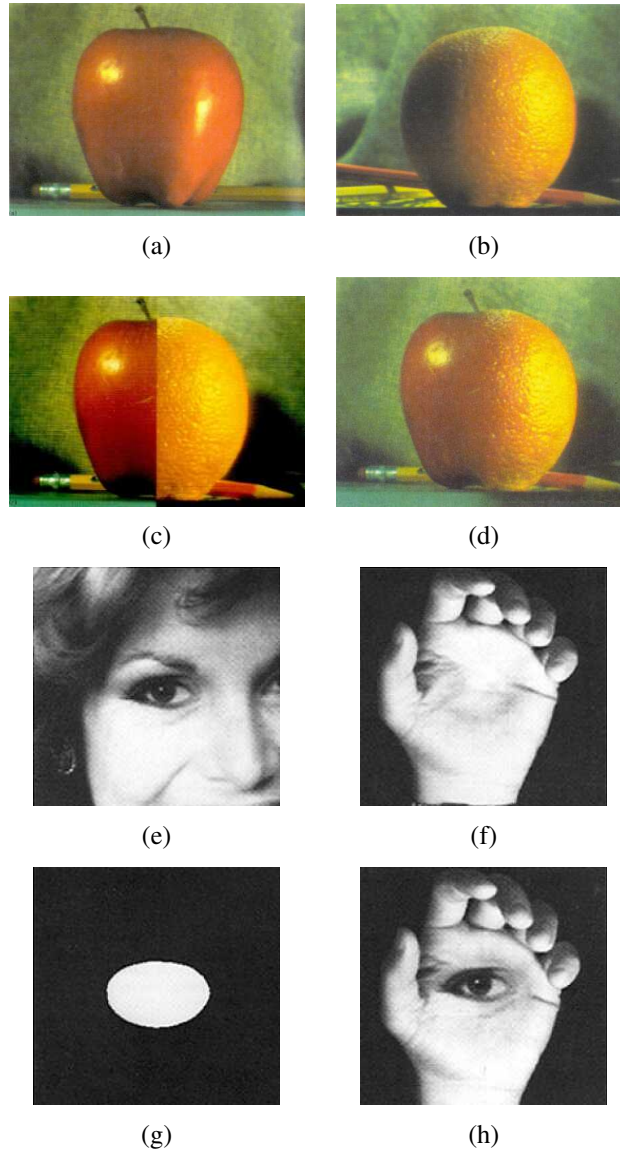


Figure 3.41 *Laplacian pyramid blending (Burt and Adelson 1983b) © 1983 ACM: (a) original image of apple, (b) original image of orange, (c) regular splice, (d) pyramid blend. A masked blend of two images: (e) first input image, (f) second input image, (g) region mask, (h) blended image.*

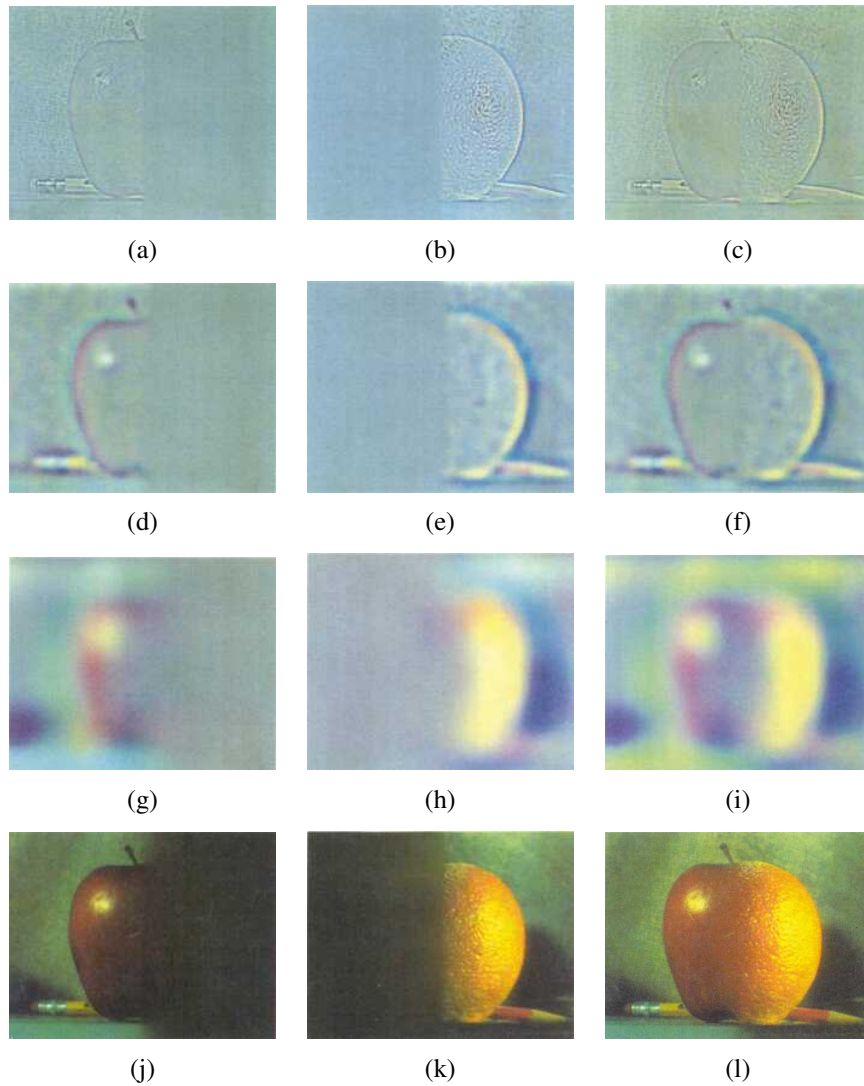


Figure 3.42 *Laplacian pyramid blending details (Burt and Adelson 1983b) © 1983 ACM. The first three rows show the high, medium, and low-frequency parts of the Laplacian pyramid (taken from levels 0, 2, and 4). The left and middle columns show the original apple and orange images weighted by the smooth interpolation functions, while the right column shows the averaged contributions.*

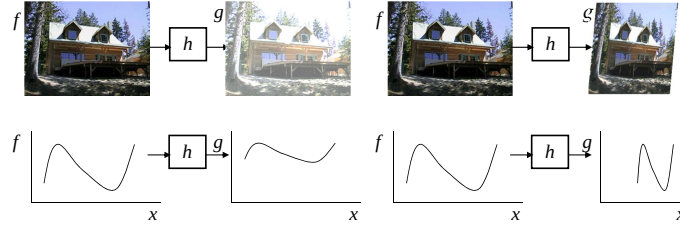


Figure 3.43 Image warping involves modifying the domain of an image function rather than its range.

3.6 Geometric transformations

In the previous sections, we saw how interpolation and decimation could be used to change the *resolution* of an image. In this section, we look at how to perform more general transformations, such as image rotations or general warps. In contrast to the point processes we saw in Section 3.1, where the function applied to an image transforms the *range* of the image,

$$g(\mathbf{x}) = h(f(\mathbf{x})), \quad (3.73)$$

here we look at functions that transform the *domain*,

$$g(\mathbf{x}) = f(\mathbf{h}(\mathbf{x})), \quad (3.74)$$

as shown in Figure 3.43.

We begin by studying the global *parametric* 2D transformation first introduced in Section 2.1.1. (Such a transformation is called parametric because it is controlled by a small number of parameters.) We then turn our attention to more local general deformations such as those defined on meshes (Section 3.6.2). Finally, we show in Section 3.6.3 how image warps can be combined with cross-dissolves to create interesting *morphs* (in-between animations). For readers interested in more details on these topics, there is an excellent survey by Heckbert (1986) as well as very accessible textbooks by Wolberg (1990), Gomes, Darsa *et al.* (1999) and Akenine-Möller and Haines (2002). Note that Heckbert’s survey is on *texture mapping*, which is how the computer graphics community refers to the topic of warping images onto surfaces.

3.6.1 Parametric transformations

Parametric transformations apply a global deformation to an image, where the behavior of the transformation is controlled by a small number of parameters. Figure 3.44 shows a few ex-

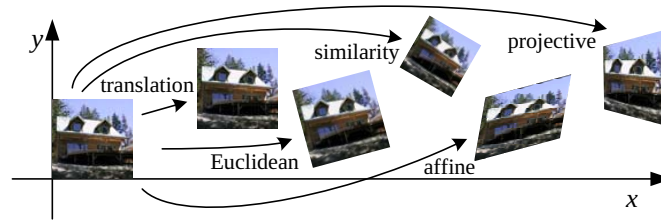


Figure 3.44 Basic set of 2D geometric image transformations.

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} \mathbf{I} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	2	orientation	
rigid (Euclidean)	$\begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	3	lengths	
similarity	$\begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	4	angles	
affine	$\begin{bmatrix} \mathbf{A} \end{bmatrix}_{2 \times 3}$	6	parallelism	
projective	$\begin{bmatrix} \tilde{\mathbf{H}} \end{bmatrix}_{3 \times 3}$	8	straight lines	

Table 3.3 Hierarchy of 2D coordinate transformations. Each transformation also preserves the properties listed in the rows below it, i.e., similarity preserves not only angles but also parallelism and straight lines. The 2×3 matrices are extended with a third $[\mathbf{0}^T \ 1]$ row to form a full 3×3 matrix for homogeneous coordinate transformations.

amples of such transformations, which are based on the 2D geometric transformations shown in Figure 2.4. The formulas for these transformations were originally given in Table 2.1 and are reproduced here in Table 3.3 for ease of reference.

In general, given a transformation specified by a formula $\mathbf{x}' = \mathbf{h}(\mathbf{x})$ and a source image $f(\mathbf{x})$, how do we compute the values of the pixels in the new image $g(\mathbf{x})$, as given in (3.74)? Think about this for a minute before proceeding and see if you can figure it out.

If you are like most people, you will come up with an algorithm that looks something like Algorithm 3.1. This process is called *forward warping* or *forward mapping* and is shown in Figure 3.45a. Can you think of any problems with this approach?

In fact, this approach suffers from several limitations. The process of copying a pixel $f(\mathbf{x})$ to a location $\mathbf{x}'$ in g is not well defined when $\mathbf{x}'$ has a non-integer value. What do we

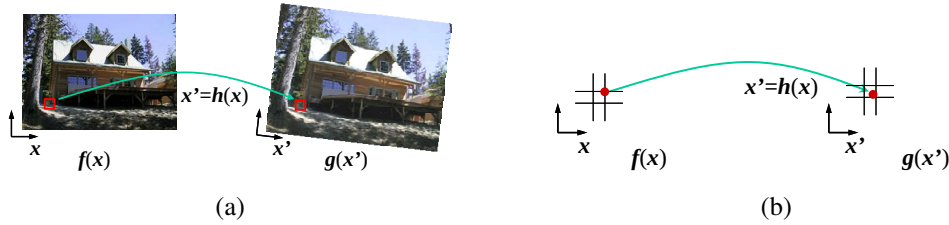


Figure 3.45 Forward warping algorithm: (a) a pixel $f(x)$ is copied to its corresponding location $x' = h(x)$ in image $g(x')$; (b) detail of the source and destination pixel locations.

procedure *forwardWarp*($f, h, \text{out } g$):

For every pixel x in $f(x)$

1. Compute the destination location $x' = h(x)$.
2. Copy the pixel $f(x)$ to $g(x')$.

Algorithm 3.1 Forward warping algorithm for transforming an image $f(x)$ into an image $g(x')$ through the parametric transform $x' = h(x)$.

do in such a case? What would you do?

You can round the value of x' to the nearest integer coordinate and copy the pixel there, but the resulting image has severe aliasing and pixels that jump around a lot when animating the transformation. You can also “distribute” the value among its four nearest neighbors in a weighted (bilinear) fashion, keeping track of the per-pixel weights and normalizing at the end. This technique is called *splatting* and is sometimes used for volume rendering in the graphics community (Levoy and Whitted 1985; Levoy 1988; Westover 1989; Rusinkiewicz and Levoy 2000). Unfortunately, it suffers from both moderate amounts of aliasing and a fair amount of blur (loss of high-resolution detail).

The second major problem with forward warping is the appearance of cracks and holes, especially when magnifying an image. Filling such holes with their nearby neighbors can lead to further aliasing and blurring.

What can we do instead? A preferable solution is to use *inverse warping* (Algorithm 3.2), where each pixel in the destination image $g(x')$ is sampled from the original image $f(x)$ (Figure 3.46).

How does this differ from the forward warping algorithm? For one thing, since $\hat{h}(x')$ is (presumably) defined for all pixels in $g(x')$, we no longer have holes. More importantly,

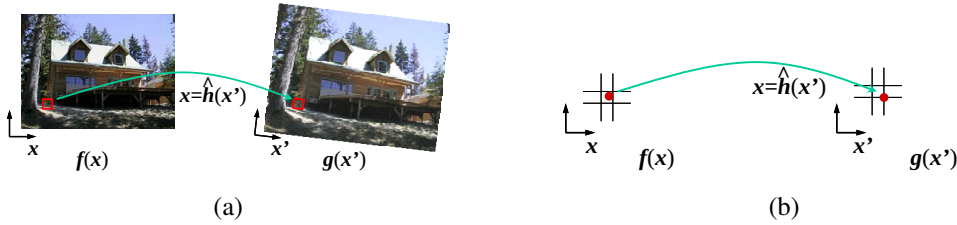


Figure 3.46 *Inverse warping algorithm: (a) a pixel $g(x')$ is sampled from its corresponding location $x = \hat{h}(x')$ in image $f(x)$; (b) detail of the source and destination pixel locations.*

procedure *inverseWarp*($f, h, \text{out } g$):

For every pixel x' in $g(x')$

1. Compute the source location $x = \hat{h}(x')$
2. Resample $f(x)$ at location x and copy to $g(x')$

Algorithm 3.2 *Inverse warping algorithm for creating an image $g(x')$ from an image $f(x)$ using the parametric transform $x' = h(x)$.*

resampling an image at non-integer locations is a well-studied problem (general image interpolation, see Section 3.5.2) and high-quality filters that control aliasing can be used.

Where does the function $\hat{h}(x')$ come from? Quite often, it can simply be computed as the inverse of $h(x)$. In fact, all of the parametric transforms listed in Table 3.3 have closed form solutions for the inverse transform: simply take the inverse of the 3×3 matrix specifying the transform.

In other cases, it is preferable to formulate the problem of image warping as that of resampling a source image $f(x)$ given a mapping $x = \hat{h}(x')$ from destination pixels x' to source pixels x . For example, in optical flow (Section 9.3), we estimate the flow field as the location of the *source* pixel that produced the current pixel whose flow is being estimated, as opposed to computing the *destination* pixel to which it is going. Similarly, when correcting for radial distortion (Section 2.1.5), we calibrate the lens by computing for each pixel in the final (undistorted) image the corresponding pixel location in the original (distorted) image.

What kinds of interpolation filter are suitable for the resampling process? Any of the filters we studied in Section 3.5.2 can be used, including nearest neighbor, bilinear, bicubic, and windowed sinc functions. While bilinear is often used for speed (e.g., inside the inner loop of a patch-tracking algorithm, see Section 9.1.3), bicubic, and windowed sinc are preferable

where visual quality is important.

To compute the value of $f(\mathbf{x})$ at a non-integer location $\mathbf{x}$, we simply apply our usual FIR resampling filter,

$$g(x, y) = \sum_{k, l} f(k, l) h(x - k, y - l), \quad (3.75)$$

where (x, y) are the sub-pixel coordinate values and $h(x, y)$ is some interpolating or smoothing kernel. Recall from Section 3.5.2 that when decimation is being performed, the smoothing kernel is stretched and re-scaled according to the downsampling rate r .

Unfortunately, for a general (non-zoom) image transformation, the resampling rate r is not well defined. Consider a transformation that stretches the x dimensions while squashing the y dimensions. The resampling kernel should be performing regular interpolation along the x dimension and smoothing (to anti-alias the blurred image) in the y direction. This gets even more complicated for the case of general affine or perspective transforms.

What can we do? Fortunately, Fourier analysis can help. The two-dimensional generalization of the one-dimensional *domain scaling* law is

$$g(\mathbf{A}\mathbf{x}) \Leftrightarrow |\mathbf{A}|^{-1} G(\mathbf{A}^{-T}\mathbf{f}). \quad (3.76)$$

For all of the transforms in Table 3.3 except perspective, the matrix $\mathbf{A}$ is already defined. For perspective transformations, the matrix $\mathbf{A}$ is the linearized *derivative* of the perspective transformation (Figure 3.47a), i.e., the local affine approximation to the stretching induced by the projection (Heckbert 1986; Wolberg 1990; Gomes, Darsa *et al.* 1999; Akenine-Möller and Haines 2002).

To prevent aliasing, we need to prefilter the image $f(\mathbf{x})$ with a filter whose frequency response is the projection of the final desired spectrum through the $\mathbf{A}^{-T}$ transform (Szeliski, Winder, and Uyttendaele 2010). In general (for non-zoom transforms), this filter is non-separable and hence is very slow to compute. Therefore, a number of approximations to this filter are used in practice, include MIP-mapping, elliptically weighted Gaussian averaging, and anisotropic filtering (Akenine-Möller and Haines 2002).

MIP-mapping

MIP-mapping was first proposed by Williams (1983) as a means to rapidly prefilter images being used for *texture mapping* in computer graphics. A MIP-map¹⁶ is a standard image pyramid (Figure 3.31), where each level is prefiltered with a high-quality filter rather than a poorer quality approximation, such as Burt and Adelson's (1983b) five-tap binomial. To resample

¹⁶The term "MIP" stands for *multi in parvo*, meaning "many in one".

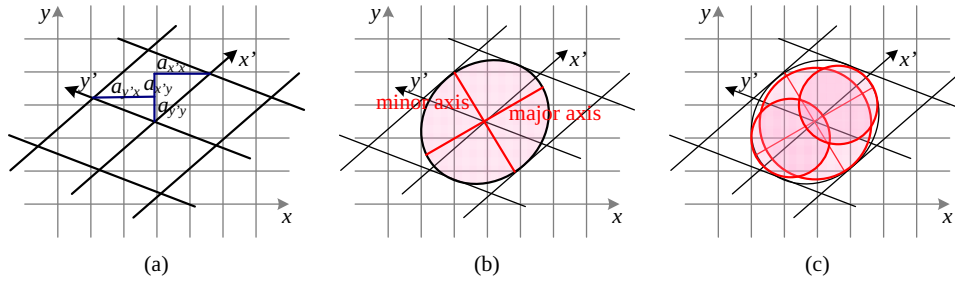


Figure 3.47 Anisotropic texture filtering: (a) Jacobian of transform $\mathbf{A}$ and the induced horizontal and vertical resampling rates $\{a_{x'x}, a_{x'y}, a_{y'x}, a_{y'y}\}$; (b) elliptical footprint of an EWA smoothing kernel; (c) anisotropic filtering using multiple samples along the major axis. Image pixels lie at line intersections.

an image from a MIP-map, a scalar estimate of the resampling rate r is first computed. For example, r can be the maximum of the absolute values in $\mathbf{A}$ (which suppresses aliasing) or it can be the minimum (which reduces blurring). Akenine-Möller and Haines (2002) discuss these issues in more detail.

Once a resampling rate has been specified, a *fractional* pyramid level is computed using the base 2 logarithm,

$$l = \log_2 r. \quad (3.77)$$

One simple solution is to resample the texture from the next higher or lower pyramid level, depending on whether it is preferable to reduce aliasing or blur. A better solution is to resample *both* images and blend them linearly using the fractional component of l . Since most MIP-map implementations use bilinear resampling within each level, this approach is usually called *trilinear MIP-mapping*. Computer graphics rendering APIs, such as OpenGL and Direct3D, have parameters that can be used to select which variant of MIP-mapping (and of the sampling rate r computation) should be used, depending on the desired tradeoff between speed and quality. Exercise 3.22 has you examine some of these tradeoffs in more detail.

Elliptical Weighted Average

The Elliptical Weighted Average (EWA) filter invented by Greene and Heckbert (1986) is based on the observation that the affine mapping $\mathbf{x} = \mathbf{A}\mathbf{x}'$ defines a skewed two-dimensional coordinate system in the vicinity of each source pixel $\mathbf{x}$ (Figure 3.47a). For every destination pixel $\mathbf{x}'$, the ellipsoidal projection of a small pixel grid in $\mathbf{x}'$ onto $\mathbf{x}$ is computed (Figure 3.47b). This is then used to filter the source image $g(\mathbf{x})$ with a Gaussian whose inverse

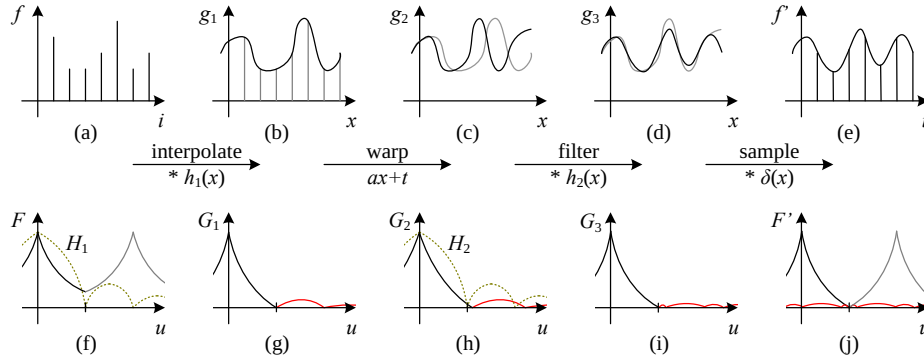


Figure 3.48 One-dimensional signal resampling (Szeliski, Winder, and Uyttendaele 2010): (a) original sampled signal $f(i)$; (b) interpolated signal $g_1(x)$; (c) warped signal $g_2(x)$; (d) filtered signal $g_3(x)$; (e) sampled signal $f'(i)$. The corresponding spectra are shown below the signals, with the aliased portions shown in red.

covariance matrix is this ellipsoid.

Despite its reputation as a high-quality filter (Akenine-Möller and Haines 2002), we have found in our work (Szeliski, Winder, and Uyttendaele 2010) that because a Gaussian kernel is used, the technique suffers simultaneously from both blurring and aliasing, compared to higher-quality filters. The EWA is also quite slow, although faster variants based on MIP-mapping have been proposed, as described in (Szeliski, Winder, and Uyttendaele 2010).

Anisotropic filtering

An alternative approach to filtering oriented textures, which is sometimes implemented in graphics hardware (GPUs), is to use anisotropic filtering (Barkans 1997; Akenine-Möller and Haines 2002). In this approach, several samples at different resolutions (fractional levels in the MIP-map) are combined along the major axis of the EWA Gaussian (Figure 3.47c).

Multi-pass transforms

The optimal approach to warping images without excessive blurring or aliasing is to adaptively prefilter the source image at each pixel using an ideal low-pass filter, i.e., an oriented skewed sinc or low-order (e.g., cubic) approximation (Figure 3.47a). Figure 3.48 shows how this works in one dimension. The signal is first (theoretically) interpolated to a continuous waveform, (ideally) low-pass filtered to below the new Nyquist rate, and then re-sampled to

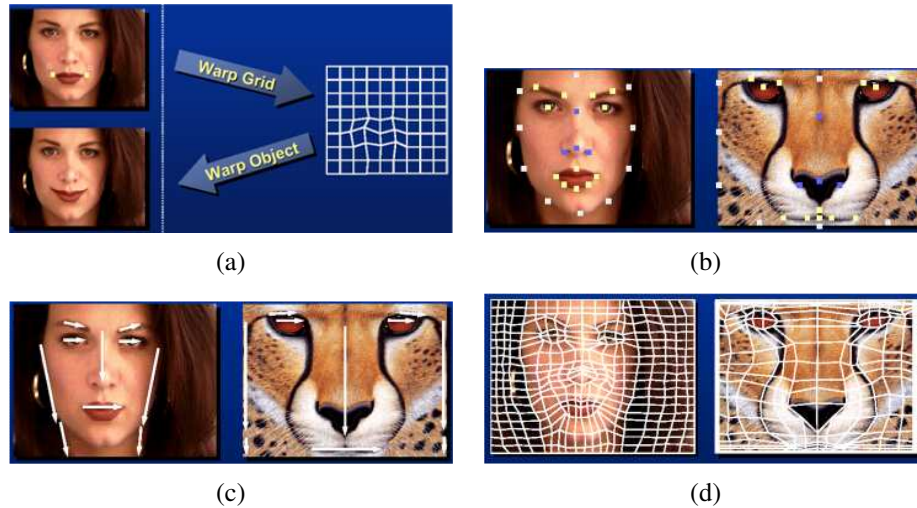


Figure 3.49 Image warping alternatives (Gomes, Darsa et al. 1999) © 1999 Morgan Kaufmann: (a) sparse control points $\rightarrow$ deformation grid; (b) denser set of control point correspondences; (c) oriented line correspondences; (d) uniform quadrilateral grid.

the final desired resolution. In practice, the interpolation and decimation steps are concatenated into a single *polyphase* digital filtering operation (Szeliski, Winder, and Uyttendaele 2010).

For parametric transforms, the oriented two-dimensional filtering and resampling operations can be approximated using a series of one-dimensional resampling and shearing transforms (Catmull and Smith 1980; Heckbert 1989; Wolberg 1990; Gomes, Darsa et al. 1999; Szeliski, Winder, and Uyttendaele 2010). The advantage of using a series of one-dimensional transforms is that they are much more efficient (in terms of basic arithmetic operations) than large, non-separable, two-dimensional filter kernels. In order to prevent aliasing, however, it may be necessary to upsample in the opposite direction before applying a shearing transformation (Szeliski, Winder, and Uyttendaele 2010).

3.6.2 Mesh-based warping

While parametric transforms specified by a small number of global parameters have many uses, *local* deformations with more degrees of freedom are often required.

Consider, for example, changing the appearance of a face from a frown to a smile (Figure 3.49a). What is needed in this case is to curve the corners of the mouth upwards while

leaving the rest of the face intact.¹⁷ To perform such a transformation, different amounts of motion are required in different parts of the image. Figure 3.49 shows some of the commonly used approaches.

The first approach, shown in Figure 3.49a–b, is to specify a *sparse* set of corresponding points. The displacement of these points can then be interpolated to a dense *displacement field* (Chapter 9) using a variety of techniques, which are described in more detail in Section 4.1 on scattered data interpolation. One possibility is to *triangulate* the set of points in one image (de Berg, Cheong *et al.* 2006; Litwinowicz and Williams 1994; Buck, Finkelstein *et al.* 2000) and to use an *affine* motion model (Table 3.3), specified by the three triangle vertices, inside each triangle. If the destination image is triangulated according to the new vertex locations, an inverse warping algorithm (Figure 3.46) can be used. If the source image is triangulated and used as a *texture map*, computer graphics rendering algorithms can be used to draw the new image (but care must be taken along triangle edges to avoid potential aliasing).

Alternative methods for interpolating a sparse set of displacements include moving nearby quadrilateral mesh vertices, as shown in Figure 3.49a, using *variational* (energy minimizing) interpolants such as regularization (Litwinowicz and Williams 1994), see Section 4.2, or using locally weighted (*radial basis function*) combinations of displacements (Section 4.1.1). (See Section 4.1 for additional *scattered data interpolation* techniques.) If quadrilateral meshes are used, it may be desirable to interpolate displacements down to individual pixel values using a smooth interpolant such as a quadratic B-spline (Farin 2002; Lee, Wolberg *et al.* 1996).

In some cases, e.g., if a dense depth map has been estimated for an image (Shade, Gortler *et al.* 1998), we only know the forward displacement for each pixel. As mentioned before, drawing source pixels at their destination location, i.e., forward warping (Figure 3.45), suffers from several potential problems, including aliasing and the appearance of small cracks. An alternative technique in this case is to forward warp the *displacement field* (or depth map) to its new location, fill small holes in the resulting map, and then use inverse warping to perform the resampling (Shade, Gortler *et al.* 1998). The reason that this generally works better than forward warping is that displacement fields tend to be much smoother than images, so the aliasing introduced during the forward warping of the displacement field is much less noticeable.

A second approach to specifying displacements for local deformations is to use corresponding *oriented line segments* (Beier and Neely 1992), as shown in Figures 3.49c and 3.50. Pixels along each line segment are transferred from source to destination exactly as specified, and other pixels are warped using a smooth interpolation of these displacements. Each line

¹⁷See Section 6.2.4 on active appearance models for more sophisticated examples of changing facial expression and appearance.

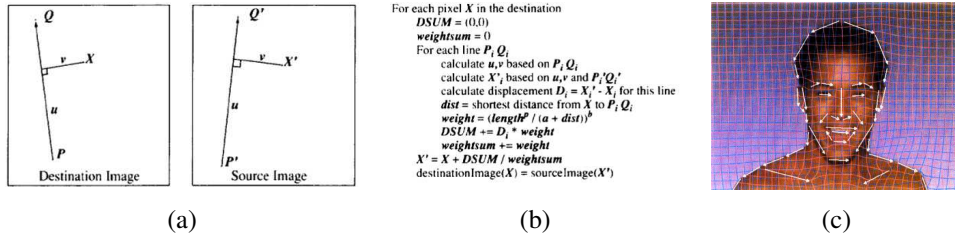


Figure 3.50 Line-based image warping (Beier and Neely 1992) © 1992 ACM: (a) distance computation and position transfer; (b) rendering algorithm; (c) two intermediate warps used for morphing.

segment correspondence specifies a translation, rotation, and scaling, i.e., a *similarity transform* (Table 3.3), for pixels in its vicinity, as shown in Figure 3.50a. Line segments influence the overall displacement of the image using a weighting function that depends on the minimum distance to the line segment (v in Figure 3.50a if $u \in [0, 1]$, else the shorter of the two distances to P and Q).

One final possibility for specifying displacement fields is to use a mesh specifically *adapted* to the underlying image content, as shown in Figure 3.49d. Specifying such meshes by hand can involve a fair amount of work; Gomes, Darsa *et al.* (1999) describe an interactive system for doing this. Once the two meshes have been specified, intermediate warps can be generated using linear interpolation and the displacements at mesh nodes can be interpolated using splines.

3.6.3 Application: Feature-based morphing

While warps can be used to change the appearance of or to animate a *single* image, even more powerful effects can be obtained by warping and blending two or more images using a process now commonly known as *morphing* (Beier and Neely 1992; Lee, Wolberg *et al.* 1996; Gomes, Darsa *et al.* 1999).

Figure 3.51 shows the essence of image morphing. Instead of simply cross-dissolving between two images, which leads to ghosting as shown in the top row, each image is warped toward the other image before blending, as shown in the bottom row. If the correspondences have been set up well (using any of the techniques shown in Figure 3.49), corresponding features are aligned and no ghosting results.

The above process is repeated for each intermediate frame being generated during a morph, using different blends (and amounts of deformation) at each interval. Let $t \in [0, 1]$ be the time parameter that describes the sequence of interpolated frames. The weighting func-

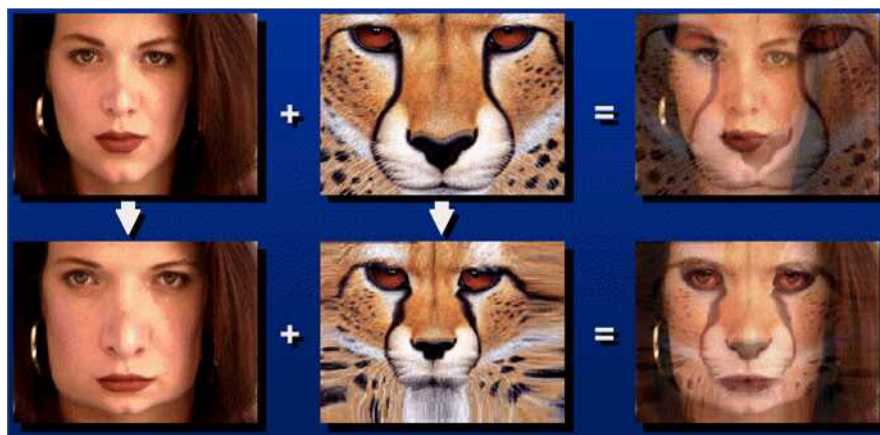


Figure 3.51 *Image morphing (Gomes, Darsa et al. 1999) © 1999 Morgan Kaufmann. Top row: if the two images are just blended, visible ghosting results. Bottom row: both images are first warped to the same intermediate location (e.g., halfway towards the other image) and the resulting warped images are then blended resulting in a seamless morph.*

tions for the two warped images in the blend are $(1 - t)$ and t and the movements of the pixels specified by the correspondences are also linearly interpolated. Some care must be taken in defining what it means to partially warp an image towards a destination, especially if the desired motion is far from linear (Sederberg, Gao *et al.* 1993). Exercise 3.25 has you implement a morphing algorithm and test it out under such challenging conditions.

3.7 Additional reading

If you are interested in exploring the topic of image processing in more depth, some popular textbooks have been written by Gomes and Velho (1997), Jähne (1997), Pratt (2007), Burger and Burge (2009), and Gonzalez and Woods (2017). The pre-eminent conference and journal in this field are the IEEE International Conference on Image Processing and the IEEE Transactions on Image Processing.

For image compositing operators, the seminal reference is by Porter and Duff (1984) while Blinn (1994a,b) provides a more detailed tutorial. For image compositing, Smith and Blinn (1996) were the first to bring this topic to the attention of the graphics community, while Wang and Cohen (2009) provide a good in-depth survey.

In the realm of linear filtering, Freeman and Adelson (1991) provide a great introduction to separable and steerable oriented band-pass filters, while Perona (1995) shows how to

approximate any filter as a sum of separable components.

The literature on non-linear filtering is quite wide and varied; it includes such topics as bi-lateral filtering (Tomasi and Manduchi 1998; Durand and Dorsey 2002; Chen, Paris, and Durand 2007; Paris and Durand 2009; Paris, Kornprobst *et al.* 2008), related iterative algorithms (Saint-Marc, Chen, and Medioni 1991; Nielsen, Florack, and Deriche 1997; Black, Sapiro *et al.* 1998; Weickert, ter Haar Romeny, and Viergever 1998; Weickert 1998; Barash 2002; Scharr, Black, and Haussecker 2003; Barash and Comaniciu 2004) and variational approaches (Chan, Osher, and Shen 2001; Tschumperlé and Deriche 2005; Tschumperlé 2006; Kaftory, Schechner, and Zeevi 2007), and guided filtering (Eisemann and Durand 2004; Petschnigg, Agrawala *et al.* 2004; He, Sun, and Tang 2013).

Good references to image morphology include Haralick and Shapiro (1992, Section 5.2), Bovik (2000, Section 2.2), Ritter and Wilson (2000, Section 7) Serra (1982), Serra and Vincent (1992), Yuille, Vincent, and Geiger (1992), and Soille (2006).

The classic papers for image pyramids and pyramid blending are by Burt and Adelson (1983a,b). Wavelets were first introduced to the computer vision community by Mallat (1989) and good tutorial and review papers and books are available (Strang 1989; Simoncelli and Adelson 1990b; Rioul and Vetterli 1991; Chui 1992; Meyer 1993; Sweldens 1997). Wavelets are widely used in the computer graphics community to perform multi-resolution geometric processing (Stollnitz, DeRose, and Salesin 1996) and have been used in computer vision for similar applications (Szeliski 1990b; Pentland 1994; Gortler and Cohen 1995; Yaou and Chang 1994; Lai and Vemuri 1997; Szeliski 2006b; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013), as well as for multi-scale oriented filtering (Simoncelli, Freeman *et al.* 1992) and denoising (Portilla, Strela *et al.* 2003).

While image pyramids (Section 3.5.3) are usually constructed using linear filtering operators, more recent work uses non-linear filters, since these can better preserve details and other salient features. Some representative papers in the computer vision literature are by Gluckman (2006a,b); Lyu and Simoncelli (2008) and in computational photography by Bae, Paris, and Durand (2006), Farbman, Fattal *et al.* (2008), and Fattal (2009).

High-quality algorithms for image warping and resampling are covered both in the image processing literature (Wolberg 1990; Dodgson 1992; Gomes, Darsa *et al.* 1999; Szeliski, Winder, and Uyttendaele 2010) and in computer graphics (Williams 1983; Heckbert 1986; Barkan 1997; Weinhaus and Devarajan 1997; Akenine-Möller and Haines 2002), where they go under the name of *texture mapping*. Combinations of image warping and image blending techniques are used to enable *morphing* between images, which is covered in a series of seminal papers and books (Beier and Neely 1992; Gomes, Darsa *et al.* 1999).

3.8 Exercises

Ex 3.1: Color balance. Write a simple application to change the color balance of an image by multiplying each color value by a different user-specified constant. If you want to get fancy, you can make this application interactive, with sliders.

1. Do you get different results if you take out the gamma transformation before or after doing the multiplication? Why or why not?
2. Take the same picture with your digital camera using different color balance settings (most cameras control the color balance from one of the menus). Can you recover what the color balance ratios are between the different settings? You may need to put your camera on a tripod and align the images manually or automatically to make this work. Alternatively, use a color checker chart (Figure 10.3b), as discussed in Sections 2.3 and 10.1.1.
3. Can you think of any reason why you might want to perform a color twist (Section 3.1.2) on the images? See also Exercise 2.8 for some related ideas.

Ex 3.2: Demosaicing. If you have access to the RAW image for the camera, perform the demosaicing yourself (Section 10.3.1). If not, just subsample an RGB image in a Bayer mosaic pattern. Instead of just bilinear interpolation, try one of the more advanced techniques described in Section 10.3.1. Compare your result to the one produced by the camera. Does your camera perform a simple linear mapping between RAW values and the color-balanced values in a JPEG? Some high-end cameras have a RAW+JPEG mode, which makes this comparison much easier.

Ex 3.3: Compositing and reflections. Section 3.1.3 describes the process of compositing an alpha-matted image on top of another. Answer the following questions and optionally validate them experimentally:

1. Most captured images have gamma correction applied to them. Does this invalidate the basic compositing equation (3.8); if so, how should it be fixed?
2. The additive (pure reflection) model may have limitations. What happens if the glass is tinted, especially to a non-gray hue? How about if the glass is dirty or smudged? How could you model wavy glass or other kinds of refractive objects?

Ex 3.4: Blue screen matting. Set up a blue or green background, e.g., by buying a large piece of colored posterboard. Take a picture of the empty background, and then of the background with a new object in front of it. *Pull the matte* using the difference between each colored pixel and its assumed corresponding background pixel, using one of the techniques described in Section 3.1.3 or by [Smith and Blinn \(1996\)](#).

Ex 3.5: Difference keying. Implement a difference keying algorithm (see Section 3.1.3) ([Toyama, Krumm et al. 1999](#)), consisting of the following steps:

1. Compute the mean and variance (or median and robust variance) at each pixel in an “empty” video sequence.
2. For each new frame, classify each pixel as foreground or background (set the background pixels to RGBA=0).
3. (Optional) Compute the alpha channel and composite over a new background.
4. (Optional) Clean up the image using morphology (Section 3.3.1), label the connected components (Section 3.3.3), compute their centroids, and track them from frame to frame. Use this to build a “people counter”.

Ex 3.6: Photo effects. Write a variety of photo enhancement or effects filters: contrast, solarization (quantization), etc. Which ones are useful (perform sensible corrections) and which ones are more creative (create unusual images)?

Ex 3.7: Histogram equalization. Compute the gray level (luminance) histogram for an image and equalize it so that the tones look better (and the image is less sensitive to exposure settings). You may want to use the following steps:

1. Convert the color image to luminance (Section 3.1.2).
2. Compute the histogram, the cumulative distribution, and the compensation transfer function (Section 3.1.4).
3. (Optional) Try to increase the “punch” in the image by ensuring that a certain fraction of pixels (say, 5%) are mapped to pure black and white.
4. (Optional) Limit the local *gain* $f'(I)$ in the transfer function. One way to do this is to limit $f(I) < \gamma I$ or $f'(I) < \gamma$ while performing the accumulation (3.9), keeping any unaccumulated values “in reserve”. (I’ll let you figure out the exact details.)

5. Compensate the luminance channel through the lookup table and re-generate the color image using color ratios (2.117).
6. (Optional) Color values that are *clipped* in the original image, i.e., have one or more saturated color channels, may appear unnatural when remapped to a non-clipped value. Extend your algorithm to handle this case in some useful way.

Ex 3.8: Local histogram equalization. Compute the gray level (luminance) histograms for each patch, but add to vertices based on distance (a spline).

1. Build on Exercise 3.7 (luminance computation).
2. Distribute values (counts) to adjacent vertices (bilinear).
3. Convert to CDF (look-up functions).
4. (Optional) Use low-pass filtering of CDFs.
5. Interpolate adjacent CDFs for final lookup.

Ex 3.9: Padding for neighborhood operations. Write down the formulas for computing the padded pixel values $\tilde{f}(i, j)$ as a function of the original pixel values $f(k, l)$ and the image width and height (M, N) for *each* of the padding modes shown in Figure 3.13. For example, for replication (clamping),

$$\tilde{f}(i, j) = f(k, l), \quad \begin{aligned} k &= \max(0, \min(M - 1, i)), \\ l &= \max(0, \min(N - 1, j)), \end{aligned}$$

(Hint: you may want to use the min, max, mod, and absolute value operators in addition to the regular arithmetic operators.)

- Describe in more detail the advantages and disadvantages of these various modes.
- (Optional) Check what your graphics card does by drawing a texture-mapped rectangle where the texture coordinates lie beyond the $[0.0, 1.0]$ range and using different texture clamping modes.

Ex 3.10: Separable filters. Implement convolution with a separable kernel. The input should be a grayscale or color image along with the horizontal and vertical kernels. Make sure you support the padding mechanisms developed in the previous exercise. You will need this functionality for some of the later exercises. If you already have access to separable filtering in an image processing package you are using (such as IPL), skip this exercise.

- (Optional) Use Pietro Perona's (1995) technique to approximate convolution as a sum of a number of separable kernels. Let the user specify the number of kernels and report back some sensible metric of the approximation fidelity.

Ex 3.11: Discrete Gaussian filters. Discuss the following issues with implementing a discrete Gaussian filter:

- If you just sample the equation of a continuous Gaussian filter at discrete locations, will you get the desired properties, e.g., will the coefficients sum up to 1? Similarly, if you sample a derivative of a Gaussian, do the samples sum up to 0 or have vanishing higher-order moments?
- Would it be preferable to take the original signal, interpolate it with a sinc, blur with a continuous Gaussian, then prefilter with a sinc before re-sampling? Is there a simpler way to do this in the frequency domain?
- Would it make more sense to produce a Gaussian frequency response in the Fourier domain and to then take an inverse FFT to obtain a discrete filter?
- How does truncation of the filter change its frequency response? Does it introduce any additional artifacts?
- Are the resulting two-dimensional filters as rotationally invariant as their continuous analogs? Is there some way to improve this? In fact, can any 2D discrete (separable or non-separable) filter be truly rotationally invariant?

Ex 3.12: Sharpening, blur, and noise removal. Implement some softening, sharpening, and non-linear diffusion (selective sharpening or noise removal) filters, such as Gaussian, median, and bilateral (Section 3.3.1), as discussed in Section 3.4.2.

Take blurry or noisy images (shooting in low light is a good way to get both) and try to improve their appearance and legibility.

Ex 3.13: Steerable filters. Implement Freeman and Adelson's (1991) steerable filter algorithm. The input should be a grayscale or color image and the output should be a multi-banded image consisting of $G_1^{0^\circ}$ and $G_1^{90^\circ}$. The coefficients for the filters can be found in the paper by Freeman and Adelson (1991).

Test the various order filters on a number of images of your choice and see if you can reliably find corner and intersection features. These filters will be quite useful later to detect elongated structures, such as lines (Section 7.4).

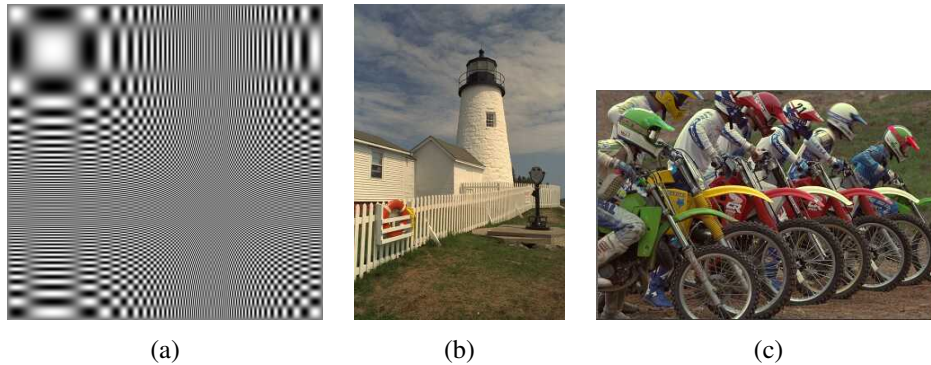


Figure 3.52 *Sample images for testing the quality of resampling algorithms: (a) a synthetic chirp; (b) and (c) some high-frequency images from the image compression community.*

Ex 3.14: Bilateral and guided image filters. Implement or download code for bilateral and/or guided image filtering and use this to implement some image enhancement or processing application, such as those described in Section 3.3.2

Ex 3.15: Fourier transform. Prove the properties of the Fourier transform listed in Szeliski (2010, Table 3.1) and derive the formulas for the Fourier transforms pairs listed in Szeliski (2010, Table 3.2) and Table 3.1. These exercises are very useful if you want to become comfortable working with Fourier transforms, which is a very useful skill when analyzing and designing the behavior and efficiency of many computer vision algorithms.

Ex 3.16: High-quality image resampling. Implement several of the low-pass filters presented in Section 3.5.2 and also the windowed sinc shown in Figure 3.28. Feel free to implement other filters (Wolberg 1990; Unser 1999).

Apply your filters to continuously resize an image, both magnifying (interpolating) and minifying (decimating) it; compare the resulting animations for several filters. Use both a synthetic chirp image (Figure 3.52a) and natural images with lots of high-frequency detail (Figure 3.52b–c).

You may find it helpful to write a simple visualization program that continuously plays the animations for two or more filters at once and that let you “blink” between different results.

Discuss the merits and deficiencies of each filter, as well as the tradeoff between speed and quality.

Ex 3.17: Pyramids. Construct an image pyramid. The inputs should be a grayscale or color image, a separable filter kernel, and the number of desired levels. Implement at least the following kernels:

- 2×2 block filtering;
- Burt and Adelson’s binomial kernel $1/16(1, 4, 6, 4, 1)$ (Burt and Adelson 1983a);
- a high-quality seven- or nine-tap filter.

Compare the visual quality of the various decimation filters. Also, shift your input image by 1 to 4 pixels and compare the resulting decimated (quarter size) image sequence.

Ex 3.18: Pyramid blending. Write a program that takes as input two color images and a binary mask image and produces the Laplacian pyramid blend of the two images.

1. Construct the Laplacian pyramid for each image.
2. Construct the Gaussian pyramid for the two mask images (the input image and its complement).
3. Multiply each Laplacian image by its corresponding mask and sum the images (see Figure 3.41).
4. Reconstruct the final image from the blended Laplacian pyramid.

Generalize your algorithm to input n images and a label image with values $1 \dots n$ (the value 0 can be reserved for “no input”). Discuss whether the weighted summation stage (step 3) needs to keep track of the total weight for renormalization, or whether the math just works out. Use your algorithm either to blend two differently exposed image (to avoid under- and over-exposed regions) or to make a creative blend of two different scenes.

Ex 3.19: Pyramid blending in PyTorch. Re-write your pyramid blending exercise in PyTorch.

1. PyTorch has support for all of the primitives you need, i.e., fixed size convolutions (make sure they filter each channel separately), downsampling, upsampling, and addition, subtraction, and multiplication (although the latter is rarely used).
2. The goal of this exercise is *not* to train the convolution weights, but just to become familiar with the DNN primitives available in PyTorch.
3. Compare your results to the ones using a standard Python or C++ computer vision library. They should be identical.
4. Discuss whether you like this API better or worse for these kinds of fixed pipeline imaging tasks.

Ex 3.20: Local Laplacian—challenging. Implement the local Laplacian contrast manipulation technique (Paris, Hasinoff, and Kautz 2011; Aubry, Paris *et al.* 2014) and use this to implement edge-preserving filtering and tone manipulation.

Ex 3.21: Wavelet construction and applications. Implement one of the wavelet families described in Section 3.5.4 or by Simoncelli and Adelson (1990b), as well as the basic Laplacian pyramid (Exercise 3.17). Apply the resulting representations to one of the following two tasks:

- **Compression:** Compute the entropy in each band for the different wavelet implementations, assuming a given quantization level (say, $1/4$ gray level, to keep the rounding error acceptable). Quantize the wavelet coefficients and reconstruct the original images. Which technique performs better? (See Simoncelli and Adelson (1990b) or any of the multitude of wavelet compression papers for some typical results.)
- **Denoising.** After computing the wavelets, suppress small values using *coring*, i.e., set small values to zero using a piecewise linear or other C^0 function. Compare the results of your denoising using different wavelet and pyramid representations.

Ex 3.22: Parametric image warping. Write the code to do affine and perspective image warps (optionally bilinear as well). Try a variety of interpolants and report on their visual quality. In particular, discuss the following:

- In a MIP-map, selecting only the coarser level adjacent to the computed fractional level will produce a blurrier image, while selecting the finer level will lead to aliasing. Explain why this is so and discuss whether blending an aliased and a blurred image (tri-linear MIP-mapping) is a good idea.
- When the ratio of the horizontal and vertical resampling rates becomes very different (anisotropic), the MIP-map performs even worse. Suggest some approaches to reduce such problems.

Ex 3.23: Local image warping. Open an image and deform its appearance in one of the following ways:

1. Click on a number of pixels and move (drag) them to new locations. Interpolate the resulting sparse displacement field to obtain a dense motion field (Sections 3.6.2 and 3.5.1).
2. Draw a number of lines in the image. Move the endpoints of the lines to specify their new positions and use the Beier–Neely interpolation algorithm (Beier and Neely 1992), discussed in Section 3.6.2, to get a dense motion field.

3. Overlay a spline control grid and move one grid point at a time (optionally select the level of the deformation).
4. Have a dense per-pixel flow field and use a soft “paintbrush” to design a horizontal and vertical velocity field.
5. (Optional): Prove whether the Beier–Neely warp does or does not reduce to a sparse point-based deformation as the line segments become shorter (reduce to points).

Ex 3.24: Forward warping. Given a displacement field from the previous exercise, write a forward warping algorithm:

1. Write a forward warper using splatting, either nearest neighbor or soft accumulation (Section 3.6.1).
2. Write a two-pass algorithm that forward warps the displacement field, fills in small holes, and then uses inverse warping (Shade, Gortler *et al.* 1998).
3. Compare the quality of these two algorithms.

Ex 3.25: Feature-based morphing. Extend the warping code you wrote in Exercise 3.23 to import two different images and specify correspondences (point, line, or mesh-based) between the two images.

1. Create a morph by partially warping the images towards each other and cross-dissolving (Section 3.6.3).
2. Try using your morphing algorithm to perform an image rotation and discuss whether it behaves the way you want it to.

Ex 3.26: 2D image editor. Extend the program you wrote in Exercise 2.2 to import images and let you create a “collage” of pictures. You should implement the following steps:

1. Open up a new image (in a separate window).
2. Shift drag (rubber-band) to crop a subregion (or select whole image).
3. Paste into the current canvas.
4. Select the deformation mode (motion model): translation, rigid, similarity, affine, or perspective.
5. Drag any corner of the outline to change its transformation.



Figure 3.53 *There is a faint image of a rainbow visible in the right-hand side of this picture. Can you think of a way to enhance it (Exercise 3.29)?*

6. (Optional) Change the relative ordering of the images and which image is currently being manipulated.

The user should see the composition of the various images' pieces on top of each other.

This exercise should be built on the image transformation classes supported in the software library. Persistence of the created representation (save and load) should also be supported (for each image, save its transformation).

Ex 3.27: 3D texture-mapped viewer. Extend the viewer you created in Exercise 2.3 to include texture-mapped polygon rendering. Augment each polygon with (u, v, w) coordinates into an image.

Ex 3.28: Image denoising. Implement at least two of the various image denoising techniques described in this chapter and compare them on both synthetically noised image sequences and real-world (low-light) sequences. Does the performance of the algorithm depend on the correct choice of noise level estimate? Can you draw any conclusions as to which techniques work better?

Ex 3.29: Rainbow enhancer—challenging. Take a picture containing a rainbow, such as Figure 3.53, and enhance the strength (saturation) of the rainbow.

1. Draw an arc in the image delineating the extent of the rainbow.
2. Fit an *additive* rainbow function (explain why it is additive) to this arc (it is best to work with linearized pixel values), using the spectrum as the cross-section, and estimating the width of the arc and the amount of color being added. This is the trickiest part of

the problem, as you need to tease apart the (low-frequency) rainbow pattern and the natural image hiding behind it.

3. Amplify the rainbow signal and add it back into the image, re-applying the gamma function if necessary to produce the final image.

Chapter 4

Model fitting and optimization

4.1	Scattered data interpolation	194
4.1.1	Radial basis functions	196
4.1.2	Overfitting and underfitting	199
4.1.3	Robust data fitting	202
4.2	Variational methods and regularization	204
4.2.1	Discrete energy minimization	206
4.2.2	Total variation	210
4.2.3	Bilateral solver	210
4.2.4	<i>Application: Interactive colorization</i>	211
4.3	Markov random fields	212
4.3.1	Conditional random fields	222
4.3.2	<i>Application: Interactive segmentation</i>	227
4.4	Additional reading	230
4.5	Exercises	232

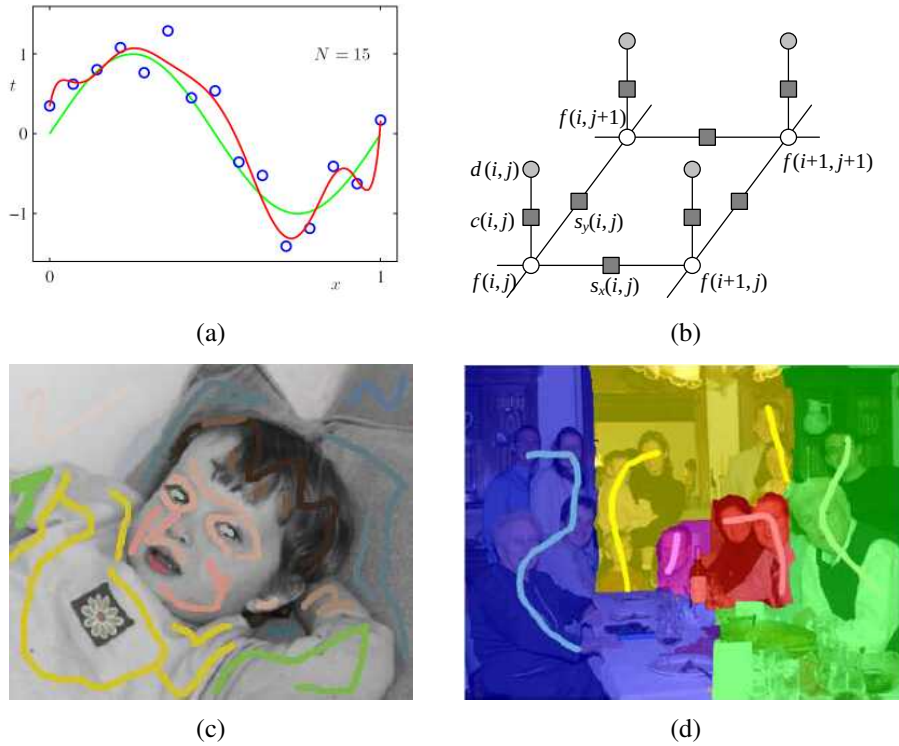


Figure 4.1 Examples of data interpolation and global optimization: (a) scattered data interpolation (curve fitting) (Bishop 2006) © 2006 Springer; (b) graphical model interpretation of first-order regularization; (c) colorization using optimization (Levin, Lischinski, and Weiss 2004) © 2004 ACM; (d) multi-image photomontage formulated as an unordered label MRF (Agarwala, Dontcheva et al. 2004) © 2004 ACM.

In the previous chapter, we covered a large number of image processing operators that take as input one or more images and produce some filtered or transformed version of these images. In many situations, however, we are given *incomplete* data as input, such as depths at a sparse number of locations, or user scribbles suggesting how an image should be colorized or segmented (Figure 4.1c–d).

The problem of interpolating a complete image (or more generally a *function* or *field*) from incomplete or varying quality data is often called *scattered data interpolation*. We begin this chapter with a review of techniques in this area, since in addition to being widely used in computer vision, they also form the basis of most machine learning algorithms, which we will study in the next chapter.

Instead of doing an exhaustive survey, we present in Section 4.1 some easy-to-use techniques, such as triangulation, spline interpolation, and radial basis functions. While these techniques are widely used, they cannot easily be modified to provide *controlled continuity*, i.e., to produce the kinds of piecewise continuous reconstructions we expect when estimating depth maps, label maps, or even color images.

For this reason, we introduce in Section 4.2 *variational methods*, which formulate the interpolation problem as the recovery of a piecewise smooth function subject to exact or approximate data constraints. Because the smoothness is controlled using penalties formulated as norms of the function, this class of techniques are often called *regularization* or *energy-based* approaches. To find the minimum-energy solutions to these problems, we discretize them (typically on a pixel grid), resulting in a discrete energy, which can then be minimized using sparse linear systems or related iterative techniques.

In the last part of this chapter, Section 4.3, we show how such energy-based formulations are related to Bayesian inference techniques formulated as *Markov random fields*, which are a special case of general probabilistic *graphical models*. In these formulations, data constraints can be interpreted as noisy and/or incomplete measurements, and piecewise smoothness constraints as *prior assumptions* or *models* over the solution space. Such formulations are also often called *generative models*, since we can, in principle, generate random samples from the prior distribution to see if they conform with our expectations. Because the prior models can be more complex than simple smoothness constraints, and because the solution space can have multiple local minima, more sophisticated optimization techniques have been developed, which we discuss in this section.

4.1 Scattered data interpolation

The goal of *scattered data interpolation* is to produce a (usually continuous and smooth) function $f(\mathbf{x})$ that passes *through* a set of data points $\mathbf{d}_k$ placed at locations $\mathbf{x}_k$ such that

$$f(\mathbf{x}_k) = \mathbf{d}_k. \quad (4.1)$$

The related problem of *scattered data approximation* only requires the function to pass *near* the data points (Amidror 2002; Wendland 2004; Anjyo, Lewis, and Pighin 2014). This is usually formulated using a penalty function such as

$$E_D = \sum_k \|f(\mathbf{x}_k) - \mathbf{d}_k\|^2, \quad (4.2)$$

with the squared norm in the above formula sometimes replaced by a different norm or robust function (Section 4.1.3). In statistics and machine learning, the problem of predicting an output function given a finite number of samples is called *regression* (Section 5.1). The $\mathbf{x}$ vectors are called the *inputs* and the outputs $\mathbf{y}$ are called the *targets*. Figure 4.1a shows an example of one-dimensional scattered data interpolation, while Figures 4.2 and 4.8 show some two-dimensional examples.

At first glance, scattered data interpolation seems closely related to *image interpolation*, which we studied in Section 3.5.1. However, unlike images, which are regularly gridded, the data points in scattered data interpolation are irregularly placed throughout the domain, as shown in Figure 4.2. This requires some adjustments to the interpolation methods we use.

If the domain $\mathbf{x}$ is two-dimensional, as is the case with images, one simple approach is to *triangulate* the domain $\mathbf{x}$ using the data locations $\mathbf{x}_k$ as the triangle vertices. The resulting triangular network, shown in Figure 4.2a, is called a *triangular irregular network* (TIN), and was one of the early techniques used to produce elevation maps from scattered field measurements collected by surveys.

The triangulation in Figure 4.2a was produced using a *Delaunay triangulation*, which is the most widely used planar triangulation technique due to its attractive computational properties, such as the avoidance of long skinny triangles. Algorithms for efficiently computing such triangulation are readily available¹ and covered in textbooks on computational geometry (Preparata and Shamos 1985; de Berg, Cheong *et al.* 2008). The Delaunay triangulation can be extended to higher-dimensional domains using the property of circumscribing spheres, i.e., the requirement that all selected simplices (triangles, tetrahedra, etc.) have no other vertices inside their circumscribing spheres.

¹For example, <https://docs.scipy.org/doc/scipy/reference/tutorial/spatial.html>

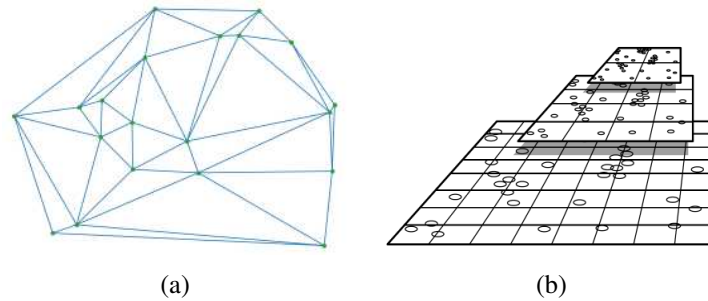


Figure 4.2 Some simple scattered data interpolation and approximation algorithms: (a) a Delaunay triangulation defined over a set of data point locations; (b) data structure and intermediate results for the pull-push algorithm (Gortler, Grzeszczuk et al. 1996) © 1996 ACM.

Once the triangulation has been defined, it is straightforward to define a piecewise-linear interpolant over each triangle, resulting in an interpolant that is C_0 but not generally C_1 continuous. The formulas for the function inside each triangle are usually derived using *barycentric coordinates*, which attain their maximal values at the vertices and sum up to one (Farin 2002; Amidror 2002).

If a smoother surface is desired as the interpolant, we can replace the piecewise linear functions on each triangle with higher-order *splines*, much as we did for image interpolation (Section 3.5.1). However, since these splines are now defined over irregular triangulations, more sophisticated techniques must be used (Farin 2002; Amidror 2002). Other, more recent interpolators based on geometric modeling techniques in computer graphics include subdivision surfaces (Peters and Reif 2008).

An alternative to triangulating the data points is to use a regular n -dimensional grid, as shown in Figure 4.2b. Splines defined on such domains are often called *tensor product splines* and have been used to interpolate scattered data (Lee, Wolberg, and Shin 1997).

An even faster, but less accurate, approach is called the *pull-push* algorithm and was originally developed for interpolating missing 4D lightfield samples in a Lumigraph (Gortler, Grzeszczuk et al. 1996). The algorithm proceeds in three phases, as schematically illustrated in Figure 4.2b.

First, the irregular data samples are *splatted* onto (i.e., spread across) the nearest grid vertices, using the same approach we discussed in Section 3.6.1 on parametric image transformations. The splatting operations accumulate both values and weights at nearby vertices. In the second, *pull*, phase, values and weights are computed at a hierarchical set of lower resolution grids by combining the coefficient values from the higher resolution grids. In the lower

resolution grids, the gaps (regions where the weights are low) become smaller. In the third, *push*, phase, information from each lower resolution grid is combined with the next higher resolution grid, filling in the gaps while not unduly blurring the higher resolution information already computed. Details of these three stages can be found in (Gortler, Grzeszczuk *et al.* 1996).

The pull-push algorithm is very fast, since it is essentially linear in the number of input data points and fine-level grid samples.

4.1.1 Radial basis functions

While the mesh-based representations I have just described can provide good-quality interpolants, they are typically limited to low-dimensional domains, because the size of the mesh grows combinatorially with the dimensionality of the domain. In higher dimensions, it is common to use *mesh-free* approaches that define the desired interpolant as a weighted sum of basis functions, similar to the formulation used in image interpolation (3.64). In machine learning, such approaches are often called *kernel functions* or *kernel regression* (Bishop 2006, Chapter 6; Murphy 2012, Chapter 14; Schölkopf and Smola 2001).

In more detail, the interpolated function f is a weighted sum (or *superposition*) of basis functions centered at each input data point

$$\mathbf{f}(\mathbf{x}) = \sum_k \mathbf{w}_k \phi(\|\mathbf{x} - \mathbf{x}_k\|), \quad (4.3)$$

where the $\mathbf{x}_k$ are the locations of the scattered data points, the ϕ s are the *radial basis functions* (or kernels), and $\mathbf{w}_k$ are the local *weights* associated with each kernel. The basis functions $\phi()$ are called *radial* because they are applied to the radial distance between a data sample $\mathbf{x}_k$ and an evaluation point $\mathbf{x}$. The choice of ϕ determines the smoothness properties of the interpolant, while the choice of weights w_k determines how closely the function approximates the input.

Some commonly used basis functions (Anjyo, Lewis, and Pighin 2014) include

$$\text{Gaussian} \quad \phi(r) = \exp(-r^2/c^2) \quad (4.4)$$

$$\text{Hardy multiquadric} \quad \phi(r) = \sqrt{r^2 + c^2} \quad (4.5)$$

$$\text{Inverse multiquadric} \quad \phi(r) = 1/\sqrt{r^2 + c^2} \quad (4.6)$$

$$\text{Thin plate spline} \quad \phi(r) = r^2 \log r. \quad (4.7)$$

In these equations, r is the radial distance and c is a scale parameter that controls the size (radial falloff) of the basis functions, and hence its smoothness (more compact bases lead to

“peakier” solutions). The thin plate spline equation holds for two dimensions (the general n -dimensional spline is called the *polyharmonic spline* and is given in (Anjyo, Lewis, and Pighin 2014)) and is the analytic solution to the second degree variational spline derived in (4.19).

If we want our function to exactly interpolate the data values, we solve the linear system of equations (4.1), i.e.,

$$\mathbf{f}(\mathbf{x}_k) = \sum_l \mathbf{w}_l \phi(\|\mathbf{x}_k - \mathbf{x}_l\|) = \mathbf{d}_k, \quad (4.8)$$

to obtain the desired set of weights $\mathbf{w}_k$. Note that for large amounts of basis function overlap (large values of c), these equations may be quite *ill-conditioned*, i.e., small changes in data values or locations can result in large changes in the interpolated function. Note also that the solution of such a system of equations is in general $O(m^3)$, where m is the number of data points (unless we use basis functions with finite extent to obtain a sparse set of equations).

A more prudent approach is to solve the *regularized data approximation problem*, which involves minimizing the data constraint energy (4.2) together with a weight penalty (*regularizer*) of the form

$$E_W = \sum_k \|\mathbf{w}_k\|^p, \quad (4.9)$$

and to then minimize the regularized least squares problem

$$E(\{w_k\}) = E_D + \lambda E_W \quad (4.10)$$

$$= \sum_k \left\| \sum_l \mathbf{w}_l \phi(\|\mathbf{x}_k - \mathbf{x}_l\|) - \mathbf{d}_k \right\|^2 + \lambda \sum_k \|\mathbf{w}_k\|^p. \quad (4.11)$$

When $p = 2$ (quadratic weight penalty), the resulting energy is a pure least squares problem, and can be solved using the *normal equations* (Appendix A.2), where the λ value gets added along the diagonal to stabilize the system of equations.

In statistics and machine learning, the quadratic (regularized least squares) problem is called *ridge regression*. In neural networks, adding a quadratic penalty on the weights is called *weight decay*, because it encourages weights to decay towards zero (Section 5.3.3). When $p = 1$, the technique is called *lasso* (least absolute shrinkage and selection operator), since for sufficiently large values of λ , many of the weights $\mathbf{w}_k$ get driven to zero (Tibshirani 1996; Bishop 2006; Murphy 2012; Deisenroth, Faisal, and Ong 2020). This results in a *sparse* set of basis functions being used in the interpolant, which can greatly speed up the computation of new values of $\mathbf{f}(\mathbf{x})$. We will have more to say on sparse kernel techniques in the section on Support Vector Machines (Section 5.1.4).

An alternative to solving a set of equations to determine the weights $\mathbf{w}_k$ is to simply set them to the input data values $\mathbf{d}_k$. However, this fails to interpolate the data, and instead

produces higher values in higher density regions. This can be useful if we are trying to estimate a probability density function from a set of samples. In this case, the resulting density function, obtained after normalizing the sum of sample-weighted basis functions to have a unit integral, is called the *Parzen window* or *kernel* approach to probability density estimation (Duda, Hart, and Stork 2001, Section 4.3; Bishop 2006, Section 2.5.1). Such probability densities can be used, among other things, for (spatially) clustering color values together for image segmentation in what is known as the *mean shift* approach (Comaniciu and Meer 2002) (Section 7.5.2).

If, instead of just estimating a density, we wish to actually interpolate a set of data values $\mathbf{d}_k$, we can use a related technique known as *kernel regression* or the *Nadaraya-Watson* model, in which we divide the data-weighted summed basis functions by the sum of all the basis functions,

$$\mathbf{f}(\mathbf{x}) = \frac{\sum_k \mathbf{d}_k \phi(\|\mathbf{x} - \mathbf{x}_k\|)}{\sum_l \phi(\|\mathbf{x} - \mathbf{x}_l\|)}. \quad (4.12)$$

Note how this operation is similar, in concept, to the *splatting* method for forward rendering we discussed in Section 3.6.1, except that here, the bases can be much wider than the nearest-neighbor bilinear bases used in graphics (Takeda, Farsiu, and Milanfar 2007).

Kernel regression is equivalent to creating a new set of spatially varying normalized shifted basis functions

$$\phi'_k(\mathbf{x}) = \frac{\phi(\|\mathbf{x} - \mathbf{x}_k\|)}{\sum_l \phi(\|\mathbf{x} - \mathbf{x}_l\|)}, \quad (4.13)$$

which form a *partition of unity*, i.e., sum up to 1 at every location (Anjyo, Lewis, and Pighin 2014). While the resulting interpolant can now be written more succinctly as

$$\mathbf{f}(\mathbf{x}) = \sum_k \mathbf{d}_k \phi'_k(\|\mathbf{x} - \mathbf{x}_k\|), \quad (4.14)$$

in most cases, it is more expensive to precompute and store the K ϕ'_k functions than to evaluate (4.12).

While not that widely used in computer vision, kernel regression techniques have been applied by Takeda, Farsiu, and Milanfar (2007) to a number of low-level image processing operations, including state-of-the-art handheld multi-frame super-resolution (Wronski, Garcia-Dorado *et al.* 2019).

One last scattered data interpolation technique worth mentioning is *moving least squares*, where a weighted subset of nearby points is used to compute a local smooth surface. Such techniques are mostly widely used in 3D computer graphics, especially for point-based surface modeling, as discussed in Section 13.4 and (Alexa, Behr *et al.* 2003; Pauly, Keiser *et al.* 2003; Anjyo, Lewis, and Pighin 2014).

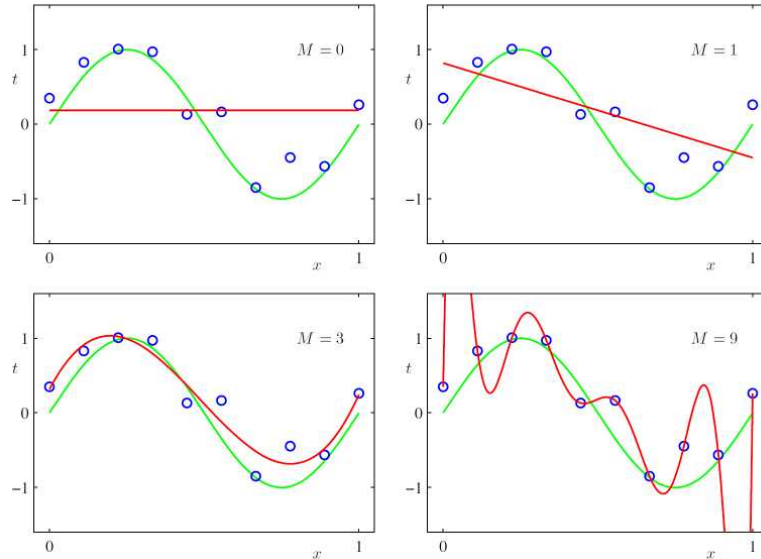


Figure 4.3 Polynomial curve fitting to the blue circles, which are noisy samples from the green sine curve (Bishop 2006) © 2006 Springer. The four plots show the 0th order constant function, the first order linear fit, the $M = 3$ cubic polynomial, and the 9th degree polynomial. Notice how the first two curves exhibit underfitting, while the last curve exhibits overfitting, i.e., excessive wiggle.

4.1.2 Overfitting and underfitting

When we introduced weight regularization in (4.9), we said that it was usually preferable to approximate the data but we did not explain why. In most data fitting problems, the samples $\mathbf{d}_k$ (and sometimes even their locations $\mathbf{x}_k$) are noisy, so that fitting them exactly makes no sense. In fact, doing so can introduce a lot of spurious wiggles, when the true solution is likely to be smoother.

To delve into this phenomenon, let us start with a simple polynomial fitting example taken from (Bishop 2006, Chapter 1.1). Figure 4.3 shows a number of polynomial curves of different orders M fit to the blue circles, which are noisy samples from the underlying green sine curve. Notice how the low-order ($M = 0$ and $M = 1$) polynomials severely *underfit* the underlying data, resulting in curves that are too flat, while the $M = 9$ polynomial, which exactly fits the data, exhibits far more wiggle than is likely.

How can we quantify this amount of underfitting and overfitting, and how can we get just the right amount? This topic is widely studied in machine learning and covered in a number of

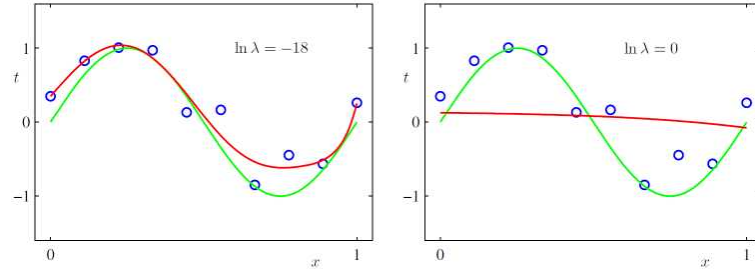


Figure 4.4 Regularized $M = 9$ polynomial fitting for two different values of λ (Bishop 2006) © 2006 Springer. The left plot shows a reasonable amount of regularization, resulting in a plausible fit, while the larger value of λ on the right causes underfitting.

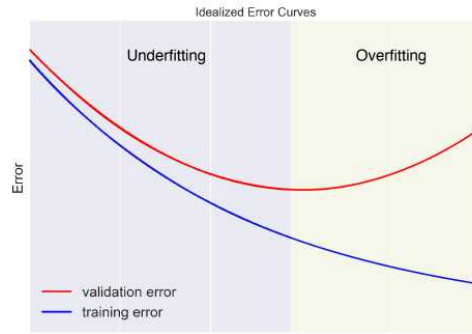


Figure 4.5 Fitting (training) and validation errors as a function of the amount of regularization or smoothing © Glassner (2018). The less regularized solutions on the right, while exhibiting lower fitting error, perform less well on the validation data.

texts, including Bishop (2006, Chapter 1.1), Glassner (2018, Chapter 9), Deisenroth, Faisal, and Ong (2020, Chapter 8), and Zhang, Lipton *et al.* (2021, Section 4.4.3).

One approach is to use regularized least squares, introduced in (4.11). Figure 4.4 shows an $M = 9$ th degree polynomial fit obtained by minimizing (4.11) with the polynomial basis functions $\phi_k(x) = x^k$ for two different values of λ . The left plot shows a reasonable amount of regularization, resulting in a plausible fit, while the larger value of λ on the right causes underfitting. Note that the $M = 9$ interpolant shown in the lower right quadrant of Figure 4.3 corresponds to the unregularized $\lambda = 0$ case.

If we were to now measure the difference between the red (estimated) and green (noise-free) curves, we see that choosing a good intermediate value of λ will produce the best result.

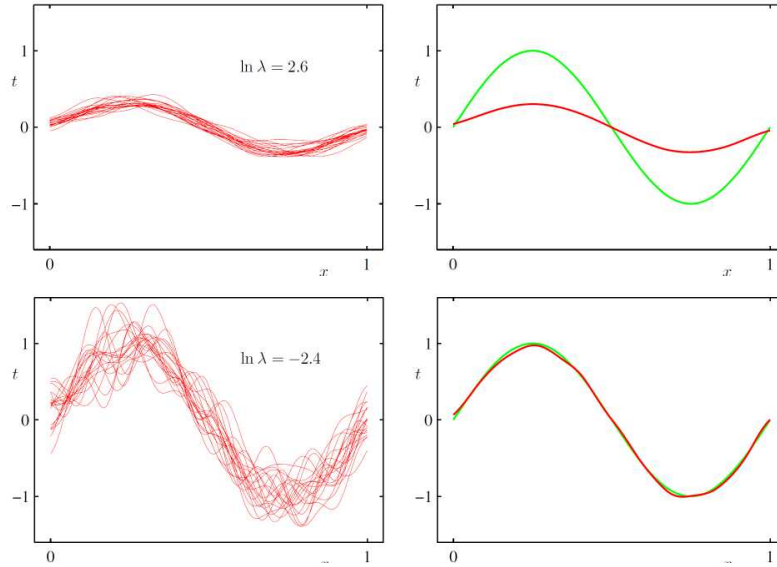


Figure 4.6 The more heavily regularized solution $\log \lambda = 2.6$ exhibits higher bias (deviation from original curve) than the less heavily regularized version ($\log \lambda = -2.4$), which has much higher variance (Bishop 2006) © 2006 Springer. The red curves on the left are $M = 24$ Gaussian basis fits to 25 randomly sampled points on the green curve. The red curve on the right is their mean.

In practice, however, we never have access to samples from the noise-free data.

Instead, if we are given a set of samples to interpolate, we can save some in a *validation* set in order to see if the function we compute is underfitting or overfitting. When we vary a parameter such as λ (or use some other measure to control smoothness), we typically obtain a curve such as the one shown in Figure 4.5. In this figure, the blue curve denotes the fitting error, which in this case is called the *training error*, since in machine learning, we usually split the given data into a (typically larger) training set and a (typically smaller) validation set.

To obtain an even better estimate of the ideal amount of regularization, we can repeat the process of splitting our sample data into training and validation sets several times. One well-known technique, called *cross-validation* (Craven and Wahba 1979; Wahba and Wendelberger 1980; Bishop 2006, Section 1.3; Murphy 2012, Section 1.4.8; Deisenroth, Faisal, and Ong 2020, Chapter 8; Zhang, Lipton *et al.* 2021, Section 4.4.2), splits the training data into K *folds* (equal sized pieces). You then put aside each fold, in turn, and train on the remaining

data. You can then estimate the best regularization parameter by averaging over all K training runs. While this generally works well ($K = 5$ is often used), it may be too expensive when training large neural networks because of the long training times involved.

Cross-validation is just one example of a class of *model selection* techniques that estimate *hyperparameters* in a training algorithm to achieve good performance. Additional methods include *information criteria* such as the Bayesian information criterion (BIC) (Torr 2002) and the Akaike information criterion (AIC) (Kanatani 1998), and Bayesian modeling approaches (Szeliski 1989; Bishop 2006; Murphy 2012).

One last topic worth mention with regard to data fitting, since it comes up often in discussions of statistical machine learning techniques, is the *bias-variance tradeoff* (Bishop 2006, Section 3.2). As you can see in Figure 4.6, using a large amount of regularization (top row) results in much lower variance between different random sample solutions, but much higher bias away from the true solution. Using insufficient regularization increases the variance dramatically, although an average over a large number of samples has low bias. The trick is to determine a reasonable compromise in terms of regularization so that any individual solution has a good expectation of being close to the *ground truth* (original clean continuous) data.

4.1.3 Robust data fitting

When we added a regularizer on the weights in (4.9), we noted that it did not have to be a quadratic penalty and could, instead, be a lower-order monomial that encouraged *sparsity* in the weights.

This same idea can be applied to data terms such as (4.2), where, instead of using a quadratic penalty, we can use a *robust loss function* $\rho()$,

$$E_R = \sum_k \rho(\|\mathbf{r}_k\|), \quad \text{with } \mathbf{r}_k = \mathbf{f}(\mathbf{x}_k) - \mathbf{d}_k, \quad (4.15)$$

which gives lower weights to larger data fitting errors, which are more likely to be outlier measurements. (The fitting error term $\mathbf{r}_k$ is called the *residual error*.)

Some examples of loss functions from (Barron 2019) are shown in Figure 4.7 along with their derivatives. The regular quadratic ($\alpha = 2$) penalty gives full (linear) weight to each error, whereas the $\alpha = 1$ loss gives equal weight to all larger residuals, i.e., it behaves as an L_1 loss for large residuals, and L_2 for small ones. Even larger values of α discount large errors (outliers) even more, although they result in optimization problems that are *non-convex*, i.e., that can have multiple local minima. We will discuss techniques for finding good initial guesses for such problems later on in Section 8.1.4.

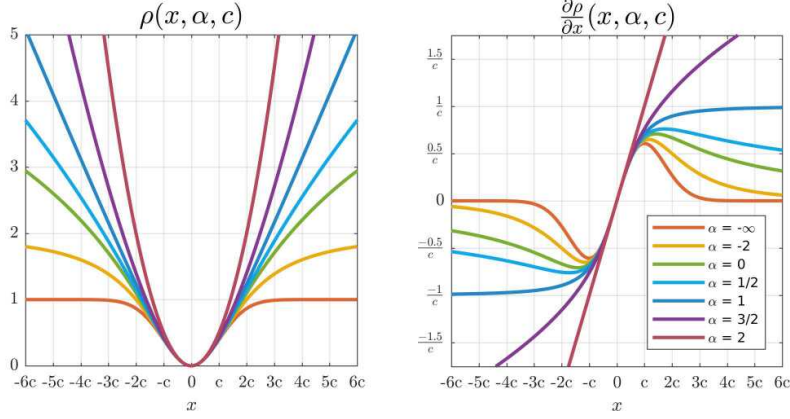


Figure 4.7 A general and adaptive loss function (left) and its gradient (right) for different values of its shape parameter α (Barron 2019) © 2019 IEEE. Several values of α reproduce existing loss functions: L_2 loss ($\alpha = 2$), Charbonnier loss ($\alpha = 1$), Cauchy loss ($\alpha = 0$), Geman-McClure loss ($\alpha = -2$), and Welsch loss ($\alpha = -1$).

In statistics, minimizing non-quadratic loss functions to deal with potential outlier measurements is known as *M-estimation* (Huber 1981; Hampel, Ronchetti *et al.* 1986; Black and Rangarajan 1996; Stewart 1999). Such estimation problems are often solved using *iteratively reweighted least squares*, which we discuss in more detail in Section 8.1.4 and Appendix B.3. The Appendix also discusses the relationship between robust statistics and non-Gaussian probabilistic models.

The generalized loss function introduced by Barron (2019) has two free parameters. The first one, α , controls how drastically outlier residuals are downweighted. The second (scale) parameter c controls the width of the quadratic well near the minimum, i.e., what range of residual values roughly corresponds to inliers. Traditionally, the choice of α , which corresponds to a variety of previously published loss functions, was determined heuristically, based on the expected shape of the outlier distribution and computational considerations (e.g., whether a convex loss was desired). The scale parameter c could be estimated using a robust measure of variance, as discussed in Appendix B.3.

In his paper, Barron (2019) discusses how both parameters can be determined at run time by maximizing the likelihood (or equivalently, minimizing the negative log-likelihood) of the given residuals, making such an algorithm self-tuning to a wide variety of noise levels and outlier distributions.

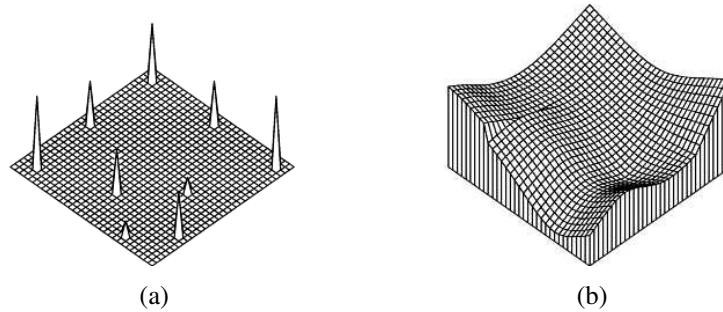


Figure 4.8 A simple surface interpolation problem: (a) nine data points of various heights scattered on a grid; (b) second-order, controlled-continuity, thin-plate spline interpolator, with a tear along its left edge and a crease along its right (Szeliski 1989) © 1989 Springer.

4.2 Variational methods and regularization

The theory of regularization we introduced in the previous section was first developed by statisticians trying to fit models to data that severely underconstrained the solution space (Tikhonov and Arsenin 1977; Engl, Hanke, and Neubauer 1996). Consider, for example, finding a smooth surface that passes through (or near) a set of measured data points (Figure 4.8). Such a problem is described as *ill-posed* because many possible surfaces can fit this data. Since small changes in the input can sometimes lead to large changes in the fit (e.g., if we use polynomial interpolation), such problems are also often *ill-conditioned*. Since we are trying to recover the unknown function $f(x, y)$ from which the data points $d(x_i, y_i)$ were sampled, such problems are also often called *inverse problems*. Many computer vision tasks can be viewed as inverse problems, since we are trying to recover a full description of the 3D world from a limited set of images.

In the previous section, we attacked this problem using basis functions placed at the data points, or other heuristics such as the pull-push algorithm. While such techniques can provide reasonable solutions, they do not let us directly *quantify* and hence *optimize* the amount of smoothness in the solution, nor do they give us local control over where the solution should be discontinuous (Figure 4.8).

To do this, we use norms (measures) on function derivatives (described below) to formulate the problem and then find minimal energy solutions to these norms. Such techniques are often called *energy-based* or *optimization-based* approaches to computer vision. They are also often called *variational*, since we can use the *calculus of variations* to find the optimal solutions. Variational methods have been widely used in computer vision since the early

1980s to pose and solve a number of fundamental problems, including optical flow (Horn and Schunck 1981; Black and Anandan 1993; Brox, Bruhn *et al.* 2004; Werlberger, Pock, and Bischof 2010), segmentation (Kass, Witkin, and Terzopoulos 1988; Mumford and Shah 1989; Chan and Vese 2001), denoising (Rudin, Osher, and Fatemi 1992; Chan, Osher, and Shen 2001; Chan and Shen 2005), and multi-view stereo (Faugeras and Keriven 1998; Pons, Keriven, and Faugeras 2007; Kolev, Klodt *et al.* 2009). A more detailed list of relevant papers can be found in the Additional Reading section at the end of this chapter.

In order to quantify what it means to find a smooth solution, we can define a norm on the solution space. For one-dimensional functions $f(x)$, we can integrate the squared first derivative of the function,

$$\mathcal{E}_1 = \int f_x^2(x) dx \quad (4.16)$$

or perhaps integrate the squared second derivative,

$$\mathcal{E}_2 = \int f_{xx}^2(x) dx. \quad (4.17)$$

(Here, we use subscripts to denote differentiation.) Such energy measures are examples of *functionals*, which are operators that map functions to scalar values. They are also often called *variational methods*, because they measure the variation (non-smoothness) in a function.

In two dimensions (e.g., for images, flow fields, or surfaces), the corresponding smoothness functionals are

$$\mathcal{E}_1 = \int f_x^2(x, y) + f_y^2(x, y) dx dy = \int \|\nabla f(x, y)\|^2 dx dy \quad (4.18)$$

and

$$\mathcal{E}_2 = \int f_{xx}^2(x, y) + 2f_{xy}^2(x, y) + f_{yy}^2(x, y) dx dy, \quad (4.19)$$

where the mixed $2f_{xy}^2$ term is needed to make the measure rotationally invariant (Grimson 1983).

The first derivative norm is often called the *membrane*, since interpolating a set of data points using this measure results in a tent-like structure. (In fact, this formula is a small-deflection approximation to the surface area, which is what soap bubbles minimize.) The second-order norm is called the *thin-plate spline*, since it approximates the behavior of thin plates (e.g., flexible steel) under small deformations. A blend of the two is called the *thin-plate spline under tension* (Terzopoulos 1986b).

The regularizers (smoothness functions) we have just described force the solution to be smooth and C_0 and/or C_1 continuous everywhere. In most computer vision applications, however, the fields we are trying to model or recover are only piecewise continuous, e.g.,

depth maps and optical flow fields jump at object discontinuities. Color images are even more discontinuous, since they also change appearance at albedo (surface color) and shading discontinuities.

To better model such functions, Terzopoulos (1986b) introduced *controlled-continuity splines*, where each derivative term is multiplied by a local weighting function,

$$\begin{aligned} \mathcal{E}_{CC} = \int \rho(x, y) \{ [1 - \tau(x, y)] [f_x^2(x, y) + f_y^2(x, y)] \\ + \tau(x, y) [f_{xx}^2(x, y) + 2f_{xy}^2(x, y) + f_{yy}^2(x, y)] \} dx dy. \end{aligned} \quad (4.20)$$

Here, $\rho(x, y) \in [0, 1]$ controls the *continuity* of the surface and $\tau(x, y) \in [0, 1]$ controls the local *tension*, i.e., how flat the surface wants to be. Figure 4.8 shows a simple example of a controlled-continuity interpolator fit to nine scattered data points. In practice, it is more common to find first-order smoothness terms used with images and flow fields (Section 9.3) and second-order smoothness associated with surfaces (Section 13.3.1).

In addition to the smoothness term, variational problems also require a data term (or *data penalty*). For scattered data interpolation (Nielson 1993), the data term measures the distance between the function $f(x, y)$ and a set of data points $d_i = d(x_i, y_i)$,

$$\mathcal{E}_D = \sum_i [f(x_i, y_i) - d_i]^2. \quad (4.21)$$

For a problem like noise removal, a continuous version of this measure can be used,

$$\mathcal{E}_D = \int [f(x, y) - d(x, y)]^2 dx dy. \quad (4.22)$$

To obtain a global energy that can be minimized, the two energy terms are usually added together,

$$\mathcal{E} = \mathcal{E}_D + \lambda \mathcal{E}_S, \quad (4.23)$$

where $\mathcal{E}_S$ is the *smoothness penalty* ($\mathcal{E}_1$, $\mathcal{E}_2$ or some weighted blend such as $\mathcal{E}_{CC}$) and λ is the *regularization parameter*, which controls the smoothness of the solution. As we saw in Section 4.1.2, good values for the regularization parameter can be estimated using techniques such as cross-validation.

4.2.1 Discrete energy minimization

In order to find the minimum of this continuous problem, the function $f(x, y)$ is usually first discretized on a regular grid.² The most principled way to perform this discretization is to use

²The alternative of using *kernel basis functions* centered on the data points (Boult and Kender 1986; Nielson 1993) is discussed in more detail in Section 13.3.1.

finite element analysis, i.e., to approximate the function with a piecewise continuous spline, and then perform the analytic integration (Bathe 2007).

Fortunately, for both the first-order and second-order smoothness functionals, the judicious selection of appropriate finite elements results in particularly simple discrete forms (Terzopoulos 1983). The corresponding *discrete* smoothness energy functions become

$$E_1 = \sum_{i,j} s_x(i,j)[f(i+1,j) - f(i,j) - g_x(i,j)]^2 + s_y(i,j)[f(i,j+1) - f(i,j) - g_y(i,j)]^2 \quad (4.24)$$

and

$$E_2 = h^{-2} \sum_{i,j} c_x(i,j)[f(i+1,j) - 2f(i,j) + f(i-1,j)]^2 + 2c_m(i,j)[f(i+1,j+1) - f(i+1,j) - f(i,j+1) + f(i,j)]^2 + c_y(i,j)[f(i,j+1) - 2f(i,j) + f(i,j-1)]^2, \quad (4.25)$$

where h is the size of the finite element grid. The h factor is only important if the energy is being discretized at a variety of resolutions, as in coarse-to-fine or multigrid techniques.

The optional smoothness weights $s_x(i,j)$ and $s_y(i,j)$ control the location of horizontal and vertical tears (or weaknesses) in the surface. For other problems, such as colorization (Levin, Lischinski, and Weiss 2004) and interactive tone mapping (Lischinski, Farbman *et al.* 2006), they control the smoothness in the interpolated chroma or exposure field and are often set inversely proportional to the local luminance gradient strength. For second-order problems, the crease variables $c_x(i,j)$, $c_m(i,j)$, and $c_y(i,j)$ control the locations of creases in the surface (Terzopoulos 1988; Szeliski 1990a).

The data values $g_x(i,j)$ and $g_y(i,j)$ are gradient data terms (constraints) used by algorithms, such as photometric stereo (Section 13.1.1), HDR tone mapping (Section 10.2.1) (Fattal, Lischinski, and Werman 2002), Poisson blending (Section 8.4.4) (Pérez, Gangnet, and Blake 2003), gradient-domain blending (Section 8.4.4) (Levin, Zomet *et al.* 2004), and Poisson surface reconstruction (Section 13.5.1) (Kazhdan, Bolitho, and Hoppe 2006; Kazhdan and Hoppe 2013). They are set to zero when just discretizing the conventional first-order smoothness functional (4.18). Note how separate smoothness and curvature terms can be imposed in the x , y , and mixed directions to produce local tears or creases (Terzopoulos 1988; Szeliski 1990a).

The two-dimensional discrete data energy is written as

$$E_D = \sum_{i,j} c(i,j)[f(i,j) - d(i,j)]^2, \quad (4.26)$$

where the local confidence weights $c(i, j)$ control how strongly the data constraint is enforced. These values are set to zero where there is no data and can be set to the inverse variance of the data measurements when there is data (as discussed by Szeliski (1989) and in Section 4.3).

The total energy of the discretized problem can now be written as a *quadratic form*

$$E = E_D + \lambda E_S = \mathbf{x}^T \mathbf{A} \mathbf{x} - 2 \mathbf{x}^T \mathbf{b} + c, \quad (4.27)$$

where $\mathbf{x} = [f(0, 0) \dots f(m-1, n-1)]$ is called the *state vector*.³

The sparse symmetric positive-definite matrix $\mathbf{A}$ is called the *Hessian* since it encodes the second derivative of the energy function.⁴ For the one-dimensional, first-order problem, $\mathbf{A}$ is tridiagonal; for the two-dimensional, first-order problem, it is multi-banded with five non-zero entries per row. We call $\mathbf{b}$ the *weighted data vector*. Minimizing the above quadratic form is equivalent to solving the sparse linear system

$$\mathbf{A} \mathbf{x} = \mathbf{b}, \quad (4.28)$$

which can be done using a variety of sparse matrix techniques, such as multigrid (Briggs, Henson, and McCormick 2000) and hierarchical preconditioners (Szeliski 2006b; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013), as described in Appendix A.5 and illustrated in Figure 4.11. Using such techniques is essential to obtaining reasonable run-times, since properly preconditioned sparse linear systems have convergence times that are *linear* in the number of pixels.

While regularization was first introduced to the vision community by Poggio, Torre, and Koch (1985) and Terzopoulos (1986b) for problems such as surface interpolation, it was quickly adopted by other vision researchers for such varied problems as edge detection (Section 7.2), optical flow (Section 9.3), and shape from shading (Section 13.1) (Poggio, Torre, and Koch 1985; Horn and Brooks 1986; Terzopoulos 1986b; Bertero, Poggio, and Torre 1988; Brox, Bruhn *et al.* 2004). Poggio, Torre, and Koch (1985) also showed how the discrete energy defined by Equations (4.24–4.26) could be implemented in a resistive grid, as shown in Figure 4.9. In computational photography (Chapter 10), regularization and its variants are commonly used to solve problems such as high-dynamic range tone mapping (Fattal, Lischinski, and Werman 2002; Lischinski, Farbman *et al.* 2006), Poisson and gradient-domain blending (Pérez, Gangnet, and Blake 2003; Levin, Zomet *et al.* 2004; Agarwala, Dontcheva *et al.*

³We use $\mathbf{x}$ instead of $\mathbf{f}$ because this is the more common form in the numerical analysis literature (Golub and Van Loan 1996).

⁴In numerical analysis, $\mathbf{A}$ is called the *coefficient* matrix (Saad 2003); in finite element analysis (Bathe 2007), it is called the *stiffness* matrix.

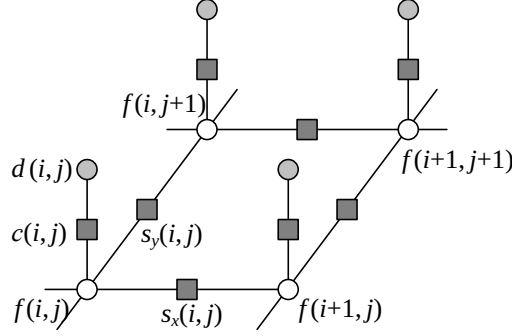


Figure 4.9 Graphical model interpretation of first-order regularization. The white circles are the unknowns $f(i, j)$ while the dark circles are the input data $d(i, j)$. In the resistive grid interpretation, the d and f values encode input and output voltages and the black squares denote resistors whose conductance is set to $s_x(i, j)$, $s_y(i, j)$, and $c(i, j)$. In the spring-mass system analogy, the circles denote elevations and the black squares denote springs. The same graphical model can be used to depict a first-order Markov random field (Figure 4.12).

2004), colorization (Levin, Lischinski, and Weiss 2004), and natural image matting (Levin, Lischinski, and Weiss 2008).

Robust regularization

While regularization is most commonly formulated using quadratic (L_2) norms, i.e., the squared derivatives in (4.16–4.19) and squared differences in (4.24–4.25), it can also be formulated using the non-quadratic *robust* penalty functions first introduced in Section 4.1.3 and discussed in more detail in Appendix B.3. For example, (4.24) can be generalized to

$$E_{1R} = \sum_{i,j} s_x(i, j) \rho(f(i+1, j) - f(i, j)) + s_y(i, j) \rho(f(i, j+1) - f(i, j)), \quad (4.29)$$

where $\rho(x)$ is some monotonically increasing penalty function. For example, the family of norms $\rho(x) = |x|^p$ is called p -norms. When $p < 2$, the resulting smoothness terms become more piecewise continuous than totally smooth, which can better model the discontinuous nature of images, low fields, and 3D surfaces.

An early example of robust regularization is the *graduated non-convexity* (GNC) algorithm of Blake and Zisserman (1987). Here, the norms on the data and derivatives are

clamped,

$$\rho(x) = \min(x^2, V). \quad (4.30)$$

Because the resulting problem is highly non-convex (it has many local minima), a *continuation* method is proposed, where a quadratic norm (which is convex) is gradually replaced by the non-convex robust norm (Allgower and Georg 2003). (Around the same time, Terzopoulos (1988) was also using continuation to infer the tear and crease variables in his surface interpolation problems.)

4.2.2 Total variation

Today, many regularized problems are formulated using the L_1 ($p = 1$) norm, which is often called *total variation* (Rudin, Osher, and Fatemi 1992; Chan, Osher, and Shen 2001; Chambolle 2004; Chan and Shen 2005; Tschumperlé and Deriche 2005; Tschumperlé 2006; Cremers 2007; Kaftory, Schechner, and Zeevi 2007; Kolev, Klodt *et al.* 2009; Werlberger, Pock, and Bischof 2010). The advantage of this norm is that it tends to better preserve discontinuities, but still results in a convex problem that has a globally unique solution. Other norms, for which the *influence* (derivative) more quickly decays to zero, are presented by Black and Rangarajan (1996), Black, Sapiro *et al.* (1998), and Barron (2019) and discussed in Section 4.1.3 and Appendix B.3.

Even more recently, *hyper-Laplacian* norms with $p < 1$ have gained popularity, based on the observation that the log-likelihood distribution of image derivatives follows a $p \approx 0.5 - 0.8$ slope and is therefore a hyper-Laplacian distribution (Simoncelli 1999; Levin and Weiss 2007; Weiss and Freeman 2007; Krishnan and Fergus 2009). Such norms have an even stronger tendency to prefer large discontinuities over small ones. See the related discussion in Section 4.3 (4.43).

While least squares regularized problems using L_2 norms can be solved using linear systems, other p -norms require different iterative techniques, such as iteratively reweighted least squares (IRLS), Levenberg–Marquardt, alternation between local non-linear subproblems and global quadratic regularization (Krishnan and Fergus 2009), or primal-dual algorithms (Chambolle and Pock 2011). Such techniques are discussed in Section 8.1.3 and Appendices A.3 and B.3.

4.2.3 Bilateral solver

In our discussion of variational methods, we have focused on energy minimization problems based on gradients and higher-order derivatives, which in the discrete setting involves

evaluating weighted errors between neighboring pixels. As we saw previously in our discussion of bilateral filtering in Section 3.3.2, we can often get better results by looking at a larger spatial neighborhood and combining pixels with similar colors or grayscale values. To extend this idea to a variational (energy minimization) setting, Barron and Poole (2016) propose replacing the usual first-order nearest-neighbor smoothness penalty (4.24) with a wider-neighborhood, bilaterally weighted version

$$E_B = \sum_{i,j} \sum_{k,l} \hat{w}(i,j,k,l) [f(k,l) - f(i,j)]^2, \quad (4.31)$$

where

$$\hat{w}(i,j,k,l) = \frac{w(i,j,k,l)}{\sum_{m,n} w(i,j,m,n)}, \quad (4.32)$$

is the *bistochastized* (normalized) version of the *bilateral weight function* given in (3.37), which may depend on an input guide image, but not on the estimated values of f .⁵

To efficiently solve the resulting set of equations (which are much denser than nearest-neighbor versions), the authors use the same approach originally used to accelerate bilateral filtering, i.e., solving a related problem on a (spatially coarser) bilateral grid. The sequence of operations resembles those used for bilateral filtering, except that after splatting and before slicing, an iterative least squares solver is used instead of a multi-dimensional Gaussian blur. To further speed up the conjugate gradient solver, Barron and Poole (2016) use a multi-level preconditioner inspired by previous work on image-adapted preconditioners (Szeliski 2006b; Krishnan, Fattal, and Szeliski 2013).

Since its introduction, the bilateral solver has been used in a number of video processing and 3D reconstruction applications, including the stitching of binocular omnidirectional panoramic videos (Anderson, Gallup *et al.* 2016). The smartphone AR system developed by Valentin, Kowdle *et al.* (2018) extends the bilateral solver to have local *planar* models and uses a hardware-friendly real-time implementation (Mazumdar, Alaghi *et al.* 2017) to produce dense occlusion effects.

4.2.4 Application: Interactive colorization

A good use of edge-aware interpolation techniques is in *colorization*, i.e., manually adding colors to a “black and white” (grayscale) image. In most applications of colorization, the user draws some scribbles indicating the desired colors in certain regions (Figure 4.10a) and the system interpolates the specified chrominance (u, v) values to the whole image, which

⁵Note that in their paper, Barron and Poole (2016) use different σ_r values for the luminance and chrominance components of pixel color differences.

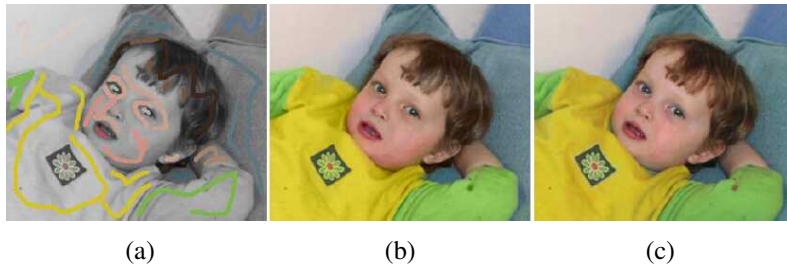


Figure 4.10 Colorization using optimization (Levin, Lischinski, and Weiss 2004) © 2004 ACM: (a) grayscale image with some color scribbles overlaid; (b) resulting colorized image; (c) original color image from which the grayscale image and the chrominance values for the scribbles were derived. Original photograph by Rotem Weiss.

are then re-combined with the input luminance channel to produce a final colorized image, as shown in Figure 4.10b. In the system developed by Levin, Lischinski, and Weiss (2004), the interpolation is performed using locally weighted regularization (4.24), where the local smoothness weights are inversely proportional to luminance gradients. This approach to locally weighted regularization has inspired later algorithms for high dynamic range tone mapping (Lischinski, Farbman *et al.* 2006)(Section 10.2.1, as well as other applications of the weighted least squares (WLS) formulation (Farbman, Fattal *et al.* 2008). These techniques have benefitted greatly from image-adapted regularization techniques, such as those developed in Szeliski (2006b), Krishnan and Szeliski (2011), Krishnan, Fattal, and Szeliski (2013), and Barron and Poole (2016), as shown in Figure 4.11. An alternative approach to performing the sparse chrominance interpolation based on geodesic (edge-aware) distance functions has been developed by Yatziv and Sapiro (2006). Neural networks can also be used to implement *deep priors* for image colorization (Zhang, Zhu *et al.* 2017).

4.3 Markov random fields

As we have just seen, regularization, which involves the minimization of energy functionals defined over (piecewise) continuous functions, can be used to formulate and solve a variety of low-level computer vision problems. An alternative technique is to formulate a *Bayesian* or *generative* model, which separately models the noisy image formation (*measurement*) process, as well as assuming a statistical *prior* model over the solution space (Bishop 2006, Section 1.5.4). In this section, we look at priors based on Markov random fields, whose log-likelihood can be described using local neighborhood interaction (or penalty) terms (Kin-



Figure 4.11 *Speeding up the inhomogeneous least squares colorization solver using locally adapted hierarchical basis preconditioning (Szeliski 2006b) © 2006 ACM: (a) input gray image with color strokes overlaid; (b) solution after 20 iterations of conjugate gradient; (c) using one iteration of hierarchical basis function preconditioning; (d) using one iteration of locally adapted hierarchical basis functions.*

dermann and Snell 1980; Geman and Geman 1984; Marroquin, Mitter, and Poggio 1987; Li 1995; Szeliski, Zabih *et al.* 2008; Blake, Kohli, and Rother 2011).

The use of Bayesian modeling has several potential advantages over regularization (see also Appendix B). The ability to model measurement processes statistically enables us to extract the maximum information possible from each measurement, rather than just guessing what weighting to give the data. Similarly, the parameters of the prior distribution can often be *learned* by observing samples from the class we are modeling (Roth and Black 2007a; Tappen 2007; Li and Huttenlocher 2008). Furthermore, because our model is probabilistic, it is possible to estimate (in principle) complete probability *distributions* over the unknowns being recovered and, in particular, to model the *uncertainty* in the solution, which can be useful in later processing stages. Finally, Markov random field models can be defined over *discrete* variables, such as image labels (where the variables have no proper ordering), for which regularization does not apply.

According to Bayes' rule (Appendix B.4), the *posterior* distribution $p(\mathbf{x}|\mathbf{y})$ over the unknowns $\mathbf{x}$ given the measurements $\mathbf{y}$ can be obtained by multiplying the measurement likelihood $p(\mathbf{y}|\mathbf{x})$ by the prior distribution $p(\mathbf{x})$ and normalizing,

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{y})}, \quad (4.33)$$

where $p(\mathbf{y}) = \int_{\mathbf{x}} p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$ is a normalizing constant used to make the $p(\mathbf{x}|\mathbf{y})$ distribution *proper* (integrate to 1). Taking the negative logarithm of both sides of (4.33), we get

$$-\log p(\mathbf{x}|\mathbf{y}) = -\log p(\mathbf{y}|\mathbf{x}) - \log p(\mathbf{x}) + C, \quad (4.34)$$

which is the *negative posterior log likelihood*.

To find the most likely (*maximum a posteriori* or MAP) solution $\mathbf{x}$ given some measurements $\mathbf{y}$, we simply minimize this negative log likelihood, which can also be thought of as an *energy*,

$$E(\mathbf{x}, \mathbf{y}) = E_D(\mathbf{x}, \mathbf{y}) + E_P(\mathbf{x}). \quad (4.35)$$

(We drop the constant C because its value does not matter during energy minimization.) The first term $E_D(\mathbf{x}, \mathbf{y})$ is the *data energy* or *data penalty*; it measures the negative log likelihood that the data were observed given the unknown state $\mathbf{x}$. The second term $E_P(\mathbf{x})$ is the *prior energy*; it plays a role analogous to the smoothness energy in regularization. Note that the MAP estimate may not always be desirable, as it selects the “peak” in the posterior distribution rather than some more stable statistic—see the discussion in Appendix B.2 and by Levin, Weiss *et al.* (2009).

For the remainder of this section, we focus on Markov random fields, which are probabilistic models defined over two or three-dimensional pixel or voxel grids. Before we dive into this, however, we should mention that MRFs are just one special case of the more general family of *graphical models* (Bishop 2006, Chapter 8; Koller and Friedman 2009; Nowozin and Lampert 2011; Murphy 2012, Chapters 10, 17, 19), which have sparse interactions between variables that can be captured in a *factor graph* (Dellaert and Kaess 2017; Dellaert 2021), such as the one shown in Figure 4.12. Graphical models come in a wide variety of topologies, including chains (used for audio and speech processing), trees (often used for modeling kinematic chains in tracking people (e.g., Felzenszwalb and Huttenlocher 2005)), stars (simplified models for people; Dalal and Triggs 2005; Felzenszwalb, Girshick *et al.* 2010, and constellations (Fergus, Perona, and Zisserman 2007). Such models were widely used for part-based recognition, as discussed in Section 6.2.1. For graphs that are acyclic, efficient linear-time inference algorithms based on dynamic programming can be used.

For image processing applications, the unknowns $\mathbf{x}$ are the set of output pixels

$$\mathbf{x} = [f(0, 0) \dots f(m - 1, n - 1)], \quad (4.36)$$

and the data are (in the simplest case) the input pixels

$$\mathbf{y} = [d(0, 0) \dots d(m - 1, n - 1)] \quad (4.37)$$

as shown in Figure 4.12.

For a Markov random field, the probability $p(\mathbf{x})$ is a *Gibbs* or *Boltzmann distribution*, whose negative log likelihood (according to the Hammersley–Clifford theorem) can be written as a sum of pairwise interaction potentials,

$$E_P(\mathbf{x}) = \sum_{\{(i,j),(k,l)\} \in \mathcal{N}(i,j)} V_{i,j,k,l}(f(i,j), f(k,l)), \quad (4.38)$$

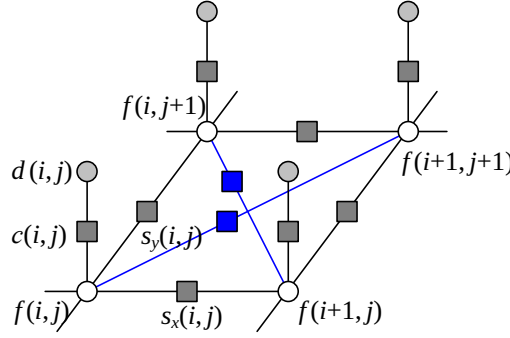


Figure 4.12 Graphical model for an $\mathcal{N}_4$ neighborhood Markov random field. (The blue edges are added for an $\mathcal{N}_8$ neighborhood.) The white circles are the unknowns $f(i, j)$, while the dark circles are the input data $d(i, j)$. The $s_x(i, j)$ and $s_y(i, j)$ black boxes denote arbitrary interaction potentials between adjacent nodes in the random field, and the $c(i, j)$ denote the data penalty functions. The same graphical model can be used to depict a discrete version of a first-order regularization problem (Figure 4.9).

where $\mathcal{N}(i, j)$ denotes the *neighbors* of pixel (i, j) . In fact, the general version of the theorem says that the energy may have to be evaluated over a larger set of *cliques*, which depend on the *order* of the Markov random field (Kindermann and Snell 1980; Geman and Geman 1984; Bishop 2006; Kohli, Ladický, and Torr 2009; Kohli, Kumar, and Torr 2009).

The most commonly used neighborhood in Markov random field modeling is the $\mathcal{N}_4$ neighborhood, where each pixel in the field $f(i, j)$ interacts only with its immediate neighbors. The model in Figure 4.12, which we previously used in Figure 4.9 to illustrate the discrete version of first-order regularization, shows an $\mathcal{N}_4$ MRF. The $s_x(i, j)$ and $s_y(i, j)$ black boxes denote arbitrary *interaction potentials* between adjacent nodes in the random field and the $c(i, j)$ denote the data penalty functions. These square nodes can also be interpreted as *factors* in a *factor graph* version of the (undirected) graphical model (Bishop 2006; Dellaert and Kaess 2017; Dellaert 2021), which is another name for interaction potentials. (Strictly speaking, the factors are (improper) probability functions whose product is the (un-normalized) posterior distribution.)

As we will see in (4.41–4.42), there is a close relationship between these interaction potentials and the discretized versions of regularized image restoration problems. Thus, to a first approximation, we can view energy minimization being performed when solving a regularized problem and the maximum *a posteriori* inference being performed in an MRF as equivalent.

While $\mathcal{N}_4$ neighborhoods are most commonly used, in some applications $\mathcal{N}_8$ (or even higher order) neighborhoods perform better at tasks such as image segmentation because they can better model discontinuities at different orientations (Boykov and Kolmogorov 2003; Rother, Kohli *et al.* 2009; Kohli, Ladický, and Torr 2009; Kohli, Kumar, and Torr 2009).

Binary MRFs

The simplest possible example of a Markov random field is a binary field. Examples of such fields include 1-bit (black and white) scanned document images as well as images segmented into foreground and background regions.

To denoise a scanned image, we set the data penalty to reflect the agreement between the scanned and final images,

$$E_D(i, j) = w\delta(f(i, j), d(i, j)) \quad (4.39)$$

and the smoothness penalty to reflect the agreement between neighboring pixels

$$E_P(i, j) = s\delta(f(i, j), f(i + 1, j)) + s\delta(f(i, j), f(i, j + 1)). \quad (4.40)$$

Once we have formulated the energy, how do we minimize it? The simplest approach is to perform gradient descent, flipping one state at a time if it produces a lower energy. This approach is known as *contextual classification* (Kittler and Föglein 1984), *iterated conditional modes* (ICM) (Besag 1986), or *highest confidence first* (HCF) (Chou and Brown 1990) if the pixel with the largest energy decrease is selected first.

Unfortunately, these downhill methods tend to get easily stuck in local minima. An alternative approach is to add some randomness to the process, which is known as *stochastic gradient descent* (Metropolis, Rosenbluth *et al.* 1953; Geman and Geman 1984). When the amount of noise is decreased over time, this technique is known as *simulated annealing* (Kirkpatrick, Gelatt, and Vecchi 1983; Carnevali, Coletti, and Patarnello 1985; Wolberg and Pavlidis 1985; Swendsen and Wang 1987) and was first popularized in computer vision by Geman and Geman (1984) and later applied to stereo matching by Barnard (1989), among others.

Even this technique, however, does not perform that well (Boykov, Veksler, and Zabih 2001). For binary images, a much better technique, introduced to the computer vision community by Boykov, Veksler, and Zabih (2001) is to re-formulate the energy minimization as a *max-flow/min-cut* graph optimization problem (Greig, Porteous, and Seheult 1989). This technique has informally come to be known as *graph cuts* in the computer vision community (Boykov and Kolmogorov 2011). For simple energy functions, e.g., those where the penalty for non-identical neighboring pixels is a constant, this algorithm is guaranteed to produce the

global minimum. Kolmogorov and Zabih (2004) formally characterize the class of binary energy potentials (*regularity conditions*) for which these results hold, while newer work by Komodakis, Tziritas, and Paragios (2008) and Rother, Kolmogorov *et al.* (2007) provide good algorithms for the cases when they do not, i.e., for energy functions that are not *regular* or *sub-modular*.

In addition to the above mentioned techniques, a number of other optimization approaches have been developed for MRF energy minimization, such as (loopy) belief propagation and dynamic programming (for one-dimensional problems). These are discussed in more detail in Appendix B.5 as well as the comparative survey papers by Szeliski, Zabih *et al.* (2008) and Kappes, Andres *et al.* (2015), which have associated benchmarks and code at <https://vision.middlebury.edu/MRF> and <http://hciweb2.iwr.uni-heidelberg.de/opengm>.

Ordinal-valued MRFs

In addition to binary images, Markov random fields can be applied to ordinal-valued labels such as grayscale images or depth maps. The term “ordinal” indicates that the labels have an implied ordering, e.g., that higher values are lighter pixels. In the next section, we look at unordered labels, such as source image labels for image compositing.

In many cases, it is common to extend the binary data and smoothness prior terms as

$$E_D(i, j) = c(i, j)\rho_d(f(i, j) - d(i, j)) \quad (4.41)$$

and

$$E_P(i, j) = s_x(i, j)\rho_p(f(i, j) - f(i + 1, j)) + s_y(i, j)\rho_p(f(i, j) - f(i, j + 1)), \quad (4.42)$$

which are robust generalizations of the quadratic penalty terms (4.26) and (4.24), first introduced in (4.29). As before, the $c(i, j)$, $s_x(i, j)$, and $s_y(i, j)$ weights can be used to locally control the data weighting and the horizontal and vertical smoothness. Instead of using a quadratic penalty, however, a general monotonically increasing penalty function $\rho(\cdot)$ is used. (Different functions can be used for the data and smoothness terms.) For example, ρ_p can be a hyper-Laplacian penalty

$$\rho_p(d) = |d|^p, \quad p < 1, \quad (4.43)$$

which better encodes the distribution of gradients (mainly edges) in an image than either a quadratic or linear (total variation) penalty.⁶ Levin and Weiss (2007) use such a penalty to

⁶Note that, unlike a quadratic penalty, the sum of the horizontal and vertical derivative p -norms is not rotationally invariant. A better approach may be to locally estimate the gradient direction and to impose different norms on the perpendicular and parallel components, which Roth and Black (2007b) call a *steerable random field*.

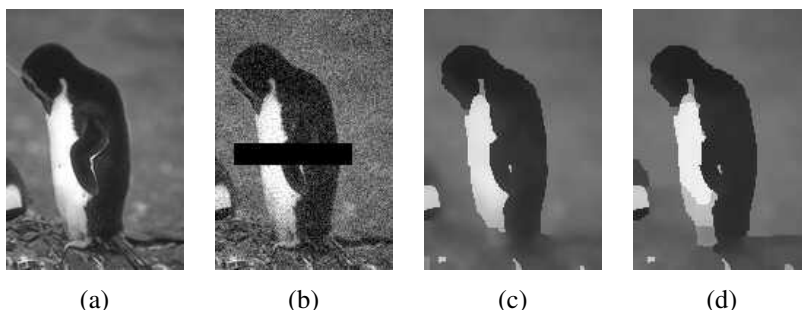


Figure 4.13 Grayscale image denoising and inpainting: (a) original image; (b) image corrupted by noise and with missing data (black bar); (c) image restored using loopy belief propagation; (d) image restored using expansion move graph cuts. Images are from <https://vision.middlebury.edu/MRF/results> (Szeliski, Zabih et al. 2008).

separate a transmitted and reflected image (Figure 9.16) by encouraging gradients to lie in one or the other image, but not both. Levin, Fergus *et al.* (2007) use the hyper-Laplacian as a prior for image deconvolution (deblurring) and Krishnan and Fergus (2009) develop a faster algorithm for solving such problems. For the data penalty, ρ_d can be quadratic (to model Gaussian noise) or the log of a *contaminated Gaussian* (Appendix B.3).

When ρ_p is a quadratic function, the resulting Markov random field is called a Gaussian Markov random field (GMRF) and its minimum can be found by sparse linear system solving (4.28). When the weighting functions are uniform, the GMRF becomes a special case of Wiener filtering (Section 3.4.1). Allowing the weighting functions to depend on the input image (a special kind of conditional random field, which we describe below) enables quite sophisticated image processing algorithms to be performed, including colorization (Levin, Lischinski, and Weiss 2004), interactive tone mapping (Lischinski, Farbman *et al.* 2006), natural image matting (Levin, Lischinski, and Weiss 2008), and image restoration (Tappen, Liu *et al.* 2007).

When ρ_d or ρ_p are non-quadratic functions, gradient descent techniques such as non-linear least squares or iteratively re-weighted least squares can sometimes be used (Appendix A.3). However, if the search space has lots of local minima, as is the case for stereo matching (Barnard 1989; Boykov, Veksler, and Zabih 2001), more sophisticated techniques are required.

The extension of graph cut techniques to multi-valued problems was first proposed by Boykov, Veksler, and Zabih (2001). In their paper, they develop two different algorithms, called the *swap move* and the *expansion move*, which iterate among a series of binary labeling

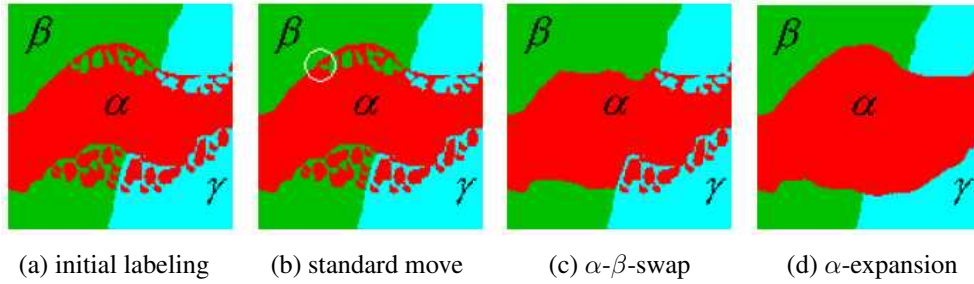


Figure 4.14 Multi-level graph optimization from Boykov, Veksler, and Zabih (2001) © 2001 IEEE: (a) initial problem configuration; (b) the standard move only changes one pixel; (c) the α - β -swap optimally exchanges all α and β -labeled pixels; (d) the α -expansion move optimally selects among current pixel values and the α label.

sub-problems to find a good solution (Figure 4.14). Note that a global solution is generally not achievable, as the problem is provably NP-hard for general energy functions. Because both these algorithms use a binary MRF optimization inside their inner loop, they are subject to the kind of constraints on the energy functions that occur in the binary labeling case (Kolmogorov and Zabih 2004).

Another MRF inference technique is *belief propagation* (BP). While belief propagation was originally developed for inference over trees, where it is exact (Pearl 1988), it has more recently been applied to graphs with loops such as Markov random fields (Freeman, Pasztor, and Carmichael 2000; Yedidia, Freeman, and Weiss 2001). In fact, some of the better performing stereo-matching algorithms use loopy belief propagation (LBP) to perform their inference (Sun, Zheng, and Shum 2003). LBP is discussed in more detail in comparative survey papers on MRF optimization (Szeliski, Zabih *et al.* 2008; Kappes, Andres *et al.* 2015).

Figure 4.13 shows an example of image denoising and inpainting (hole filling) using a non-quadratic energy function (non-Gaussian MRF). The original image has been corrupted by noise and a portion of the data has been removed (the black bar). In this case, the loopy belief propagation algorithm computes a slightly lower energy and also a smoother image than the alpha-expansion graph cut algorithm.

Of course, the above formula (4.42) for the smoothness term $E_P(i, j)$ just shows the simplest case. In follow-on work, Roth and Black (2009) propose a *Field of Experts* (FoE) model, which sums up a large number of exponentiated local filter outputs to arrive at the smoothness penalty. Weiss and Freeman (2007) analyze this approach and compare it to the simpler hyper-Laplacian model of natural image statistics. Lyu and Simoncelli (2009) use

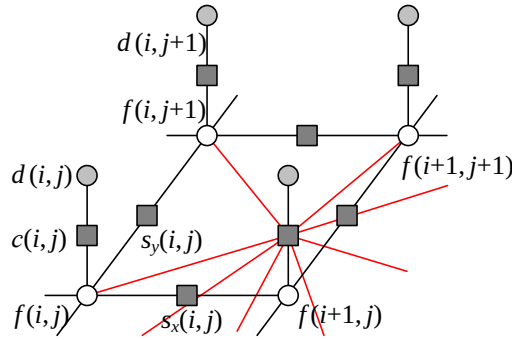


Figure 4.15 Graphical model for a Markov random field with a more complex measurement model. The additional colored edges show how combinations of unknown values (say, in a sharp image) produce the measured values (a noisy blurred image). The resulting graphical model is still a classic MRF and is just as easy to sample from, but some inference algorithms (e.g., those based on graph cuts) may not be applicable because of the increased network complexity, since state changes during the inference become more entangled and the posterior MRF has much larger cliques.

Gaussian Scale Mixtures (GSMs) to construct an inhomogeneous multi-scale MRF, with one (positive exponential) GMRF modulating the variance (amplitude) of another Gaussian MRF.

It is also possible to extend the *measurement* model to make the sampled (noise-corrupted) input pixels correspond to blends of unknown (latent) image pixels, as in Figure 4.15. This is the commonly occurring case when trying to deblur an image. While this kind of a model is still a traditional generative Markov random field, i.e., we can in principle generate random samples from the prior distribution, finding an optimal solution can be difficult because the clique sizes get larger. In such situations, gradient descent techniques, such as iteratively reweighted least squares, can be used (Joshi, Zitnick *et al.* 2009). Exercise 4.4 has you explore some of these issues.

Unordered labels

Another case with multi-valued labels where Markov random fields are often applied is that of *unordered labels*, i.e., labels where there is no semantic meaning to the numerical difference between the values of two labels. For example, if we are classifying terrain from aerial imagery, it makes no sense to take the numerical difference between the labels assigned to forest, field, water, and pavement. In fact, the adjacencies of these various kinds of terrain



Figure 4.16 An unordered label MRF (Agarwala, Dontcheva et al. 2004) © 2004 ACM: Strokes in each of the source images on the left are used as constraints on an MRF optimization, which is solved using graph cuts. The resulting multi-valued label field is shown as a color overlay in the middle image, and the final composite is shown on the right.

each have different likelihoods, so it makes more sense to use a prior of the form

$$E_P(i, j) = s_x(i, j)V(l(i, j), l(i + 1, j)) + s_y(i, j)V(l(i, j), l(i, j + 1)), \quad (4.44)$$

where $V(l_0, l_1)$ is a general *compatibility* or *potential* function. (Note that we have also replaced $f(i, j)$ with $l(i, j)$ to make it clearer that these are labels rather than function samples.) An alternative way to write this prior energy (Boykov, Veksler, and Zabih 2001; Szeliski, Zabih et al. 2008) is

$$E_P = \sum_{(p,q) \in \mathcal{N}} V_{p,q}(l_p, l_q), \quad (4.45)$$

where the (p, q) are neighboring pixels and a spatially varying potential function $V_{p,q}$ is evaluated for each neighboring pair.

An important application of unordered MRF labeling is seam finding in image compositing (Davis 1998; Agarwala, Dontcheva et al. 2004) (see Figure 4.16, which is explained in more detail in Section 8.4.2). Here, the compatibility $V_{p,q}(l_p, l_q)$ measures the quality of the visual appearance that would result from placing a pixel p from image l_p next to a pixel q from image l_q . As with most MRFs, we assume that $V_{p,q}(l, l) = 0$. For different labels, however, the compatibility $V_{p,q}(l_p, l_q)$ may depend on the values of the underlying pixels $I_{l_p}(p)$ and $I_{l_q}(q)$.

Consider, for example, where one image I_0 is all sky blue, i.e., $I_0(p) = I_0(q) = B$, while the other image I_1 has a transition from sky blue, $I_1(p) = B$, to forest green, $I_1(q) = G$.

$$I_0 : \begin{array}{|c|c|} \hline p & q \\ \hline \end{array} \quad \begin{array}{|c|c|} \hline p & q \\ \hline \end{array} : I_1$$

In this case, $V_{p,q}(1, 0) = 0$ (the colors agree), while $V_{p,q}(0, 1) > 0$ (the colors disagree).

4.3.1 Conditional random fields

In a classic Bayesian model (4.33–4.35),

$$p(\mathbf{x}|\mathbf{y}) \propto p(\mathbf{y}|\mathbf{x})p(\mathbf{x}), \quad (4.46)$$

the prior distribution $p(\mathbf{x})$ is independent of the observations $\mathbf{y}$. Sometimes, however, it is useful to modify our prior assumptions, say about the smoothness of the field we are trying to estimate, in response to the sensed data. Whether this makes sense from a probability viewpoint is something we discuss once we have explained the new model.

Consider an interactive image segmentation system such as the one described in Boykov and Funka-Lea (2006). In this application, the user draws foreground and background strokes, and the system then solves a binary MRF labeling problem to estimate the extent of the foreground object. In addition to minimizing a data term, which measures the pointwise similarity between pixel colors and the inferred region distributions (Section 4.3.2), the MRF is modified so that the smoothness terms $s_x(x, y)$ and $s_y(x, y)$ in Figure 4.12 and (4.42) depend on the magnitude of the gradient between adjacent pixels.⁷

Since the smoothness term now depends on the data, Bayes' rule (4.46) no longer applies. Instead, we use a direct model for the posterior distribution $p(\mathbf{x}|\mathbf{y})$, whose negative log likelihood can be written as

$$\begin{aligned} E(\mathbf{x}|\mathbf{y}) &= E_D(\mathbf{x}, \mathbf{y}) + E_S(\mathbf{x}, \mathbf{y}) \\ &= \sum_p V_p(x_p, \mathbf{y}) + \sum_{(p,q) \in \mathcal{N}} V_{p,q}(x_p, x_q, \mathbf{y}), \end{aligned} \quad (4.47)$$

using the notation introduced in (4.45). The resulting probability distribution is called a *conditional random field* (CRF) and was first introduced to the computer vision field by Kumar and Hebert (2003), based on earlier work in text modeling by Lafferty, McCallum, and Pereira (2001).

Figure 4.17 shows a graphical model where the smoothness terms depend on the data values. In this particular model, each smoothness term depends only on its adjacent pair of data values, i.e., terms are of the form $V_{p,q}(x_p, x_q, y_p, y_q)$ in (4.47).

The idea of modifying smoothness terms in response to input data is not new. For example, Boykov and Jolly (2001) used this idea for interactive segmentation, and it is now widely used in image segmentation (Section 4.3.2) (Blake, Rother *et al.* 2004; Rother, Kolmogorov, and Blake 2004), denoising (Tappen, Liu *et al.* 2007), and object recognition (Section 6.4) (Winn and Shotton 2006; Shotton, Winn *et al.* 2009).

⁷ An alternative formulation that also uses detected edges to modulate the smoothness of a depth or motion field and hence to integrate multiple lower level vision modules is presented by Poggio, Gamble, and Little (1988).

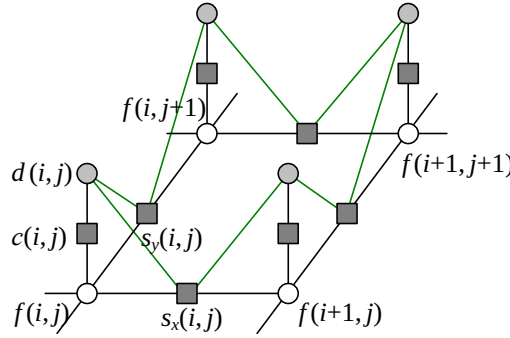


Figure 4.17 Graphical model for a conditional random field (CRF). The additional green edges show how combinations of sensed data influence the smoothness in the underlying MRF prior model, i.e., $s_x(i, j)$ and $s_y(i, j)$ in (4.42) depend on adjacent $d(i, j)$ values. These additional links (factors) enable the smoothness to depend on the input data. However, they make sampling from this MRF more complex.

In stereo matching, the idea of encouraging disparity discontinuities to coincide with intensity edges goes back even further to the early days of optimization and MRF-based algorithms (Poggio, Gamble, and Little 1988; Fua 1993; Bobick and Intille 1999; Boykov, Veksler, and Zabih 2001) and is discussed in more detail in (Section 12.5).

In addition to using smoothness terms that adapt to the input data, Kumar and Hebert (2003) also compute a neighborhood function over the input data for each $V_p(x_p, \mathbf{y})$ term, as illustrated in Figure 4.18, instead of using the classic unary MRF data term $V_p(x_p, y_p)$ shown in Figure 4.12.⁸ Because such neighborhood functions can be thought of as *discriminant* functions (a term widely used in machine learning (Bishop 2006)), they call the resulting graphical model a *discriminative random field* (DRF). In their paper, Kumar and Hebert (2006) show that DRFs outperform similar CRFs on a number of applications, such as structure detection and binary image denoising.

Here again, one could argue that previous stereo correspondence algorithms also look at a neighborhood of input data, either explicitly, because they compute correlation measures (Criminisi, Cross *et al.* 2006) as data terms, or implicitly, because even pixel-wise disparity costs look at several pixels in either the left or right image (Barnard 1989; Boykov, Veksler, and Zabih 2001).

What then are the advantages and disadvantages of using conditional or discriminative

⁸Kumar and Hebert (2006) call the unary potentials $V_p(x_p, \mathbf{y})$ *association potentials* and the pairwise potentials $V_{p,q}(x_p, y_q, \mathbf{y})$ *interaction potentials*.

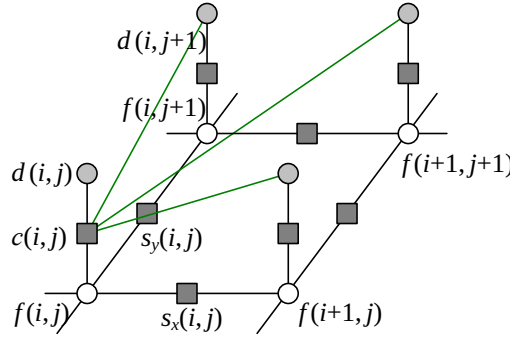


Figure 4.18 Graphical model for a discriminative random field (DRF). The additional green edges show how combinations of sensed data, e.g., $d(i, j + 1)$, influence the data term for $f(i, j)$. The generative model is therefore more complex, i.e., we cannot just apply a simple function to the unknown variables and add noise.

random fields instead of MRFs?

Classic Bayesian inference (MRF) assumes that the prior distribution of the data is independent of the measurements. This makes a lot of sense: if you see a pair of sixes when you first throw a pair of dice, it would be unwise to assume that they will always show up thereafter. However, if after playing for a long time you detect a statistically significant bias, you may want to adjust your prior. What CRFs do, in essence, is to select or modify the prior model based on observed data. This can be viewed as making a partial inference over additional hidden variables or correlations between the unknowns (say, a label, depth, or clean image) and the knowns (observed images).

In some cases, the CRF approach makes a lot of sense and is, in fact, the only plausible way to proceed. For example, in grayscale image colorization (Section 4.2.4) (Levin, Lischinski, and Weiss 2004), a commonly used way to transfer the continuity information from the input grayscale image to the unknown color image is to modify the local smoothness constraints. Similarly, for simultaneous segmentation and recognition (Winn and Shotton 2006; Shotton, Winn *et al.* 2009), it makes a lot of sense to permit strong color edges to increase the likelihood of semantic image label discontinuities.

In other cases, such as image denoising, the situation is more subtle. Using a non-quadratic (robust) smoothness term as in (4.42) plays a qualitatively similar role to setting the smoothness based on local gradient information in a Gaussian MRF (GMRF) (Tappen, Liu *et al.* 2007; Tanaka and Okutomi 2008). The advantage of Gaussian MRFs, when the smoothness can be correctly inferred, is that the resulting quadratic energy can be minimized

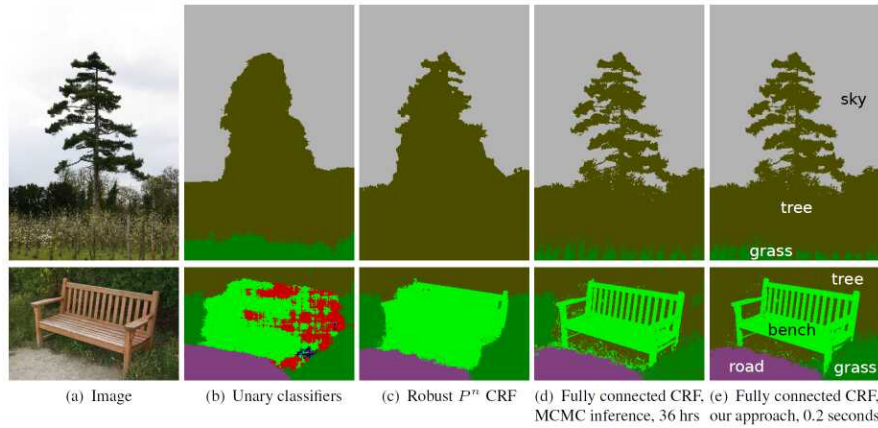


Figure 4.19 Pixel-level classification with a fully connected CRF, from © Krähenbühl and Koltun (2011). The labels in each column describe the image or algorithm being run, which include a robust P^n CRF (Kohli, Ladický, and Torr 2009) and a very slow MCMC optimization algorithm.

in a single step, i.e., by solving a sparse set of linear equations. However, for situations where the discontinuities are not self-evident in the input data, such as for piecewise-smooth sparse data interpolation (Blake and Zisserman 1987; Terzopoulos 1988), classic robust smoothness energy minimization may be preferable. Thus, as with most computer vision algorithms, a careful analysis of the problem at hand and desired robustness and computation constraints may be required to choose the best technique.

Perhaps the biggest advantage of CRFs and DRFs, as argued by Kumar and Hebert (2006), Tappen, Liu *et al.* (2007), and Blake, Rother *et al.* (2004), is that learning the model parameters is more principled and sometimes easier. While learning parameters in MRFs and their variants is not a topic that we cover in this book, interested readers can find more details in publications by Kumar and Hebert (2006), Roth and Black (2007a), Tappen, Liu *et al.* (2007), Tappen (2007), and Li and Huttenlocher (2008).

Dense Conditional Random Fields (CRFs)

As with regular Markov random fields, conditional random fields (CRFs) are normally defined over small neighborhoods, e.g., the $\mathcal{N}_4$ neighborhood shown in Figure 4.17. However, images often contain longer-range interactions, e.g., pixels of similar colors may belong to

related classes (Figure 4.19). In order to model such longer-range interactions, Krähenbühl and Koltun (2011) introduced what they call a *fully connected CRF*, which many people now call a *dense CRF*.

As with traditional conditional random fields (4.47), their energy function consists of both unary terms and pairwise terms

$$E(\mathbf{x}|\mathbf{y}) = \sum_p V_p(x_p, \mathbf{y}) + \sum_{(p,q)} V_{p,q}(x_p, x_q, y_p, y_q), \quad (4.48)$$

where the (p, q) summation is now taken over *all* pairs of pixels, and not just adjacent ones.⁹ The $\mathbf{y}$ denotes the input (guide) image over which the random field is conditioned. The pairwise interaction potentials have a restricted form

$$V_{p,q}(x_p, x_q, y_p, y_q) = \mu(x_p, x_q) \sum_{m=1}^M s_m w_m(p, q) \quad (4.49)$$

that is the product of a spatially invariant *label compatibility function* $\mu(x_p, x_q)$ and a sum of M Gaussian kernels of the same form (3.37) as is used in bilateral filtering and the bilateral solver. In their seminal paper, Krähenbühl and Koltun (2011) use two kernels, the first of which is an *appearance kernel* similar to (3.37) and the second is a spatial-only *smoothness kernel*.

Because of the special form of the long-range interaction potentials, which encapsulate all spatial and color similarity terms into a bilateral form, higher-dimensional filtering algorithms similar to those used in fast bilateral filters and solvers (Adams, Baek, and Davis 2010) can be used to efficiently compute a *mean field approximation* to the posterior conditional distribution (Krähenbühl and Koltun 2011). Figure 4.19 shows a comparison of their results (rightmost column) with previous approaches, including using simple unary terms, a robust CRF (Kohli, Ladický, and Torr 2009), and a very slow MCMC (Markov chain Monte Carlo) inference algorithm. As you can see, the fully connected CRF with a mean field solver produces dramatically better results in a very short time.

Since the publication of this paper, provably convergent and more efficient inference algorithms have been developed both by the original authors (Krähenbühl and Koltun 2013) and others (Vineet, Warrell, and Torr 2014; Desmaison, Bunel *et al.* 2016). Dense CRFs have seen widespread use in image segmentation problems and also as a “clean-up” stage for deep neural networks, as in the widely cited DeepLab paper by Chen, Papandreou *et al.* (2018).

⁹In practice, as with bilateral filtering and the bilateral solver, the spatial extent may be over a large but finite region.

4.3.2 Application: Interactive segmentation

The goal of image segmentation algorithms is to group pixels that have similar appearance (statistics) and to have the boundaries between pixels in different regions be of short length and across visible discontinuities. If we restrict the boundary measurements to be between immediate neighbors and compute region membership statistics by summing over pixels, we can formulate this as a classic pixel-based energy function using either a *variational formulation* (Section 4.2) or as a binary Markov random field (Section 4.3).

Examples of the continuous approach include Mumford and Shah (1989), Chan and Vese (2001), Zhu and Yuille (1996), and Tabb and Ahuja (1997) along with the level set approaches discussed in Section 7.3.2. An early example of a discrete labeling problem that combines both region-based and boundary-based energy terms is the work of Leclerc (1989), who used minimum description length (MDL) coding to derive the energy function being minimized. Boykov and Funka-Lea (2006) present a wonderful survey of various energy-based techniques for binary object segmentation, some of which we discuss below.

As we saw earlier in this chapter, the energy corresponding to a segmentation problem can be written (c.f. Equations (4.24) and (4.35–4.42)) as

$$E(f) = \sum_{i,j} E_R(i, j) + E_P(i, j), \quad (4.50)$$

where the region term

$$E_R(i, j) = C(I(i, j); R(f(i, j))) \quad (4.51)$$

is the negative log likelihood that pixel intensity (or color) $I(i, j)$ is consistent with the statistics of region $R(f(i, j))$ and the boundary term

$$E_P(i, j) = s_x(i, j)\delta(f(i, j), f(i+1, j)) + s_y(i, j)\delta(f(i, j), f(i, j+1)) \quad (4.52)$$

measures the inconsistency between $\mathcal{N}_4$ neighbors modulated by local horizontal and vertical smoothness terms $s_x(i, j)$ and $s_y(i, j)$.

Region statistics can be something as simple as the mean gray level or color (Leclerc 1989), in which case

$$C(I; \mu_k) = \|I - \mu_k\|^2. \quad (4.53)$$

Alternatively, they can be more complex, such as region intensity histograms (Boykov and Jolly 2001) or color Gaussian mixture models (Rother, Kolmogorov, and Blake 2004). For smoothness (boundary) terms, it is common to make the strength of the smoothness $s_x(i, j)$ inversely proportional to the local edge strength (Boykov, Veksler, and Zabih 2001).

Originally, energy-based segmentation problems were optimized using iterative gradient descent techniques, which were slow and prone to getting trapped in local minima. Boykov

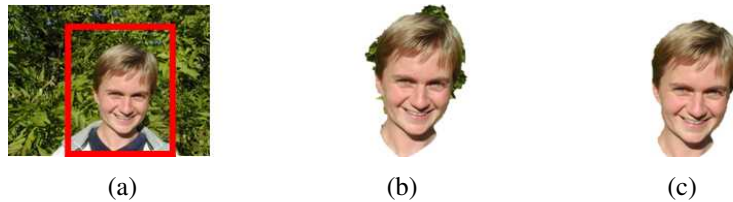


Figure 4.20 *GrabCut image segmentation (Rother, Kolmogorov, and Blake 2004) © 2004 ACM: (a) the user draws a bounding box in red; (b) the algorithm guesses color distributions for the object and background and performs a binary segmentation; (c) the process is repeated with better region statistics.*

and Jolly (2001) were the first to apply the binary MRF optimization algorithm developed by Greig, Porteous, and Seheult (1989) to binary object segmentation.

In this approach, the user first delineates pixels in the background and foreground regions using a few strokes of an image brush. These pixels then become the *seeds* that tie nodes in the S – T graph to the source and sink labels S and T . Seed pixels can also be used to estimate foreground and background region statistics (intensity or color histograms).

The capacities of the other edges in the graph are derived from the region and boundary energy terms, i.e., pixels that are more compatible with the foreground or background region get stronger connections to the respective source or sink; adjacent pixels with greater smoothness also get stronger links. Once the minimum-cut/maximum-flow problem has been solved using a polynomial time algorithm (Goldberg and Tarjan 1988; Boykov and Kolmogorov 2004), pixels on either side of the computed cut are labeled according to the source or sink to which they remain connected. While graph cuts is just one of several known techniques for MRF energy minimization, it is still the one most commonly used for solving binary MRF problems.

The basic binary segmentation algorithm of Boykov and Jolly (2001) has been extended in a number of directions. The *GrabCut* system of Rother, Kolmogorov, and Blake (2004) iteratively re-estimates the region statistics, which are modeled as a mixtures of Gaussians in color space. This allows their system to operate given minimal user input, such as a single bounding box (Figure 4.20a)—the background color model is initialized from a strip of pixels around the box outline. (The foreground color model is initialized from the interior pixels, but quickly converges to a better estimate of the object.) The user can also place additional strokes to refine the segmentation as the solution progresses. Cui, Yang *et al.* (2008) use color and edge models derived from previous segmentations of similar objects to improve the local models used in GrabCut. Graph cut algorithms and other variants of Markov and conditional

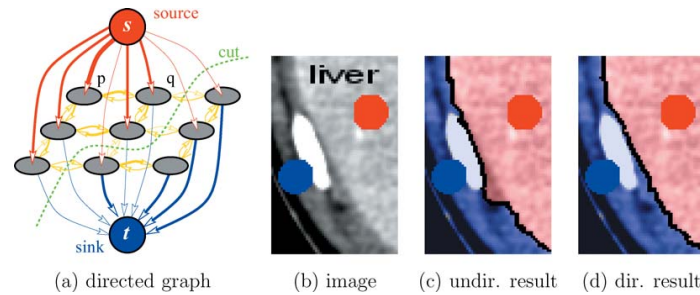


Figure 4.21 Segmentation with a directed graph cut (Boykov and Funka-Lea 2006) © 2006 Springer: (a) directed graph; (b) image with seed points; (c) the undirected graph incorrectly continues the boundary along the bright object; (d) the directed graph correctly segments the light gray region from its darker surround.

random fields have been applied to the *semantic segmentation* problem (Shotton, Winn *et al.* 2009; Krähenbühl and Koltun 2011), an example of which is shown in Figure 4.19 and which we study in more detail in Section 6.4.

Another major extension to the original binary segmentation formulation is the addition of *directed edges*, which allows boundary regions to be oriented, e.g., to prefer light to dark transitions or *vice versa* (Kolmogorov and Boykov 2005). Figure 4.21 shows an example where the directed graph cut correctly segments the light gray liver from its dark gray surround. The same approach can be used to measure the *flux* exiting a region, i.e., the signed gradient projected normal to the region boundary. Combining oriented graphs with larger neighborhoods enables approximating continuous problems such as those traditionally solved using level sets in the globally optimal graph cut framework (Boykov and Kolmogorov 2003; Kolmogorov and Boykov 2005).

More recent developments in graph cut-based segmentation techniques include the addition of connectivity priors to force the foreground to be in a single piece (Vicente, Kolmogorov, and Rother 2008) and shape priors to use knowledge about an object's shape during the segmentation process (Lempitsky and Boykov 2007; Lempitsky, Blake, and Rother 2008).

While optimizing the binary MRF energy (4.50) requires the use of combinatorial optimization techniques, such as maximum flow, an approximate solution can be obtained by converting the binary energy terms into quadratic energy terms defined over a continuous $[0, 1]$ random field, which then becomes a classical membrane-based regularization problem (4.24–4.27). The resulting quadratic energy function can then be solved using standard linear system solvers (4.27–4.28), although if speed is an issue, you should use multigrid or one

of its variants (Appendix A.5). Once the continuous solution has been computed, it can be thresholded at 0.5 to yield a binary segmentation.

The $[0, 1]$ continuous optimization problem can also be interpreted as computing the probability at each pixel that a *random walker* starting at that pixel ends up at one of the labeled seed pixels, which is also equivalent to computing the potential in a resistive grid where the resistors are equal to the edge weights (Grady 2006; Sinop and Grady 2007). K -way segmentations can also be computed by iterating through the seed labels, using a binary problem with one label set to 1 and all the others set to 0 to compute the relative membership probabilities for each pixel. In follow-on work, Grady and Ali (2008) use a precomputation of the eigenvectors of the linear system to make the solution with a novel set of seeds faster, which is related to the Laplacian matting problem presented in Section 10.4.3 (Levin, Acha, and Lischinski 2008). Couprie, Grady *et al.* (2009) relate the random walker to watersheds and other segmentation techniques. Singaraju, Grady, and Vidal (2008) add directed-edge constraints in order to support flux, which makes the energy piecewise quadratic and hence not solvable as a single linear system. The random walker algorithm can also be used to solve the Mumford–Shah segmentation problem (Grady and Alvino 2008) and to compute fast multi-grid solutions (Grady 2008). A nice review of these techniques is given by Singaraju, Grady *et al.* (2011).

An even faster way to compute a continuous $[0, 1]$ approximate segmentation is to compute *weighted geodesic distances* between the 0 and 1 seed regions (Bai and Sapiro 2009), which can also be used to estimate soft alpha mattes (Section 10.4.3). A related approach by Criminisi, Sharp, and Blake (2008) can be used to find fast approximate solutions to general binary Markov random field optimization problems.

4.4 Additional reading

Scattered data interpolation and approximation techniques are fundamental to many different branches of applied mathematics. Some good introductory texts and articles include Amidror (2002), Wendland (2004), and Anjyo, Lewis, and Pighin (2014). These techniques are also related to geometric modeling techniques in computer graphics, which continues to be a very active research area. A nice introduction to basic spline techniques for curves and surfaces can be found in Farin (2002), while more recent approaches using subdivision surfaces are covered in Peters and Reif (2008).

Data interpolation and approximation also lie at the heart of *regression techniques*, which form the mathematical basis for most of the machine learning techniques we study in the next chapter. You can find good introductions to this topic (as well as underfitting, overfitting,

and model selection) in texts on classic machine learning (Bishop 2006; Hastie, Tibshirani, and Friedman 2009; Murphy 2012; Deisenroth, Faisal, and Ong 2020) and deep learning (Goodfellow, Bengio, and Courville 2016; Glassner 2018; Zhang, Lipton *et al.* 2021).

Robust data fitting is also central to most computer vision problems. While introduced in this chapter, it is also revisited in Appendix B.3. Classic textbooks and articles on robust fitting and statistics include Huber (1981), Hampel, Ronchetti *et al.* (1986), Black and Rangarajan (1996), Rousseeuw and Leroy (1987), and Stewart (1999). The recent paper by Barron (2019) unifies many of the commonly used robust potential functions and shows how they can be used in machine learning applications.

The regularization approach to computer vision problems was first introduced to the vision community by Poggio, Torre, and Koch (1985) and Terzopoulos (1986a,b, 1988) and continues to be a popular framework for formulating and solving low-level vision problems (Ju, Black, and Jepson 1996; Nielsen, Florack, and Deriche 1997; Nordström 1990; Brox, Bruhn *et al.* 2004; Levin, Lischinski, and Weiss 2008). More detailed mathematical treatment and additional applications can be found in the applied mathematics and statistics literature (Tikhonov and Arsenin 1977; Engl, Hanke, and Neubauer 1996).

Variational formulations have been extensively used in low-level computer vision tasks, including optical flow (Horn and Schunck 1981; Nagel and Enkelmann 1986; Black and Anandan 1993; Alvarez, Weickert, and Sánchez 2000; Brox, Bruhn *et al.* 2004; Zach, Pock, and Bischof 2007a; Wedel, Cremers *et al.* 2009; Werlberger, Pock, and Bischof 2010), segmentation (Kass, Witkin, and Terzopoulos 1988; Mumford and Shah 1989; Caselles, Kimmel, and Sapiro 1997; Paragios and Deriche 2000; Chan and Vese 2001; Osher and Paragios 2003; Cremers 2007), denoising (Rudin, Osher, and Fatemi 1992), stereo (Pock, Schoenemann *et al.* 2008), multi-view stereo (Faugeras and Keriven 1998; Yezzi and Soatto 2003; Pons, Keriven, and Faugeras 2007; Labatut, Pons, and Keriven 2007; Kolev, Klodt *et al.* 2009), and scene flow (Wedel, Brox *et al.* 2011).

The literature on Markov random fields is truly immense, with publications in related fields such as optimization and control theory of which few vision practitioners are even aware. A good guide to the latest techniques is the book edited by Blake, Kohli, and Rother (2011). Other articles that contain nice literature reviews or experimental comparisons include Boykov and Funka-Lea (2006), Szeliski, Zabih *et al.* (2008), Kumar, Veksler, and Torr (2011), and Kappes, Andres *et al.* (2015). MRFs are just one version of the more general topic of graphical models, which is covered in several textbooks and survey, including Bishop (2006, Chapter 8), Koller and Friedman (2009), Nowozin and Lampert (2011), and Murphy (2012, Chapters 10, 17, 19)).

The seminal paper on Markov random fields is the work of Geman and Geman (1984),

who introduced this formalism to computer vision researchers and also introduced the notion of *line processes*, additional binary variables that control whether smoothness penalties are enforced or not. Black and Rangarajan (1996) showed how independent line processes could be replaced with robust pairwise potentials; Boykov, Veksler, and Zabih (2001) developed iterative binary graph cut algorithms for optimizing multi-label MRFs; Kolmogorov and Zabih (2004) characterized the class of binary energy potentials required for these techniques to work; and Freeman, Pasztor, and Carmichael (2000) popularized the use of loopy belief propagation for MRF inference. Many more additional references can be found in Sections 4.3 and 4.3.2, and Appendix B.5.

Continuous-energy-based (variational) approaches to interactive segmentation include Leclerc (1989), Mumford and Shah (1989), Chan and Vese (2001), Zhu and Yuille (1996), and Tabb and Ahuja (1997). Discrete variants of such problems are usually optimized using binary graph cuts or other combinatorial energy minimization methods (Boykov and Jolly 2001; Boykov and Kolmogorov 2003; Rother, Kolmogorov, and Blake 2004; Kolmogorov and Boykov 2005; Cui, Yang *et al.* 2008; Vicente, Kolmogorov, and Rother 2008; Lempitsky and Boykov 2007; Lempitsky, Blake, and Rother 2008), although continuous optimization techniques followed by thresholding can also be used (Grady 2006; Grady and Ali 2008; Singaraju, Grady, and Vidal 2008; Criminisi, Sharp, and Blake 2008; Grady 2008; Bai and Sapiro 2009; Couprie, Grady *et al.* 2009). Boykov and Funka-Lea (2006) present a good survey of various energy-based techniques for binary object segmentation.

4.5 Exercises

Ex 4.1: Data fitting (scattered data interpolation). Generate some random samples from a smoothly varying function and then implement and evaluate one or more data interpolation techniques.

1. Generate a “random” 1-D or 2-D function by adding together a small number of sinusoids or Gaussians of random amplitudes and frequencies or scales.
2. Sample this function at a few dozen random locations.
3. Fit a function to these data points using one or more of the scattered data interpolation techniques described in Section 4.1.
4. Measure the fitting error between the estimated and original functions at some set of location, e.g., on a regular grid or at different random points.

5. Manually adjust any parameters your fitting algorithm may have to minimize the output sample fitting error, or use an automated technique such as cross-validation.
6. Repeat this exercise with a new set of random input sample and output sample locations. Does the optimal parameter change, and if so, by how much?
7. (Optional) Generate a piecewise-smooth test function by using different random parameters in different parts of your image. How much more difficult does the data fitting problem become? Can you think of ways you might mitigate this?

Try to implement your algorithm in NumPy (or Matlab) using only array operations, in order to become more familiar with data-parallel programming and the linear algebra operators built into these systems. Use data visualization techniques such as those in Figures 4.3–4.6 to debug your algorithms and illustrate your results.

Ex 4.2: Graphical model optimization. Download and test out the software on the OpenGM2 library and benchmarks web site <http://hciweb2.iwr.uni-heidelberg.de/opengm> (Kappes, Andres *et al.* 2015). Try applying these algorithms to your own problems of interest (segmentation, de-noising, etc.). Which algorithms are more suitable for which problems? How does the quality compare to deep learning based approaches, which we study in the next chapter?

Ex 4.3: Image deblocking—challenging. Now that you have some good techniques to distinguish signal from noise, develop a technique to remove the *blocking artifacts* that occur with JPEG at high compression settings (Section 2.3.3). Your technique can be as simple as looking for unexpected edges along block boundaries, or looking at the quantization step as a projection of a convex region of the transform coefficient space onto the corresponding quantized values.

1. Does the knowledge of the compression factor, which is available in the JPEG header information, help you perform better deblocking? See Ehrlich, Lim *et al.* (2020) for a recent paper on this topic.
2. Because the quantization occurs in the DCT transformed YCbCr space (2.116), it may be preferable to perform the analysis in this space. On the other hand, image priors make more sense in an RGB space (or do they?). Decide how you will approach this dichotomy and discuss your choice.
3. While you are at it, since the YCbCr conversion is followed by a chrominance subsampling stage (before the DCT), see if you can restore some of the lost high-frequency chrominance signal using one of the better restoration techniques discussed in this chapter.

4. If your camera has a RAW + JPEG mode, how close can you come to the noise-free true pixel values? (This suggestion may not be that useful, since cameras generally use reasonably high quality settings for their RAW + JPEG models.)

Ex 4.4: Inference in deblurring—challenging. Write down the graphical model corresponding to Figure 4.15 for a non-blind image deblurring problem, i.e., one where the blur kernel is known ahead of time.

What kind of efficient inference (optimization) algorithms can you think of for solving such problems?

Chapter 5

Deep Learning

5.1	Supervised learning	239
5.1.1	Nearest neighbors	241
5.1.2	Bayesian classification	243
5.1.3	Logistic regression	248
5.1.4	Support vector machines	250
5.1.5	Decision trees and forests	254
5.2	Unsupervised learning	257
5.2.1	Clustering	257
5.2.2	K-means and Gaussians mixture models	259
5.2.3	Principal component analysis	262
5.2.4	Manifold learning	265
5.2.5	Semi-supervised learning	266
5.3	Deep neural networks	268
5.3.1	Weights and layers	270
5.3.2	Activation functions	272
5.3.3	Regularization and normalization	274
5.3.4	Loss functions	280
5.3.5	Backpropagation	284
5.3.6	Training and optimization	287
5.4	Convolutional neural networks	291
5.4.1	Pooling and unpooling	295
5.4.2	<i>Application: Digit classification</i>	298
5.4.3	Network architectures	299
5.4.4	Model zoos	304
5.4.5	Visualizing weights and activations	307
5.4.6	Adversarial examples	311
5.4.7	Self-supervised learning	312
5.5	More complex models	317
5.5.1	Three-dimensional CNNs	317
5.5.2	Recurrent neural networks	321
5.5.3	Transformers	322
5.5.4	Generative models	328
5.6	Additional reading	336
5.7	Exercises	337

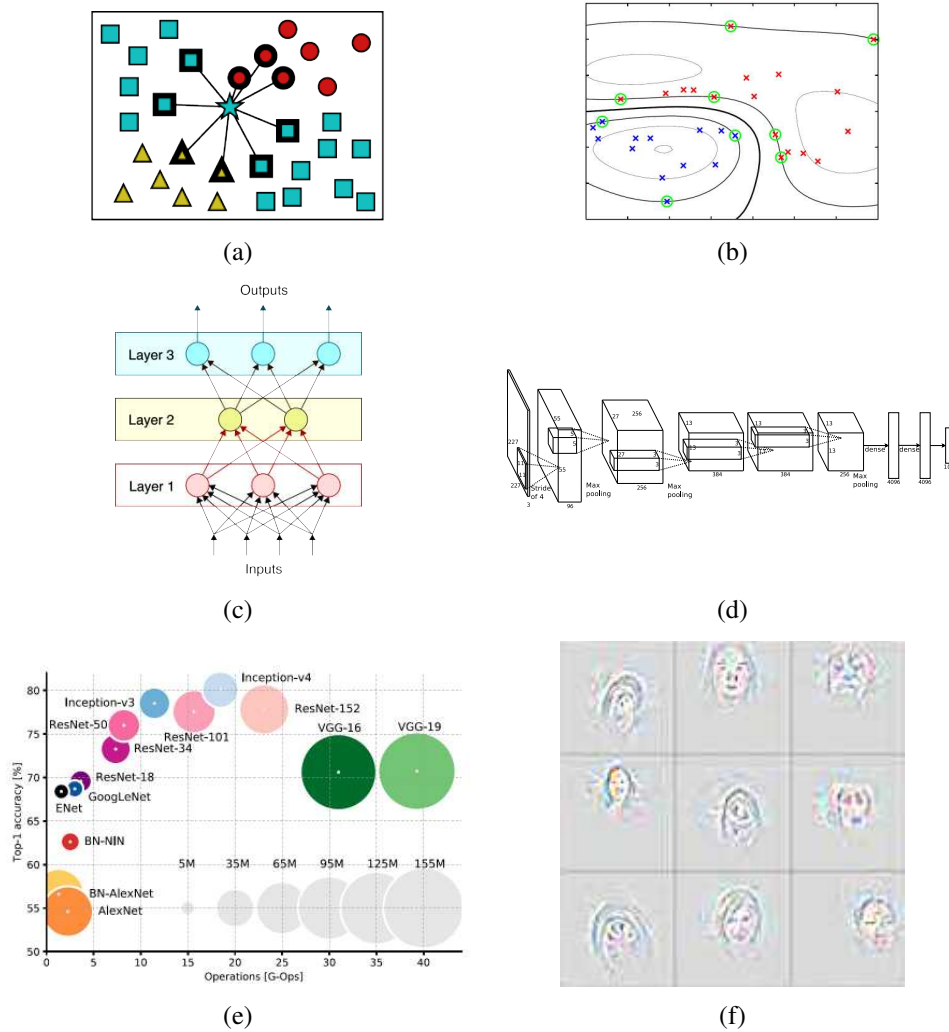


Figure 5.1 Machine learning and deep neural networks: (a) nearest neighbor classification © Glassner (2018); (b) Gaussian kernel support vector machine (Bishop 2006) © 2006 Springer; (c) a simple three-layer network © Glassner (2018); (d) the SuperVision deep neural network, courtesy of Matt Deitke after (Krizhevsky, Sutskever, and Hinton 2012); (e) network accuracy vs. size and operation counts (Canziani, Culurciello, and Paszke 2017) © 2017 IEEE; (f) visualizing network features (Zeiler and Fergus 2014) © 2014 Springer.

Machine learning techniques have always played an important and often central role in the development of computer vision algorithms. Computer vision in the 1970s grew out of the fields of artificial intelligence, digital image processing, and pattern recognition (now called machine learning), and one of the premier journals in our field (*IEEE Transactions on Pattern Analysis and Machine Intelligence*) still bears testament to this heritage.

The image processing, scattered data interpolation, variational energy minimization, and graphical model techniques introduced in the previous two chapters have been essential tools in computer vision over the last five decades. While elements of machine learning and pattern recognition have also been widely used, e.g., for fine-tuning algorithm parameters, they really came into their own with the availability of large-scale labeled image datasets, such as ImageNet (Deng, Dong *et al.* 2009; Russakovsky, Deng *et al.* 2015), COCO (Lin, Maire *et al.* 2014), and LVIS (Gupta, Dollár, and Girshick 2019). Currently, deep neural networks are the most popular and widely used machine learning models in computer vision, not just for semantic classification and segmentation, but even for lower-level tasks such as image enhancement, motion estimation, and depth recovery (Bengio, LeCun, and Hinton 2021).

Figure 5.2 shows the main distinctions between traditional computer vision techniques, in which all of the processing stages were designed by hand, machine learning algorithms, in which hand-crafted features were passed on to a machine learning stage, and deep networks, in which all of the algorithm components, including mid-level representations, are learned directly from the training data.

We begin this chapter with an overview of classical machine learning approaches, such as nearest neighbors, logistic regression, support vector machines, and decision forests. This is a broad and deep subject, and we only provide a brief summary of the main popular approaches. More details on these techniques can be found in textbooks on this subject, which include Bishop (2006), Hastie, Tibshirani, and Friedman (2009), Murphy (2012), Criminisi and Shotton (2013), and Deisenroth, Faisal, and Ong (2020).

The machine learning part of the chapter focuses mostly on *supervised learning* for *classification* tasks, in which we are given a collection of inputs $\{\mathbf{x}_i\}$, which may be features derived from input images, paired with their corresponding class labels (or targets) $\{t_i\}$, which come from a set of classes $\{\mathcal{C}_k\}$. Most of the techniques described for supervised classification can easily be extended to *regression*, i.e., associating inputs $\{\mathbf{x}_i\}$ with real-valued scalar or vector outputs $\{\mathbf{y}_i\}$, which we have already studied in Section 4.1. We also look at some examples of *unsupervised learning* (Section 5.2), where there are no labels or outputs, as well as *semi-supervised learning*, in which labels or targets are only provided for a subset of the samples.

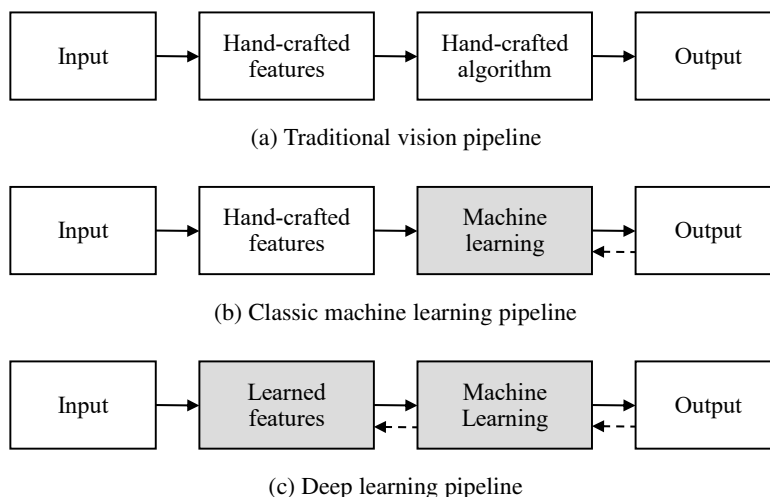


Figure 5.2 *Traditional, machine learning, and deep learning pipelines, inspired by Goodfellow, Bengio, and Courville (2016, Figure 1.5). In a classic vision pipeline such as structure from motion, both the features and the algorithm were traditionally designed by hand (although learning techniques could be used, e.g., to design more repeatable features). Classic machine learning approaches take extracted features and use machine learning to build a classifier. Deep learning pipelines learn the whole pipeline, starting from pixels all the way to outputs, using end-to-end training (indicated by the backward dashed arrows) to fine-tune the model parameters.*

The second half of this chapter focuses on *deep neural networks*, which, over the last decade, have become the method of choice for most computer vision recognition and lower-level vision tasks. We begin with the elements that make up deep neural networks, including weights and activations, regularization terms, and training using backpropagation and stochastic gradient descents. Next, we introduce convolutional layers, review some of the classic architectures, and talk about how to pre-train networks and visualize their performance. Finally, we briefly touch on more advanced networks, such as three-dimensional and spatio-temporal models, as well as recurrent and generative adversarial networks.

Because machine learning and deep learning are such rich and deep topics, this chapter just briefly summarizes some of the main concepts and techniques. Comprehensive texts on classic machine learning include Bishop (2006), Hastie, Tibshirani, and Friedman (2009), Murphy (2012), and Deisenroth, Faisal, and Ong (2020) while textbooks focusing on deep learning include Goodfellow, Bengio, and Courville (2016), Glassner (2018), Glassner (2021),

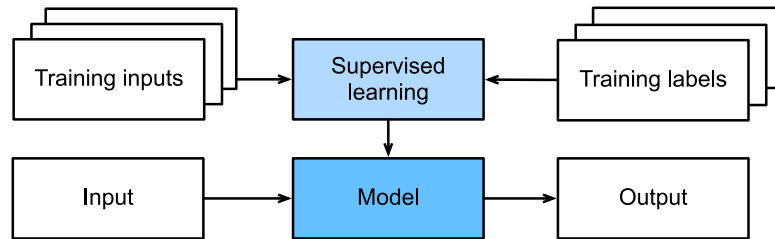


Figure 5.3 In supervised learning, paired training inputs and labels are used to estimate the model parameters that best predict the labels from their corresponding inputs. At run time, the model parameters are (usually) frozen, and the model is applied to new inputs to generate the desired outputs. © Zhang, Lipton *et al.* (2021, Figure 1.3)

and Zhang, Lipton *et al.* (2021).

5.1 Supervised learning

Machine learning algorithms are usually categorized as either *supervised*, where paired inputs and outputs are given to the learning algorithm (Figure 5.3), or *unsupervised*, where statistical samples are provided without any corresponding labeled outputs (Section 5.2).

As shown in Figure 5.3, supervised learning involves feeding pairs of inputs $\{\mathbf{x}_i\}$ and their corresponding *target* output values $\{t_i\}$ into a learning algorithm, which adjusts the model’s parameters so as to maximize the agreement between the model’s predictions and the target outputs. The outputs can either be discrete labels that come from a set of classes $\{\mathcal{C}_k\}$, or they can be a set of continuous, potentially vector-valued *values*, which we denote by $\mathbf{y}_i$ to make the distinction between the two cases clearer. The first task is called *classification*, since we are trying to predict class membership, while the second is called *regression*, since historically, fitting a trend to data was called by that name (Section 4.1).¹

After a *training phase* during which all of the *training data* (labeled input-output pairs) have been processed (often by iterating over them many times), the trained model can now be used to predict new output values for previously unseen inputs. This phase is often called the *test phase*, although this sometimes fools people into focusing excessively on performance on a given test set, rather than building a system that works robustly for any plausible inputs that might arise.

¹Note that in software engineering, a *regression* sometimes means a change in the code that results in degraded performance. That is not the kind of regression we will be studying here.

In this section, we focus more on classification, since we’ve already covered some of the simpler (linear and kernel) methods for regression in the previous chapter. One of the most common applications of classification in computer vision is semantic *image classification*, where we wish to label a complete image (or predetermined portion) with its most likely semantic category, e.g., horse, cat, or car (Section 6.2). This is the main application for which deep networks (Sections 5.3–5.4) were originally developed. More recently, however, such networks have also been applied to continuous pixel labeling tasks such as semantic segmentation, image denoising, and depth and motion estimation. More sophisticated tasks, such as object detection and instance segmentation, will be covered in Chapter 6.

Before we begin our review of traditional supervised learning techniques, we should define a little more formally what the system is trying to learn, i.e., what we meant by “maximize the agreement between the model’s predictions and the target outputs.” Ultimately, like any other computer algorithm that will occasionally make mistakes under uncertain, noisy, and/or incomplete data, we would like to maximize its expected utility, or conversely, minimize its expected *loss* or *risk*. This is the subject of *decision theory*, which is explained in more detail in textbooks on machine learning (Bishop 2006, Section 1.5; Hastie, Tibshirani, and Friedman 2009, Section 2.4; Murphy 2012, Section 6.5; Deisenroth, Faisal, and Ong 2020, Section 8.2).

We usually do not have access to the true probability distribution over the inputs, let alone the joint distribution over inputs and corresponding outputs. For this reason, we often use the training data distribution as a proxy for the real-world distribution. This approximation is known as *empirical risk minimization* (see above citations on decision theory), where the expected risk can be estimated with

$$E_{\text{Risk}}(\mathbf{w}) = \frac{1}{N} \sum L(\mathbf{y}_i, \mathbf{f}(\mathbf{x}_i; \mathbf{w})). \quad (5.1)$$

The loss function L measures the “cost” of predicting an output $\mathbf{f}(\mathbf{x}_i; \mathbf{w})$ for input $\mathbf{x}_i$ and model parameters $\mathbf{w}$ when the corresponding target is $\mathbf{y}_i$.²

This formula should by now be quite familiar, since it is the same one we introduced in the previous chapter (4.2; 4.15) for *regression*. In those cases, the cost (penalty) is a simple quadratic or robust function of the difference between the target output $\mathbf{y}_i$ and the output predicted by the model $\mathbf{f}(\mathbf{x}_i; \mathbf{w})$. In some situations, we may want the loss to model specific asymmetries in misprediction. For example, in autonomous navigation, it is usually more costly to over-estimate the distance to the nearest obstacle, potentially resulting in a collision, than to more conservatively under-estimate. We will see more examples of loss functions

²In the machine learning literature, it is more common to write the loss using the letter L . But since we have used the letter E for energy (or summed error) in the previous chapter, we will stick to that notation throughout the book.

later on in this chapter, including Section 5.1.3 on Bayesian classification (5.19–5.24) and Section 5.3.4 on neural network loss (5.54–5.56).

In classification tasks, it is common to minimize the *misclassification rate*, i.e., penalizing all class prediction errors equally using a class-agnostic delta function (Bishop 2006, Sections 1.5.1–1.5.2). However, asymmetries often exist. For example, the cost of producing a *false negative* diagnosis in medicine, which may result in an untreated illness, is often greater than that of a *false positive*, which may suggest further tests. We will discuss true and false positives and negatives, along with error rates, in more detail in Section 7.1.3.

Data preprocessing

Before we start our review of widely used machine learning techniques, we should mention that it is usually a good idea to *center*, *standardize*, and if possible, *whiten* the input data (Glassner 2018, Section 10.5; Bishop 2006, Section 12.1.3). *Centering* the feature vectors means subtracting their mean value, while *standardizing* means also re-scaling each component so that its variance (average squared distance from the mean) is 1.

Whitening is a more computationally expensive process, which involves computing the covariance matrix of the inputs, taking its SVD, and then rotating the coordinate system so that the final dimensions are uncorrelated and have unit variance (under a Gaussian model). While this may be quite practical and helpful for low-dimension inputs, it can become prohibitively expensive for large sets of images. (But see the discussion in Section 5.2.3 on principal component analysis, where it can be feasible and useful.)

With this background in place, we now turn our attention to some widely used supervised learning techniques, namely nearest neighbors, Bayesian classification, logistic regression, support vector machines, and decision trees and forests.

5.1.1 Nearest neighbors

Nearest neighbors is a very simple *non-parametric* technique, i.e., one that does not involve a low-parameter analytic form for the underlying distribution. Instead, the training examples are all retained, and at evaluation time the “nearest” k neighbors are found and then averaged to produce the output.³

Figure 5.4 shows a simple graphical example for various values of k , i.e., from using the $k = 1$ nearest neighbor all the way to finding the $k = 25$ nearest neighbors and selecting

³The reason I put “nearest” in quotations is that standardizing and/or whitening the data will affect distances between vectors, and is usually helpful.

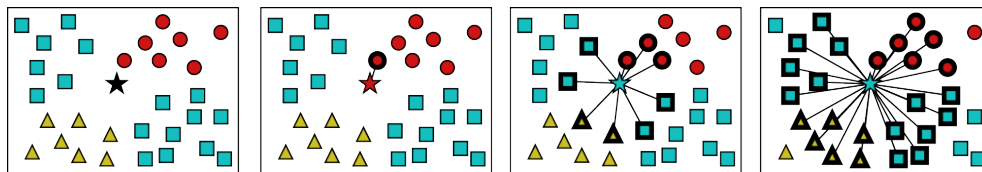


Figure 5.4 *Nearest neighbor classification. To determine the class of the star (★) test sample, we find the k nearest neighbors and select the most popular class. This figure shows the results for $k = 1, 9$, and 25 samples. © Glassner (2018)*

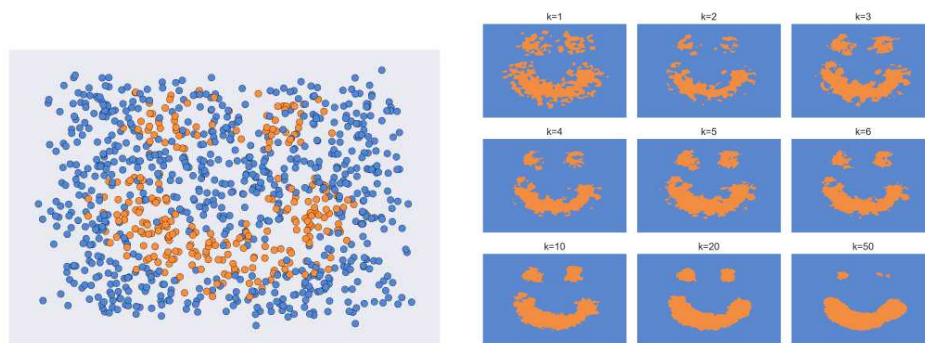


Figure 5.5 *For noisy (intermingled) data, selecting too small a value of k results in irregular decision surfaces. Selecting too large a value can cause small regions to shrink or disappear. © Glassner (2018)*

the class with the highest count as the output label. As you can see, changing the number of neighbors affects the final class label, which changes from red to blue.

Figure 5.5 shows the effect of varying the number of neighbors in another way. The left half of the figure shows the initial samples, which fall into either blue or orange categories. As you can see, the training samples are highly *intermingled*, i.e., there is no clear (plausible) boundary that will correctly label all of the samples. The right side of this figure shows the *decision boundaries* for a k -NN classifier as we vary the values of k from 1 to 50. When k is too small, the classifier acts in a very random way, i.e., it is *overfitting* to the training data (Section 4.1.2). As k gets larger, the classifier *underfits* (over-smooths) the data, resulting in the shrinkage of the two smaller regions. The optimal number of nearest neighbors to use k is a *hyperparameter* for this algorithm. Techniques for determining a good value include cross-validation, which we discussed in Section 4.1.2.

While nearest neighbors is a rather brute-force machine learning technique (although

Cover and Hart (1967) showed that it is statistically optimal in the large sample limit), but it can still be useful in many computer vision applications, such as large-scale matching and indexing (Section 7.1.4). As the number of samples gets large, however, efficient techniques must be used to find the (exact or approximate) nearest neighbors. Good algorithms for finding nearest neighbors have been developed in both the general computer science and more specialized computer vision communities.

Muja and Lowe (2014) developed a Fast Library for Approximate Nearest Neighbors (FLANN), which collects a number of previously developed algorithms and is incorporated as part of OpenCV. The library implements several powerful approximate nearest neighbor algorithms, including randomized k-d trees (Silpa-Anan and Hartley 2008), priority search k-means trees, approximate nearest neighbors (Friedman, Bentley, and Finkel 1977), and locality sensitive hashing (LSH) (Andoni and Indyk 2006). Their library can empirically determine which algorithm and parameters to use based on the characteristics of the data being indexed.

More recently, Johnson, Douze, and Jégou (2021) developed the GPU-enabled Faiss library⁴ for scaling similarity search (Section 6.2.3) to billions of vectors. The library is based on product quantization (Jégou, Douze, and Schmid 2010), which had been shown by the authors to perform better than LSH (Gordo, Perronnin *et al.* 2013) on the kinds of large-scale datasets the Faiss library was developed for.

5.1.2 Bayesian classification

For some simple machine learning problems, e.g., if we have an analytic model of feature construction and noising, or if we can gather enough samples, we can determine the probability distributions of the feature vectors for each class $p(\mathbf{x}|\mathcal{C}_k)$ as well as the prior class likelihoods $p(\mathcal{C}_k)$.⁵ According to Bayes' rule (4.33), the likelihood of class $\mathcal{C}_k$ given a feature vector $\mathbf{x}$ (Figure 5.6) is given by

$$p_k = p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{\sum_j p(\mathbf{x}|\mathcal{C}_j)p(\mathcal{C}_j)} \quad (5.2)$$

$$= \frac{\exp l_k}{\sum_j \exp l_j}, \quad (5.3)$$

⁴<https://github.com/facebookresearch/faiss>

⁵The following notation and equations are adapted from Bishop (2006, Section 4.2), which describes *probabilistic generative classification*.

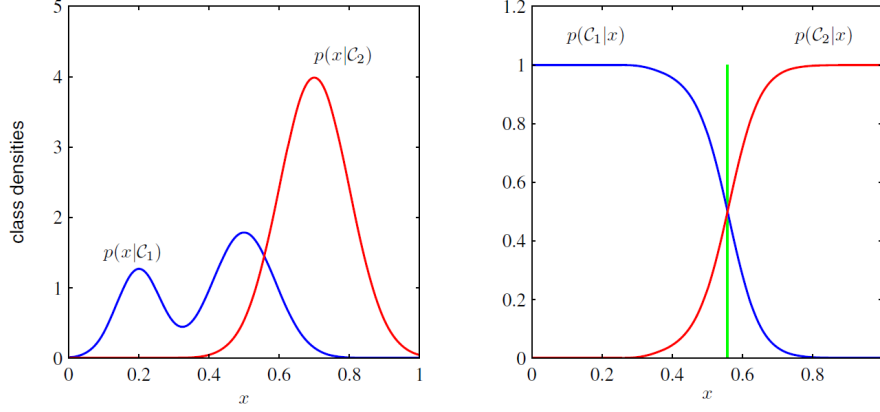


Figure 5.6 An example with two class conditional densities $p(x|C_k)$ along with the corresponding posterior class probabilities $p(C_k|x)$, which can be obtained using Bayes' rule, i.e., by dividing by the sum of the two curves (Bishop 2006) © 2006 Springer. The vertical green line is the optimal decision boundary for minimizing the misclassification rate.

where the second form (using the exp functions) is known as the *normalized exponential* or *softmax* function.⁶ The quantity

$$l_k = \log p(\mathbf{x}|C_k) + \log p(C_k) \quad (5.4)$$

is the *log-likelihood* of sample $\mathbf{x}$ being from class C_k .⁷ It is sometimes convenient to denote the softmax function (5.3) as a vector-to-vector valued function,

$$\mathbf{p} = \text{softmax}(\mathbf{l}). \quad (5.5)$$

The softmax function can be viewed as a soft version of a maximum indicator function, which returns 1 for the largest value of l_k whenever it dominates the other values. It is widely used in machine learning and statistics, including its frequent use as the final non-linearity in deep neural classification networks (Figure 5.27).

The process of using formula (5.2) to determine the likelihood of a class C_k given a feature vector $\mathbf{x}$ is known as *Bayesian classification*, since it combines a conditional feature likelihood $p(\mathbf{x}|C_k)$ with a prior distribution over classes $p(C_k)$ using Bayes' rule to determine

⁶For better numerical stability, it is common to subtract the largest value of l_j from all of the input values so that the exponentials are in the range $(0, 1]$ and there is less chance of roundoff error.

⁷Some authors (e.g., Zhang, Lipton *et al.* 2021) use the term *logit* for the log-likelihood, although it is more commonly used to denote the *log odds*, discussed below, or the softmax function itself.

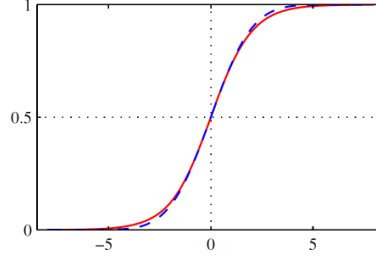


Figure 5.7 The logistic sigmoid function $\sigma(l)$, shown in red, along with a scaled error function, shown in dashed blue (Bishop 2006) © 2006 Springer.

the posterior class probabilities. In the case where the components of the feature vector are generated independently, i.e.,

$$p(\mathbf{x}|\mathcal{C}_k) = \prod_i p(x_i|\mathcal{C}_k), \quad (5.6)$$

the resulting technique is called a *naïve Bayes classifier*.

For the binary (two class) classification task, we can re-write (5.3) as

$$p(\mathcal{C}_0|\mathbf{x}) = \frac{1}{1 + \exp(-l)} = \sigma(l), \quad (5.7)$$

where $l = l_0 - l_1$ is the difference between the two class log likelihood and is known as the *log odds* or *logit*.

The $\sigma(l)$ function is called the *logistic sigmoid function* (or simply the *logistic function* or *logistic curve*), where *sigmoid* means an S-shaped curve (Figure 5.7). The sigmoid was a popular *activation function* in earlier neural networks, although it has now been replaced by functions, as discussed in Section 5.3.2.

Linear and quadratic discriminant analysis

While probabilistic generative classification based on the normalized exponential and sigmoid can be applied to any set of log likelihoods, the formulas become much simpler when the distributions are multi-dimensional Gaussians.

For Gaussians with identical covariance matrices Σ , we have

$$p(\mathbf{x}|\mathcal{C}_k) = \frac{1}{(2\pi)^{D/2}} \frac{1}{\|\Sigma\|^{1/2}} \exp \left\{ -\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu}_k)^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}_k) \right\} \quad (5.8)$$

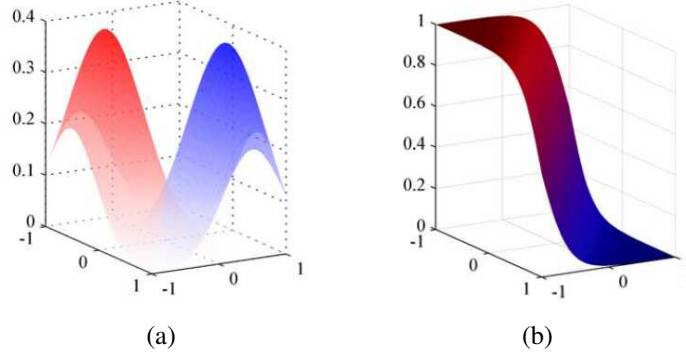


Figure 5.8 Logistic regression for two identically distributed Gaussian classes (*Bishop 2006*) © 2006 Springer: (a) two Gaussian distributions shown in red and blue; (b) the posterior probability $p(C_0|x)$, shown as both the height of the function and the proportion of red ink.

In the case of two classes (binary classification), we obtain (*Bishop 2006*, Section 4.2.1)

$$p(C_0|x) = \sigma(\mathbf{w}^T \mathbf{x} + b), \quad (5.9)$$

with

$$\mathbf{w} = \Sigma^{-1}(\boldsymbol{\mu}_0 - \boldsymbol{\mu}_1), \quad \text{and} \quad (5.10)$$

$$b = \frac{1}{2} \boldsymbol{\mu}_0^T \Sigma^{-1} \boldsymbol{\mu}_0 + \frac{1}{2} \boldsymbol{\mu}_1^T \Sigma^{-1} \boldsymbol{\mu}_1 + \log \frac{p(C_0)}{p(C_1)}. \quad (5.11)$$

Equation (5.9), which we will revisit shortly in the context of non-generative (discriminative) classification (5.18), is called *logistic regression*, since we pass the output of a linear regression formula

$$l(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b \quad (5.12)$$

through the logistic function to obtain a class probability. Figure 5.8 illustrates this in two dimensions, there the posterior likelihood of the red class $p(C_0|x)$ is shown on the right side.

In linear regression (5.12), $\mathbf{w}$ plays the role of the *weight* vector along which we project the feature vector $\mathbf{x}$, and b plays the role of the *bias*, which determines where to set the classification boundary. Note that the weight direction (5.10) aligns with the vector joining the distribution means (after rotating the coordinates by the inverse covariance Σ^{-1}), while the bias term is proportional to the mean squared moments and the log class prior ratio $\log(p(C_0)/p(C_1))$.

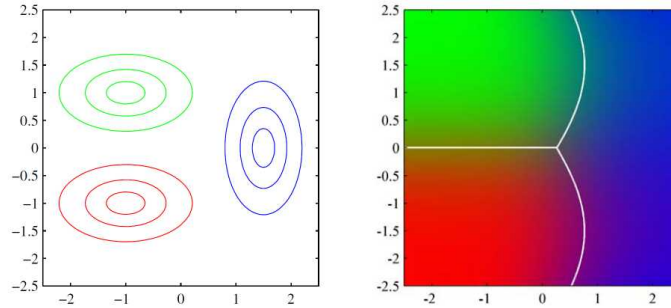


Figure 5.9 Quadratic discriminant analysis (Bishop 2006) © 2006 Springer. When the class covariances Σ_k are different, the decision surfaces between Gaussian distributions become quadratic surfaces.

For $K > 2$ classes, the softmax function (5.3) can be applied to the linear regression log likelihoods,

$$l_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} + b_k, \quad (5.13)$$

with

$$\mathbf{w}_k = \Sigma^{-1} \boldsymbol{\mu}_k, \quad \text{and} \quad (5.14)$$

$$b_k = -\frac{1}{2} \boldsymbol{\mu}_k^T \Sigma^{-1} \boldsymbol{\mu}_k + \log p(C_k). \quad (5.15)$$

Because the decision boundaries along which the classification switches from one class to another are linear,

$$\mathbf{w}_k \mathbf{x} + b_k > \mathbf{w}_l \mathbf{x} + b_l, \quad (5.16)$$

the technique of classifying examples using such criteria is known as *linear discriminant analysis* (Bishop 2006, Section 4.1; Murphy 2012, Section 4.2.2).⁸

Thus far, we have looked at the case where all of the class covariance matrices Σ_k are identical. When they vary between classes, the decision surfaces are no longer linear and they become quadratic (Figure 5.9). The derivation of these quadratic decision surfaces is known as *quadratic discriminant analysis* (Murphy 2012, Section 4.2.1).

In the case where Gaussian class distributions are not available, we can still find the best discriminant direction using Fisher discriminant analysis (Bishop 2006, Section 4.1.4; Murphy 2012, Section 8.6.3), as shown in Figure 5.10. Such analysis can be useful in separately

⁸The acronym LDA is commonly used with linear discriminant analysis, but is sometimes also used for *latent Dirichlet allocation* in graphical models.

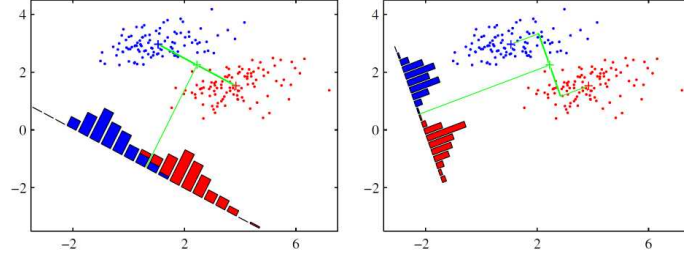


Figure 5.10 *Fisher linear discriminant (Bishop 2006) © 2006 Springer. To find the projection direction to best separate two classes, we compute the sum of the two class covariances and then use its inverse to rotate the vector between the two class means.*

modeling variability within different classes, e.g., the appearance variation of different people (Section 5.2.3).

5.1.3 Logistic regression

In the previous section, we derived classification rules based on posterior probabilities applied to multivariate Gaussian distributions. Quite often, however, Gaussians are not appropriate models of our class distributions and we must resort to alternative techniques.

One of the simplest among these is *logistic regression*, which applies the same ideas as in the previous section, i.e., a linear projection onto a weight vector,

$$l_i = \mathbf{w} \cdot \mathbf{x}_i + b \quad (5.17)$$

followed by a logistic function

$$p_i = p(C_0 | \mathbf{x}_i) = \sigma(l_i) = \sigma(\mathbf{w}^T \mathbf{x}_i + b) \quad (5.18)$$

to obtain (binary) class probabilities. Logistic regression is a simple example of a *discriminative model*, since it does not construct or assume a prior distribution over unknowns, and hence is not *generative*, i.e., we cannot generate random samples from the class (Bishop 2006, Section 1.5.4).

As we no longer have analytic estimates for the class means and covariances (or they are poor models of the class distributions), we need some other method to determine the weights $\mathbf{w}$ and bias b . We do this by maximizing the posterior log likelihoods of the correct labels.

For the binary classification task, let $t_i \in \{0, 1\}$ be the class label for each training sample $\mathbf{x}_i$ and $p_i = p(C_0 | \mathbf{x})$ be the estimated likelihood predicted by (5.18) for a given

weight and bias $(\mathbf{w}, b)$. We can maximize the likelihood of the correct labels being predicted by minimizing the negative log likelihood, i.e., the *cross-entropy loss* or error function,

$$E_{\text{CE}}(\mathbf{w}, b) = - \sum_i \{t_i \log p_i + (1 - t_i) \log(1 - p_i)\} \quad (5.19)$$

(Bishop 2006, Section 4.3.2).⁹ Note how whenever the label $t_i = 0$, we want $p_i = p(C_0|\mathbf{x}_i)$ to be high, and vice versa.

This formula can easily be extended to a multi-class loss by again defining the posterior probabilities as normalized exponentials over per-class linear regressions, as in (5.3) and (5.13),

$$p_{ik} = p(C_k|\mathbf{x}_i) = \frac{\exp l_{ik}}{\sum_j \exp l_{ij}} = \frac{1}{Z_i} \exp l_{ik}, \quad (5.20)$$

with

$$l_{ik} = \mathbf{w}_k^T \mathbf{x}_i + b_k. \quad (5.21)$$

The term $Z_i = \sum_j \exp l_{ij}$ can be a useful shorthand in derivations and is sometimes called the *partition function*. After some manipulation (Bishop 2006, Section 4.3.4), the corresponding *multi-class cross-entropy loss* (a.k.a. *multinomial logistic regression objective*) becomes

$$E_{\text{MCCE}}(\{\mathbf{w}_k, b_k\}) = - \sum_i \sum_k \tilde{t}_{ik} \log p_{ik}, \quad (5.22)$$

where the 1-of-K (or *one-hot*) encoding has $\tilde{t}_{ik} = 1$ if sample i belongs to class k (and 0 otherwise).¹⁰ It is more common to simply use the integer class value t_i as the target, in which case we can re-write this even more succinctly as

$$E(\{\mathbf{w}_k, b_k\}) = - \sum_i \log p_{it_i}, \quad (5.23)$$

i.e., we simply sum up the log likelihoods of the correct class for each training sample. Substituting the softmax formula (5.20) into this loss, we can re-write it as

$$E(\{\mathbf{w}_k, b_k\}) = \sum_i (\log Z_i - l_{it_i}). \quad (5.24)$$

⁹Note, however, that since this derivation is based on the assumption of Gaussian noise, it may not perform well if there are outliers, e.g., errors in the labels. In such a case, a more robust measure such as mean absolute error (MAE) may be preferable (Ghosh, Kumar, and Sastry 2017) or it may be necessary to re-weight the training samples (Ren, Zeng *et al.* 2018).

¹⁰This kind of representation can be useful if we wish the target classes to be a mixture, e.g., in the *mixup* data augmentation technique of Zhang, Cisse *et al.* (2018).

To determine the best set of weights and biases, $\{\mathbf{w}_k, b_k\}$, we can use gradient descent, i.e., update their values using a Newton-Raphson second-order optimization scheme (Bishop 2006, Section 4.3.3),

$$\mathbf{w} \leftarrow \mathbf{w} - \mathbf{H}^{-1} \nabla E(\mathbf{w}), \quad (5.25)$$

where ∇E is the gradient of the loss function E with respect to the weight variables $\mathbf{w}$, and $\mathbf{H}$ is the Hessian matrix of second derivatives of E . Because the cross-entropy functions are not linear in the unknown weights, we need to iteratively solve this equation a few times to arrive at a good solution. Since the elements in $\mathbf{H}$ are updated after each iteration, this technique is also known as *iteratively reweighted least squares*, which we will study in more detail in Section 8.1.4. While many non-linear optimization problems have multiple local minima, the cross-entropy functions described in this section do not, so we are guaranteed to arrive at a unique solution.

Logistic regression does have some limitations, which is why it is often used for only the simplest classification tasks. If the classes in feature space are not linearly separable, using simple projections onto weight vectors may not produce adequate decision surfaces. In this case, *kernel methods* (Sections 4.1.1 and 5.1.4; Bishop 2006, Chapter 6; Murphy 2012, Chapter 14), which measure the distances between new (test) feature vectors and select training examples, can often provide good solutions.

Another problem with logistic regression is that if the classes actually are separable (either in the original feature space, or the lifted kernel space), there can be more than a single unique separating plane, as illustrated in Figure 5.11a. Furthermore, unless regularized, the weights $\mathbf{w}$ will continue to grow larger, as larger values of $\mathbf{w}_k$ lead to larger p_{ik} values (once a separating plane has been found) and hence a smaller overall loss.

For this reason, techniques that place the decision surfaces in a way that maximizes their separation to labeled examples have been developed, as we discuss next.

5.1.4 Support vector machines

As we have just mentioned, in some applications of logistic regression we cannot determine a single optimal decision surface (choice of weight and bias vectors $\{\mathbf{w}_k, b_k\}$ in (5.21)) because there are *gaps* in the feature space where any number of planes could be introduced. Consider Figure 5.11a, where the two classes are denoted in cyan and magenta colors. In addition to the two dashed lines and the solid line, there are infinitely many other lines that will also cleanly separate the two classes, including a swath of horizontal lines. Since the classification error for any of these lines is zero, how can we choose the best decision surface, keeping in mind that we only have a limited number of training examples, and that actual run-time examples

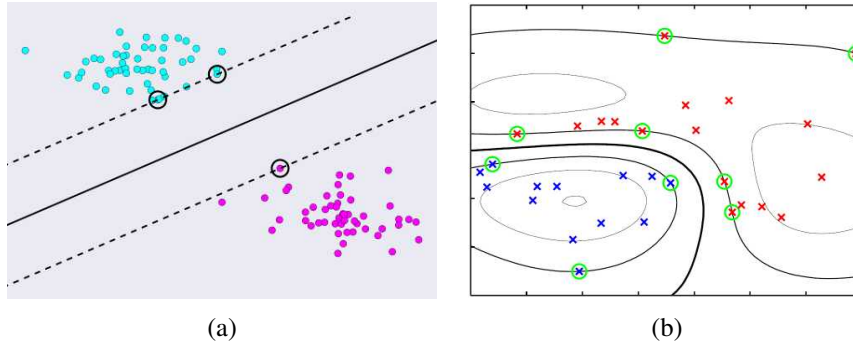


Figure 5.11 (a) A support vector machine (SVM) finds the linear decision surface (hyperplane) that maximizes the margin to the nearest training examples, which are called the support vectors © Glassner (2018). (b) A two-dimensional two class example of a Gaussian kernel support vector machine (Bishop 2006) © 2006 Springer. The red and blue $\times$ s indicate the training samples, and the samples circled in green are the support vectors. The black lines indicate iso-contours of the kernel regression function, with the contours containing the blue and red support vectors indicating the ± 1 contours and the dark contour in between being the decision surface.

may fall somewhere in between?

The answer to this problem is to use *maximum margin classifiers* (Bishop 2006, Section 7.1), as shown in Figure 5.11a, where the dashed lines indicate two parallel decision surfaces that have the *maximum margin*, i.e., the largest perpendicular distance between them. The solid line, which represents the hyperplane half-way between the dashed hyperplanes, is the maximum margin classifier.

Why is this a good idea? There are several potential derivations (Bishop 2006, Section 7.1), but a fairly intuitive explanation is that there may be real-world examples coming from the cyan and magenta classes that we have not yet seen. Under certain assumptions, the maximum margin classifier provides our best bet for correctly classifying as many of these unseen examples as possible.

To determine the maximum margin classifier, we need to find a weight-bias pair $(\mathbf{w}, b)$ for which all regression values $l_i = \mathbf{w} \cdot \mathbf{x}_i + b$ (5.17) have an absolute value of at least 1 as well as the correct sign. To denote this more compactly, let

$$\hat{t}_i = 2t_i - 1, \quad \hat{t}_i \in \{-1, 1\} \quad (5.26)$$

be the *signed class label*. We can now re-write the inequality condition as

$$\hat{t}_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1. \quad (5.27)$$

To maximize the margin, we simply find the smallest norm weight vector $\mathbf{w}$ that satisfies (5.27), i.e., we solve the optimization problem

$$\arg \min_{\mathbf{w}, b} \|\mathbf{w}\|^2 \quad (5.28)$$

subject to (5.27). This is a classic *quadratic programming problem*, which can be solved using the method of Lagrange multipliers, as described in Bishop (2006, Section 7.1).

The inequality constraints are exactly satisfied, i.e., they turn into equalities, along the two dashed lines in Figure 5.11a, where we have $l_i = \mathbf{w}\mathbf{x}_i + b = \pm 1$. The circled points that touch the dashed lines are called the *support vectors*.¹¹ For a simple linear classifier, which can be denoted with a single weight and bias pair $(\mathbf{w}, b)$, there is no real advantage to computing the support vectors, except that they help us estimate the decision surface. However, as we will shortly see, when we apply kernel regression, having a small number of support vectors is a huge advantage.

What happens if the two classes are not *linearly* separable, and in fact require a complex curved surface to correctly classify samples, as in Figure 5.11b? In this case, we can replace linear regression with *kernel regression* (4.3), which we introduced in Section 4.1.1. Rather than multiplying the weight vector $\mathbf{w}$ with the feature vector $\mathbf{x}$, we instead multiply it with the value of K kernel functions centered at the data point locations $\mathbf{x}_k$,

$$l_i = f(\mathbf{x}_i; \mathbf{w}, b) = \sum_k w_k \phi(\|\mathbf{x}_i - \mathbf{x}_k\|) + b. \quad (5.29)$$

This is where the power of support vector machines truly comes in.

Instead of requiring the summation over all training samples $\mathbf{x}_k$, once we solve for the maximum margin classifier only a small subset of support vectors needs to be retained, as shown by the circled crosses in Figure 5.11b. As you can see in this figure, the decision boundary denoted by the dark black line nicely separates the red and blue class samples. Note that as with other applications of kernel regression, the width of the radial basis functions is still a free hyperparameter that must be reasonably tuned to avoid underfitting and overfitting.

¹¹While the cyan and magenta dots may just look like points, they are, of course, schematic representations of higher-dimensional *vectors* lying in feature space.

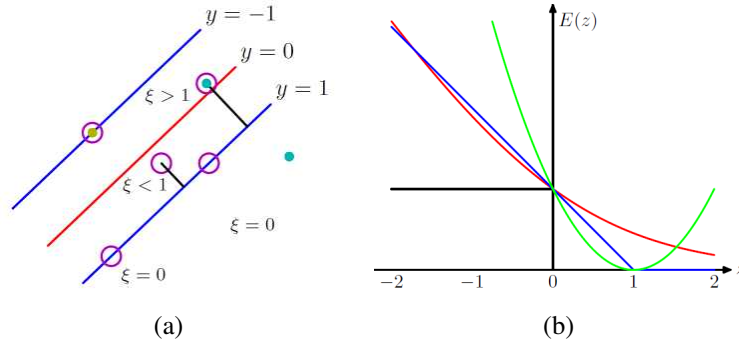


Figure 5.12 Support vector machine for overlapping class distributions (Bishop 2006) © 2006 Springer. (a) The green circled point is on the wrong side of the $y = 1$ decision contour and has a penalty of $\xi = 1 - y > 0$. (b) The “hinge” loss used in support vector machines is shown in blue, along with a rescaled version of the logistic regression loss function, shown in red, the misclassification error in black, and the squared error in green.

Hinge loss. So far, we have focused on classification problems that are separable, i.e., for which a decision boundary exists that correctly classifies all the training examples. Support vector machines can also be applied to overlapping (mixed) class distributions (Figure 5.12a), which we previously approached using logistic regression. In this case, we replace the inequality conditions (5.27), i.e., $\hat{t}_i l_i \geq 1$, with a *hinge loss* penalty

$$E_{\text{HL}}(l_i, \hat{t}_i) = [1 - \hat{t}_i l_i]_+, [1 - \hat{t}_i l_i]_+, \quad (5.30)$$

where $[\cdot]_+$ denotes the positive part, i.e. $[x]_+ = \max(0, x)$. The hinge loss penalty, shown in blue in Figure 5.12b, is 0 whenever the (previous) inequality is satisfied and ramps up linearly depending on how much the inequality is violated. To find the optimal weight values ($\mathbf{w}, b$), we minimize the regularized sum of hinge loss values,

$$E_{\text{SV}}(\mathbf{w}, b) = \sum_i E_{\text{HL}}(l_i(\mathbf{x}_i; \mathbf{w}, b), \hat{t}_i) + \lambda \|\mathbf{w}\|^2. \quad (5.31)$$

Figure 5.12b compares the hinge loss to the logistic regression (cross-entropy) loss in (5.19). The hinge loss imposes no penalty on training samples that are on the correct side of the $|l_i| > 1$ boundary, whereas the cross-entropy loss prefers larger absolute values. While, in this section, we have focused on the two-class version of support vector machines, Bishop (2006, Chapter 7) describes the extension to multiple classes as well as efficient optimization algorithms such as *sequential minimal optimization* (SMO) (Platt 1989). There’s also a nice

online tutorial on the scikit-learn website.¹² A survey of SVMs and other kernel methods applied to computer vision can be found in Lampert (2008).

5.1.5 Decision trees and forests

In contrast to most of the supervised learning techniques we have studied so far in this chapter, which process complete feature vectors all at once (with either linear projections or distances to training examples), *decision trees* perform a sequence of simpler operations, often just looking at individual feature elements before deciding which element to look at next (Hastie, Tibshirani, and Friedman 2009, Chapter 17; Glassner 2018, Section 14.5; Criminisi, Shotton, and Konukoglu 2012; Criminisi and Shotton 2013). (Note that the *boosting approaches* we study in Section 6.3.1 also use similar simple *decision stumps*.) While decision trees have been used in statistical machine learning for several decades (Breiman, Friedman *et al.* 1984), the application of their more powerful extension, namely *decision forests*, only started gaining traction in computer vision a little over a decade ago (Lepetit and Fua 2006; Shotton, Johnson, and Cipolla 2008; Shotton, Girshick *et al.* 2013). Decision trees, like support vector machines, are discriminative classifiers (or regressors), since they never explicitly form a probabilistic (generative) model of the data they are classifying.

Figure 5.13 illustrates the basic concepts behind decision trees and random forests. In this example, training samples come from four different classes, each shown in a different color (a). A decision tree (b) is constructed top-to-bottom by selecting decisions at each node that split the training samples that have made it to that node into more specific (lower entropy) distributions. The thickness of each link shows the number of samples that get classified along that path, and the color of the link is the blend of the class colors that flow through that link. The color histograms show the class distributions at a few of the interior nodes.

A random forest (c) is created by building a set of decision trees, each of which makes slightly different decisions. At test (classification) time, a new sample is classified by each of the trees in the random forest, and the class distributions at the final leaf nodes are averaged to provide an answer that is more accurate than could be obtained with a single tree (with a given depth).

Random forests have several design parameters, which can be used to tailor their accuracy, generalization, and run-time and space complexity. These parameters include:

- the depth of each tree D ,
- the number of trees T , and

¹²<https://scikit-learn.org/stable/modules/svm.html#svm-classification>

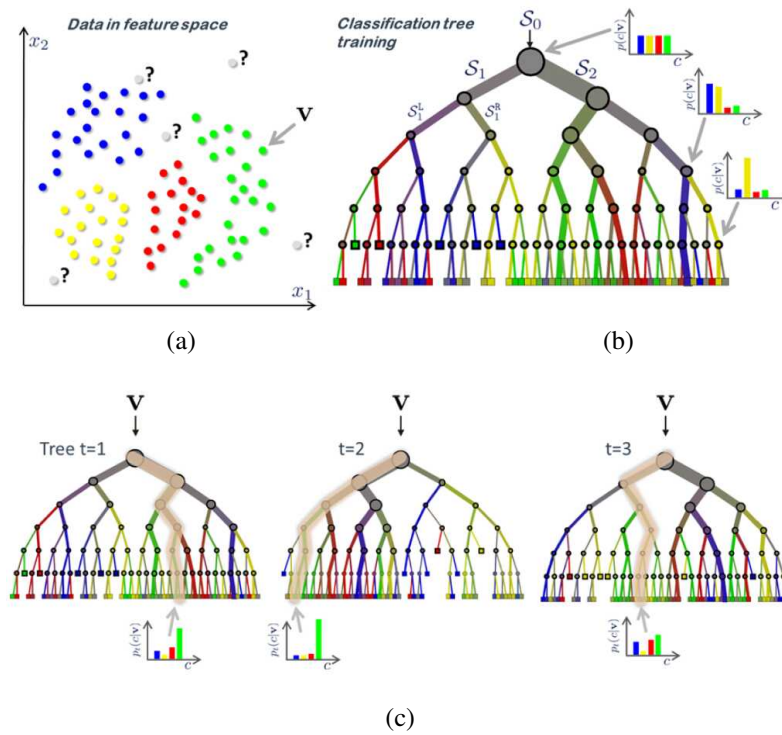


Figure 5.13 Decision trees and forests (Criminisi and Shotton 2013) © 2013 Springer. The top left figure (a) shows a set of training samples tags with four different class colors. The top right (b) shows a single decision tree with a distribution of classes at each node (the root node has the same distribution as the entire training set). During testing (c), each new example (feature vector) is tested at the root node, and depending on this test result (e.g., the comparison of some element to a threshold), a decision is made to walk down the tree to one of its children. This continues until a leaf node with a particular class distribution is reached. During training (b), decisions are selected such that they reduce the entropy (increase class specificity) at the node's children. The bottom diagram (c) shows an ensemble of three trees. After a particular test example has been classified by each tree, the class distributions of the leaf nodes of all the constituent trees are averaged.

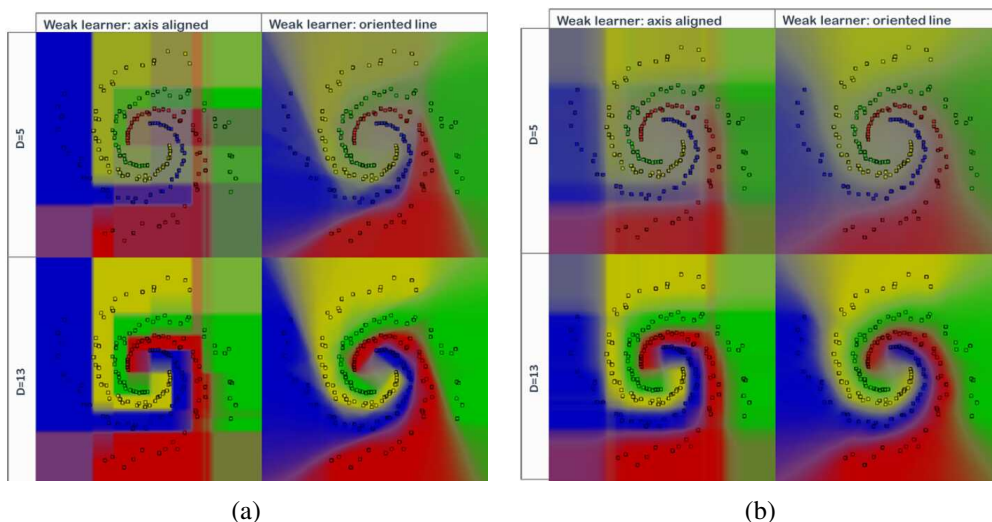


Figure 5.14 *Random forest decision surfaces (Criminisi and Shotton 2013) © 2013 Springer. Figures (a) and (b) show smaller and larger amounts of “noise” between the $T = 400$ tree forests obtained by using $\rho = 500$ and $\rho = 5$ random hypotheses at each split node. Within each figure, the two rows show trees of different depths ($D = 5$ and 13), while the columns show the effects of using axis-aligned or linear decision surfaces (“weak learners”).*

- the number of samples examined at node construction time ρ .

By only looking at a random subset ρ of all the training examples, each tree ends up having different decision functions at each node, so that the ensemble of trees can be averaged to produce softer decision boundaries.

Figure 5.14 shows the effects of some of these parameters on a simple four-class two-dimensional spiral dataset. In this figure, the number of trees has been fixed to $T = 400$. Criminisi and Shotton (2013, Chapter 4) have additional figures showing the effect of varying more parameters. The left (a) and right (b) halves of this figure show the effects of having less randomness ($\rho = 500$) and more randomness ($\rho = 5$) at the decision nodes. Less random trees produce sharper decision surfaces but may not generalize as well. Within each 2×2 grid of images, the top row shows a shallower $D = 5$ tree, while the bottom row shows a deeper $D = 13$ tree, which leads to finer details in the decision boundary. (As with all machine learning, better performance on training data may not lead to better generalization because of overfitting.) Finally, the right column shows what happens if axis-aligned (single element) decisions are replaced with linear combinations of feature elements.

When applied to computer vision, decision trees first made an impact in keypoint recognition (Lepetit and Fua 2006) and image segmentation (Shotton, Johnson, and Cipolla 2008). They were one of the key ingredients (along with massive amounts of synthetic training data) in the breakthrough success of human pose estimation from Kinect depth images (Shotton, Girshick *et al.* 2013). They also led to state-of-the-art medical image segmentation systems (Criminisi, Robertson *et al.* 2013), although these have now been supplanted by deep neural networks (Kamnitsas, Ferrante *et al.* 2016). Most of these applications, along with additional ones, are reviewed in the book edited by Criminisi and Shotton (2013).

5.2 Unsupervised learning

Thus far in this chapter, we have focused on *supervised learning* techniques where we are given training data consisting of paired input and target examples. In some applications, however, we are only given a set of data, which we wish to characterize, e.g., to see if there are any patterns, regularities, or typical distributions. This is typically the realm of classical statistics. In the machine learning community, this scenario is usually called *unsupervised learning*, since the sample data comes without labels. Examples of applications in computer vision include image segmentation (Section 7.5) and face and body recognition and reconstruction (Sections 13.6.2).

In this section, we look at some of the more widely used techniques in computer vision, namely clustering and mixture modeling (e.g., for segmentation) and principal component analysis (for appearance and shape modeling). Many other techniques are available, and are covered in textbooks on machine learning, such as Bishop (2006, Chapter 9), Hastie, Tibshirani, and Friedman (2009, Chapter 14), and Murphy (2012, Section 1.3).

5.2.1 Clustering

One of the simplest things you can do with your sample data is to group it into sets based on similarities (e.g., vector distances). In statistics, this problem is known as *cluster analysis* and is a widely studied area with hundreds of different algorithms (Jain and Dubes 1988; Kaufman and Rousseeuw 1990; Jain, Duin, and Mao 2000; Jain, Topchy *et al.* 2004). Murphy (2012, Chapter 25) has a nice exposition on clustering algorithms, including affinity propagation, spectral clustering, graph Laplacian, hierarchical, agglomerative, and divisive clustering. The survey by Xu and Wunsch (2005) is even more comprehensive, covering almost 300 different papers and such topics as similarity measures, vector quantization, mixture modeling, kernel methods, combinatorial and neural network algorithms, and visualization. Figure 5.15 shows

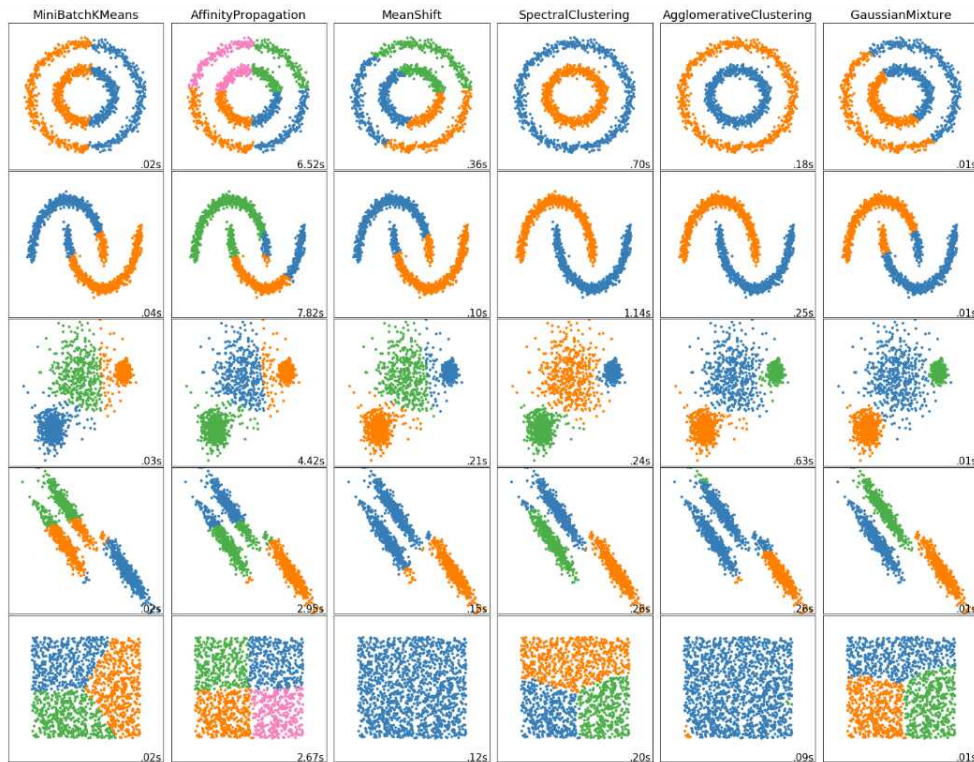


Figure 5.15 Comparison of different clustering algorithms on some toy datasets, generated using a simplified version of https://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_comparison.html#sphx-glr-auto-examples-cluster-plot-cluster-comparison-py.

some of the algorithms implemented in the <https://scikit-learn.org> cluster analysis package applied to some simple two-dimensional examples.

Splitting an image into successively finer regions (divisive clustering) is one of the oldest techniques in computer vision. Ohlander, Price, and Reddy (1978) present such a technique, which first computes a histogram for the whole image and then finds a threshold that best separates the large peaks in the histogram. This process is repeated until regions are either fairly uniform or below a certain size. More recent splitting algorithms often optimize some metric of intra-region similarity and inter-region dissimilarity. These are covered in Sections 7.5.3 and 4.3.2.

Region merging techniques also date back to the beginnings of computer vision. Brice and Fennema (1970) use a dual grid for representing boundaries between pixels and merge

regions based on their relative boundary lengths and the strength of the visible edges at these boundaries.

In data clustering, algorithms can link clusters together based on the distance between their closest points (single-link clustering), their farthest points (complete-link clustering), or something in between (Jain, Topchy *et al.* 2004). Kamvar, Klein, and Manning (2002) provide a probabilistic interpretation of these algorithms and show how additional models can be incorporated within this framework. Applications of such agglomerative clustering (region merging) algorithms to image segmentation are discussed in Section 7.5.

Mean-shift (Section 7.5.2) and mode finding techniques, such as k-means and mixtures of Gaussians, model the feature vectors associated with each pixel (e.g., color and position) as samples from an unknown probability density function and then try to find clusters (modes) in this distribution.

Consider the color image shown in Figure 7.53a. How would you segment this image based on color alone? Figure 7.53b shows the distribution of pixels in $L^*u^*v^*$ space, which is equivalent to what a vision algorithm that ignores spatial location would see. To make the visualization simpler, let us only consider the L^*u^* coordinates, as shown in Figure 7.53c. How many obvious (elongated) clusters do you see? How would you go about finding these clusters?

The k-means and mixtures of Gaussians techniques use a *parametric* model of the density function to answer this question, i.e., they assume the density is the superposition of a small number of simpler distributions (e.g., Gaussians) whose locations (centers) and shape (covariance) can be estimated. Mean shift, on the other hand, smoothes the distribution and finds its peaks as well as the regions of feature space that correspond to each peak. Since a complete density is being modeled, this approach is called *non-parametric* (Bishop 2006).

5.2.2 K-means and Gaussians mixture models

K-means implicitly model the probability density as a superposition of spherically symmetric distributions and does not require any probabilistic reasoning or modeling (Bishop 2006). Instead, the algorithm is given the number of clusters k it is supposed to find and is initialized by randomly sampling k centers from the input feature vectors. It then iteratively updates the cluster center location based on the samples that are closest to each center (Figure 5.16). Techniques have also been developed for splitting or merging cluster centers based on their statistics, and for accelerating the process of finding the nearest mean center (Bishop 2006).

In mixtures of Gaussians, each cluster center is augmented by a covariance matrix whose values are re-estimated from the corresponding samples (Figure 5.17). Instead of using nearest neighbors to associate input samples with cluster centers, a *Mahalanobis distance* (Ap-

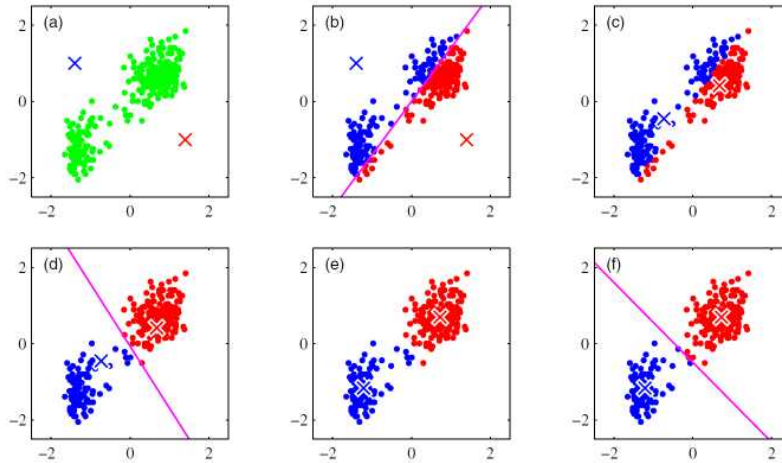


Figure 5.16 The *k*-means algorithm starts with a set of samples and the number of desired clusters (in this case, $k = 2$) (Bishop 2006) © 2006 Springer. It iteratively assigns samples to the nearest mean, and then re-computes the mean center until convergence.

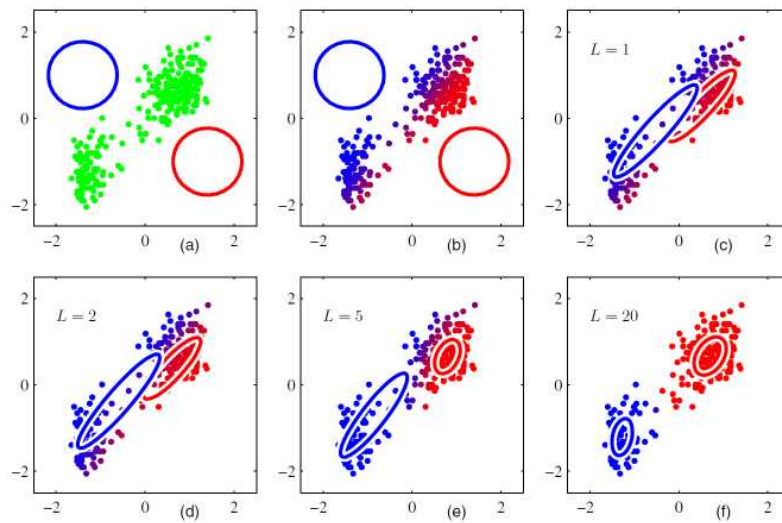


Figure 5.17 Gaussian mixture modeling (GMM) using expectation maximization (EM) (Bishop 2006) © 2006 Springer. Samples are softly assigned to cluster centers based on their Mahalanobis distance (inverse covariance weighted distance), and the new means and covariances are recomputed based on these weighted assignments.

pendix B.1) is used:

$$d(\mathbf{x}_i, \boldsymbol{\mu}_k; \boldsymbol{\Sigma}_k) = \|\mathbf{x}_i - \boldsymbol{\mu}_k\|_{\boldsymbol{\Sigma}_k^{-1}} = (\mathbf{x}_i - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_k) \quad (5.32)$$

where $\mathbf{x}_i$ are the input samples, $\boldsymbol{\mu}_k$ are the cluster centers, and $\boldsymbol{\Sigma}_k$ are their covariance estimates. Samples can be associated with the nearest cluster center (a *hard assignment* of membership) or can be *softly assigned* to several nearby clusters.

This latter, more commonly used, approach corresponds to iteratively re-estimating the parameters for a Gaussians mixture model,

$$p(\mathbf{x}|\{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}) = \sum_k \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \quad (5.33)$$

where π_k are the *mixing coefficients*, $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ are the Gaussian means and covariances, and

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{|\boldsymbol{\Sigma}_k|} e^{-d(\mathbf{x}, \boldsymbol{\mu}_k; \boldsymbol{\Sigma}_k)} \quad (5.34)$$

is the *normal* (Gaussian) distribution (Bishop 2006).

To iteratively compute (a local) maximum likely estimate for the unknown mixture parameters $\{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}$, the *expectation maximization* (EM) algorithm (Shlezinger 1968; Dempster, Laird, and Rubin 1977) proceeds in two alternating stages:

1. The *expectation* stage (E step) estimates the *responsibilities*

$$z_{ik} = \frac{1}{Z_i} \pi_k \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \quad \text{with} \quad \sum_k z_{ik} = 1, \quad (5.35)$$

which are the estimates of how likely a sample $\mathbf{x}_i$ was generated from the k th Gaussian cluster.

2. The *maximization* stage (M step) updates the parameter values

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_i z_{ik} \mathbf{x}_i, \quad (5.36)$$

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_i z_{ik} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T, \quad (5.37)$$

$$\pi_k = \frac{N_k}{N}, \quad (5.38)$$

where

$$N_k = \sum_i z_{ik}. \quad (5.39)$$

is an estimate of the number of sample points assigned to each cluster.

Bishop (2006) has a wonderful exposition of both mixture of Gaussians estimation and the more general topic of expectation maximization.

In the context of image segmentation, Ma, Derksen *et al.* (2007) present a nice review of segmentation using mixtures of Gaussians and develop their own extension based on Minimum Description Length (MDL) coding, which they show produces good results on the Berkeley segmentation dataset.

5.2.3 Principal component analysis

As we just saw in mixture analysis, modeling the samples within a cluster with a multivariate Gaussian can be a powerful way to capture their distribution. Unfortunately, as the dimensionality of our sample space increases, estimating the full covariance quickly becomes infeasible.

Consider, for example, the space of all frontal faces (Figure 5.18). For an image consisting of P pixels, the covariance matrix has a size of $P \times P$. Fortunately, the full covariance normally does not have to be modeled, since a lower-rank approximation can be estimated using *principal component analysis*, as described in Appendix A.1.2.

PCA was originally used in computer vision for modeling faces, i.e., *eigenfaces*, initially for gray-scale images (Kirby and Sirovich 1990; Turk and Pentland 1991), and then for 3D models (Blanz and Vetter 1999; Egger, Smith *et al.* 2020) (Section 13.6.2) and active appearance models (Section 6.2.4), where they were also used to model facial shape deformations (Rowland and Perrett 1995; Cootes, Edwards, and Taylor 2001; Matthews, Xiao, and Baker 2007).

Eigenfaces. Eigenfaces rely on the observation first made by Kirby and Sirovich (1990) that an arbitrary face image $\mathbf{x}$ can be compressed and reconstructed by starting with a mean image $\mathbf{m}$ (Figure 6.1b) and adding a small number of scaled signed images $\mathbf{u}_i$,

$$\tilde{\mathbf{x}} = \mathbf{m} + \sum_{i=0}^{M-1} a_i \mathbf{u}_i, \quad (5.40)$$

where the signed basis images (Figure 5.18b) can be derived from an ensemble of training images using *principal component analysis* (also known as *eigenvalue analysis* or the *Karhunen–Loève transform*). Turk and Pentland (1991) recognized that the coefficients a_i in the eigenface expansion could themselves be used to construct a fast image matching algorithm.

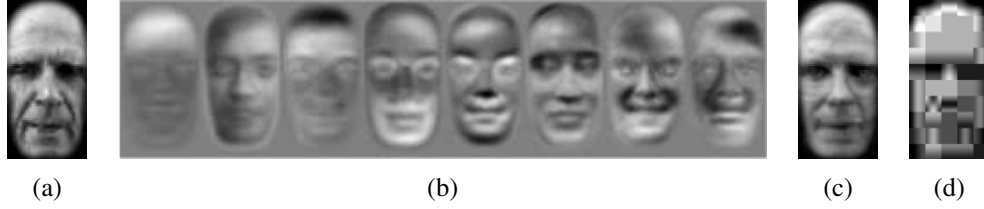


Figure 5.18 Face modeling and compression using eigenfaces (*Moghaddam and Pentland 1997*) © 1997 IEEE: (a) input image; (b) the first eight eigenfaces; (c) image reconstructed by projecting onto this basis and compressing the image to 85 bytes; (d) image reconstructed using JPEG (530 bytes).

In more detail, we start with a collection of *training images* $\{\mathbf{x}_j\}$, from which we compute the mean image $\mathbf{m}$ and a *scatter* or *covariance* matrix

$$\mathbf{C} = \frac{1}{N} \sum_{j=0}^{N-1} (\mathbf{x}_j - \mathbf{m})(\mathbf{x}_j - \mathbf{m})^T. \quad (5.41)$$

We can apply the eigenvalue decomposition (A.6) to represent this matrix as

$$\mathbf{C} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T = \sum_{i=0}^{N-1} \lambda_i \mathbf{u}_i \mathbf{u}_i^T, \quad (5.42)$$

where the λ_i are the eigenvalues of $\mathbf{C}$ and the $\mathbf{u}_i$ are the *eigenvectors*. For general images, Kirby and Sirovich (1990) call these vectors *eigenpictures*; for faces, Turk and Pentland (1991) call them *eigenfaces* (Figure 5.18b).¹³

Two important properties of the eigenvalue decomposition are that the optimal (best approximation) coefficients a_i for any new image $\mathbf{x}$ can be computed as

$$a_i = (\mathbf{x} - \mathbf{m}) \cdot \mathbf{u}_i, \quad (5.43)$$

and that, assuming the eigenvalues $\{\lambda_i\}$ are sorted in decreasing order, truncating the approximation given in (5.40) at any point M gives the best possible approximation (least error) between $\tilde{\mathbf{x}}$ and $\mathbf{x}$. Figure 5.18c shows the resulting approximation corresponding to Figure 5.18a and shows how much better it is at compressing a face image than JPEG.

Truncating the eigenface decomposition of a face image (5.40) after M components is equivalent to projecting the image onto a linear subspace F , which we can call the *face space*

¹³In actual practice, the full $P \times P$ scatter matrix (5.41) is never computed. Instead, a smaller $N \times N$ matrix consisting of the inner products between all the signed deviations $(\mathbf{x}_i - \mathbf{m})$ is accumulated instead. See Appendix A.1.2 (A.13–A.14) for details.

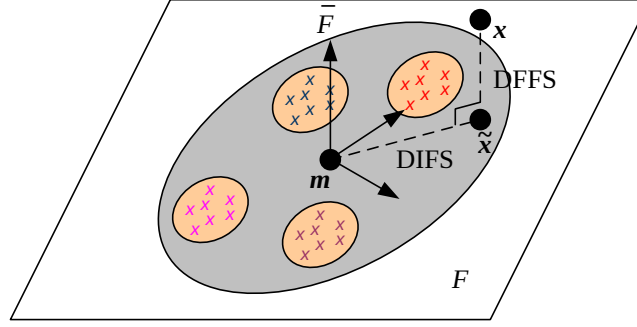


Figure 5.19 Projection onto the linear subspace spanned by the eigenface images (Moghaddam and Pentland 1997) © 1997 IEEE. The distance from face space (DFFS) is the orthogonal distance to the plane, while the distance in face space (DIFS) is the distance along the plane from the mean image. Both distances can be turned into Mahalanobis distances and given probabilistic interpretations.

(Figure 5.19). Because the eigenvectors (eigenfaces) are orthogonal and of unit norm, the distance of a projected face $\tilde{x}$ to the mean face $\mathbf{m}$ can be written as

$$\text{DIFS} = \|\tilde{x} - \mathbf{m}\| = \left[\sum_{i=0}^{M-1} a_i^2 \right]^{1/2}, \quad (5.44)$$

where DIFS stands for *distance in face space* (Moghaddam and Pentland 1997). The remaining distance between the original image x and its projection onto face space $\tilde{x}$, i.e., the *distance from face space* (DFFS), can be computed directly in pixel space and represents the “faceness” of a particular image. It is also possible to measure the distance between two different faces in face space by taking the norm of their eigenface coefficients difference.

Computing such distances in Euclidean vector space, however, does not exploit the additional information that the eigenvalue decomposition of the covariance matrix (5.42) provides. To properly weight the distance based on the measured covariance, we can use the *Mahalanobis distance* (5.32) (Appendix B.1). A similar analysis can be performed for computing a sensible difference from face space (DFFS) (Moghaddam and Pentland 1997) and the two terms can be combined to produce an estimate of the likelihood of being a true face, which can be useful in doing face detection (Section 6.3.1). More detailed explanations of probabilistic and Bayesian PCA can be found in textbooks on statistical learning (Bishop 2006; Hastie, Tibshirani, and Friedman 2009; Murphy 2012), which also discuss techniques for selecting the optimum number of components M to use in modeling a distribution.

The original work on eigenfaces for recognition (Turk and Pentland 1991) was extended in Moghaddam and Pentland (1997), Heisele, Ho *et al.* (2003), and Heisele, Serre, and Poggio (2007) to include *modular eigenspaces* for separately modeling the appearance of different facial components such as the eyes, nose, and mouth, as well as *view-based eigenspaces* to separately model different views of a face. It was also extended by Belhumeur, Hespanha, and Kriegman (1997) to handle appearance variation due to illumination, modeling *intrapersonal* and *extrapersonal* variability separately, and using Fisher linear discriminant analysis (Figure 5.10) to perform recognition. A Bayesian extension of this work was subsequently developed by Moghaddam, Jebara, and Pentland (2000). These extensions are described in more detail in the cited papers, as well as the first edition of this book (Szeliski 2010, Section 14.2).

It is also possible to generalize the bilinear factorization implicit in PCA and SVD approaches to multilinear (tensor) formulations that can model several interacting factors simultaneously (Vasilescu and Terzopoulos 2007). These ideas are related to additional topics in machine learning such as *subspace learning* (Cai, He *et al.* 2007), *local distance functions* (Frome, Singer *et al.* 2007; Ramanan and Baker 2009), and *metric learning* (Kulis 2013).

5.2.4 Manifold learning

In many cases, the data we are analyzing does not reside in a globally linear subspace, but does live on a lower-dimensional manifold. In this case, non-linear dimensionality reduction can be used (Lee and Verleysen 2007). Since these systems extract lower-dimensional manifolds in a higher-dimensional space, they are also known as *manifold learning* techniques (Zheng and Xue 2009). Figure 5.20 shows some examples of two-dimensional manifolds extracted from the three-dimensional S-shaped ribbon using the scikit-learn manifold learning package.¹⁴

These results are just a small sample from the large number of algorithms that have been developed, which include multidimensional scaling (Kruskal 1964a,b), Isomap (Tenenbaum, De Silva, and Langford 2000), Local Linear Embedding (Roweis and Saul 2000), Hessian Eigenmaps (Donoho and Grimes 2003), Laplacian Eigenmaps (Belkin and Niyogi 2003), local tangent space alignment (Zhang and Zha 2004), Dimensionality Reduction by Learning an Invariant Mapping (Hadsell, Chopra, and LeCun 2006), Modified LLE (Zhang and Wang 2007), t-distributed Stochastic Neighbor Embedding (t-SNE) (van der Maaten and Hinton 2008; van der Maaten 2014), and UMAP (McInnes, Healy, and Melville 2018). Many of these algorithms are reviewed in Lee and Verleysen (2007), Zheng and Xue (2009), and on

¹⁴<https://scikit-learn.org/stable/modules/manifold.html>

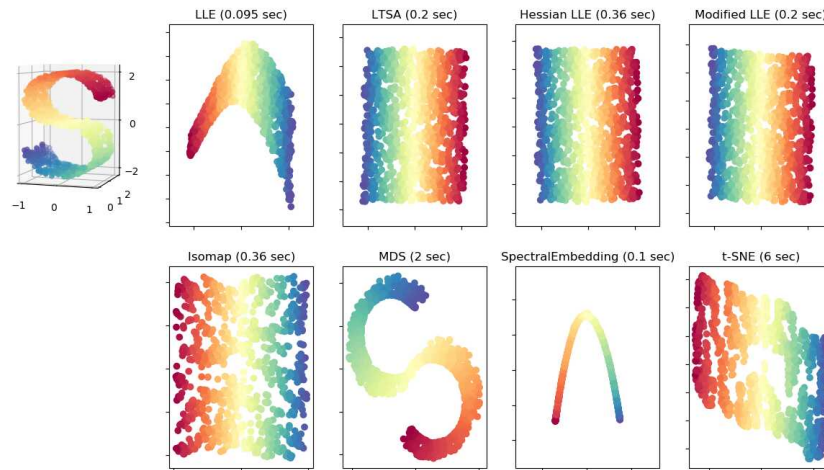


Figure 5.20 Examples of manifold learning, i.e., non-linear dimensionality reduction, applied to 1,000 points with 10 neighbors each, from <https://scikit-learn.org/stable/modules/manifold.html>. The eight sample outputs were produced by eight different embedding algorithms, as described in the *scikit-learn* manifold learning documentation page.

Wikipedia.¹⁵ Bengio, Paiement *et al.* (2004) describe a method for extending such algorithms to compute the embedding of new (“out-of-sample”) data points. McQueen, Meila *et al.* (2016) describe their megaman software package, which can efficiently solve embedding problems with millions of data points.

In addition to dimensionality reduction, which can be useful for regularizing data and accelerating similarity search, manifold learning algorithms can be used for visualizing input data distributions or neural network layer activations. Figure 5.21 show an example of applying two such algorithms (UMAP and t-SNE) to three different computer vision datasets.

5.2.5 Semi-supervised learning

In many machine learning settings, we have a modest amount of accurately labeled data and a far larger set of unlabeled or less accurate data. For example, an image classification dataset such as ImageNet may only contain one million labeled images, but the total number of images that can be found on the web is orders of magnitudes larger. Can we use this larger dataset, which still captures characteristics of our expect future inputs, to construct a better classifier or predictor?

¹⁵https://en.wikipedia.org/wiki/Nonlinear_dimensionality_reduction

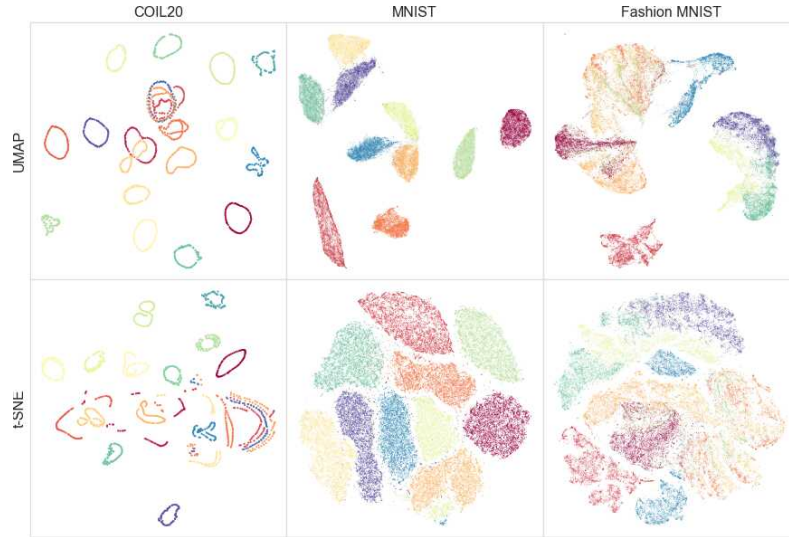


Figure 5.21 Comparison of UMAP and t-SNE manifold learning algorithms © McInnes, Healy, and Melville (2018) on three different computer vision learning recognition tasks: COIL (Nene, Nayar, and Murase 1996), MNIST (LeCun, Cortes, and Burges 1998), and Fashion MNIST (Xiao, Rasul, and Vollgraf 2017).

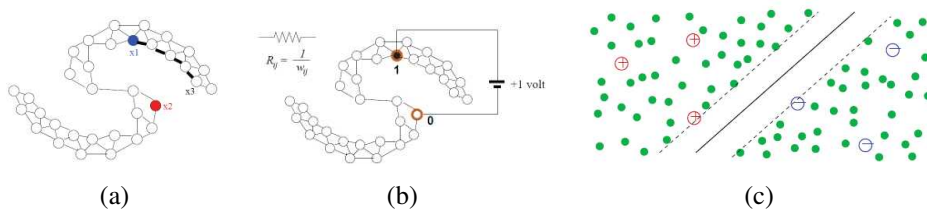


Figure 5.22 Examples of semi-supervised learning (Zhu and Goldberg 2009) © 2009 Morgan & Claypool: (a) two labeled samples and a graph connecting all of the samples; (b) solving binary labeling with harmonic functions, interpreted as a resistive electrical network; (c) using semi-supervised support vector machine (S3VM).

Consider the simple diagrams in Figure 5.22. Even if only a small number of examples are labeled with the correct class (in this case, indicated by red and blue circles or dots), we can still imagine extending these labels (inductively) to nearby samples and therefore not only labeling all of the data, but also constructing appropriate decision surfaces for future inputs.

This area of study is called *semi-supervised learning* (Zhu and Goldberg 2009; Subramanya and Talukdar 2014). In general, it comes in two varieties. In *transductive learning*, the goal is to classify all of the unlabeled inputs that are given as one batch at the same time as the labeled examples, i.e., all of the dots and circles shown in Figure 5.22. In *inductive learning*, we train a machine learning system that will classify all future inputs, i.e., all the regions in the input space. The second form is much more widely used, since in practice, most machine learning systems are used for online applications such as autonomous driving or new content classification.

Semi-supervised learning is a subset of the larger class of *weakly supervised learning* problems, where the training data may not only be missing labels, but also have labels of questionable accuracy (Zhou 2018). Some early examples from computer vision (Torresani 2014) include building whole image classifiers from image labels found on the internet (Fergus, Perona, and Zisserman 2004; Fergus, Weiss, and Torralba 2009) and object detection and/or segmentation (localization) with missing or very rough delineations in the training data (Nguyen, Torresani *et al.* 2009; Deselaers, Alexe, and Ferrari 2012). In the deep learning era, weakly supervised learning continues to be widely used (Pathak, Krahenbuhl, and Darrell 2015; Bilen and Vedaldi 2016; Arandjelovic, Gronat *et al.* 2016; Khoreva, Benenson *et al.* 2017; Novotny, Larlus, and Vedaldi 2017; Zhai, Oliver *et al.* 2019). A recent example of weakly supervised learning being applied to billions of noisily labeled images is pre-training deep neural networks on Instagram images with hashtags (Mahajan, Girshick *et al.* 2018). We will look at weakly and self-supervised learning techniques for *pre-training* neural networks in Section 5.4.7.

5.3 Deep neural networks

As we saw in the introduction to this chapter (Figure 5.2), deep learning pipelines take an end-to-end approach to machine learning, optimizing every stage of the processing by searching for parameters that minimize the training loss. In order for such search to be feasible, it helps if the loss is a differentiable function of all these parameters. Deep neural networks provide a uniform, differentiable computation architecture, while also automatically discovering useful internal representations.

Interest in building computing systems that mimic neural (biological) computation has

waxed and waned since the late 1950s, when Rosenblatt (1958) developed the *perceptron* and Widrow and Hoff (1960) derived the weight adaptation *delta rule*. Research into these topics was revitalized in the late 1970s by researchers who called themselves *connectionists*, organizing a series of meetings around this topic, which resulted in the foundation of the Neural Information Processing Systems (NeurIPS) conference in 1987. The recent book by Sejnowski (2018) has a nice historical review of this field’s development, as do the introductions in Goodfellow, Bengio, and Courville (2016) and Zhang, Lipton *et al.* (2021), the review paper by Rawat and Wang (2017), and the Turing Award lecture by Bengio, LeCun, and Hinton (2021). And while most of the deep learning community has moved away from biologically plausible models, some research still studies the connection between biological visual systems and neural network models (Yamins and DiCarlo 2016; Zhuang, Yan *et al.* 2020).

A good collection of papers from this era can be found in McClelland, Rumelhart, and PDP Research Group (1987), including the seminal paper on *backpropagation* (Rumelhart, Hinton, and Williams 1986a), which laid the foundation for the training of modern feedforward neural networks. During that time, and in the succeeding decades, a number of alternative neural network architectures were developed, including ones that used stochastic units such as Boltzmann Machines (Ackley, Hinton, and Sejnowski 1985) and Restricted Boltzmann Machines (Hinton and Salakhutdinov 2006; Salakhutdinov and Hinton 2009). The survey by Bengio (2009) has a review of some of these earlier approaches to deep learning. Many of these architectures are examples of the *generative graphical models* we saw in Section 4.3.

Today’s most popular deep neural networks are deterministic *discriminative feedforward networks* with real-valued activations, trained using gradient descent, i.e., the the backpropagation training rule (Rumelhart, Hinton, and Williams 1986b). When combined with ideas from convolutional networks (Fukushima 1980; LeCun, Bottou *et al.* 1998), deep multi-layer neural networks produced the breakthroughs in speech recognition (Hinton, Deng *et al.* 2012) and visual recognition (Krizhevsky, Sutskever, and Hinton 2012; Simonyan and Zisserman 2014b) seen in the early 2010s. Zhang, Lipton *et al.* (2021, Chapter 7) have a nice description of the components that went into these breakthroughs and the rapid evolution in deep networks that has occurred since then, as does the earlier review paper by (Rawat and Wang 2017).

Compared to other machine learning techniques, which normally rely on several pre-processing stages to extract features on which classifiers can be built, deep learning approaches are usually trained *end-to-end*, going directly from raw pixels to final desired outputs (be they classifications or other images). In the next few sections, we describe the basic

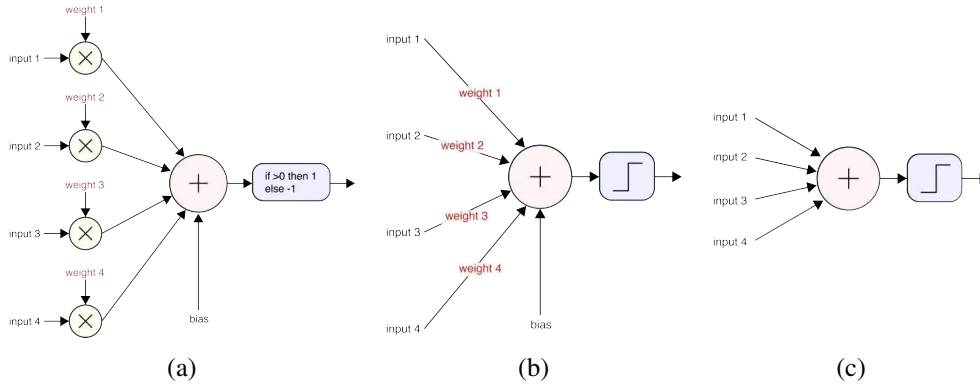


Figure 5.23 A perceptron unit (a) explicitly showing the weights being multiplied by the inputs, (b) with the weights written on the input connections, and (c) the most common form, with the weights and bias omitted. A non-linear activation function follows the weighted summation. © Glassner (2018)

components that go into constructing and training such neural networks. More detailed explanations on each topic can be found in textbooks on deep learning (Nielsen 2015; Goodfellow, Bengio, and Courville 2016; Glassner 2018, 2021; Zhang, Lipton *et al.* 2021) as well as the excellent course notes by Li, Johnson, and Yeung (2019) and Johnson (2020).

5.3.1 Weights and layers

Deep neural networks (DNNs) are *feedforward* computation graphs composed of thousands of simple interconnected “neurons” (*units*), which, much like logistic regression (5.18), perform weighted sums of their inputs

$$s_i = \mathbf{w}_i^T \mathbf{x}_i + b_i \quad (5.45)$$

followed by a non-linear *activation function* re-mapping,

$$y_i = h(s_i), \quad (5.46)$$

as illustrated in Figure 5.23. The $\mathbf{x}_i$ are the inputs to the i th unit, $\mathbf{w}_i$ and b_i are its learnable *weights* and *bias*, s_i is the output of the weighted linear sum, and y_i is the final output after s_i is fed through the activation function h .¹⁶ The outputs of each stage, which are often called

¹⁶Note that we have switched to using s_i for the weighted summations, since we will want to use l to index neural network layers.

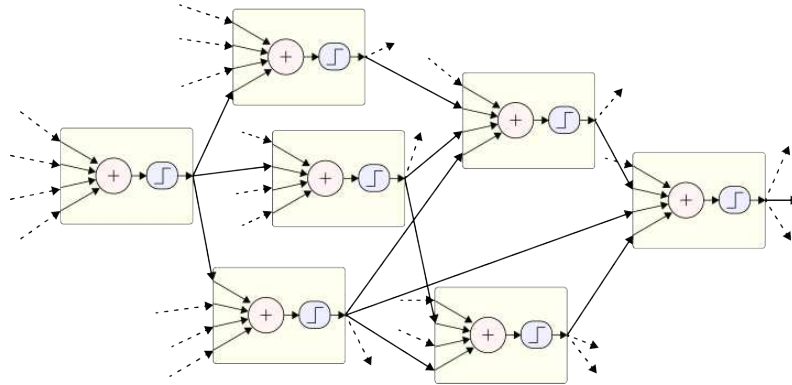


Figure 5.24 A multi-layer network, showing how the outputs of one unit are fed into additional units. © Glassner (2018)

the *activations*, are then fed into units in later stages, as shown in Figure 5.24.¹⁷

The earliest such units were called *perceptrons* (Rosenblatt 1958) and were diagrammed as shown in Figure 5.23a. Note that in this first diagram, the weights, which are optimized during the learning phase (Section 5.3.5), are shown explicitly along with the element-wise multiplications. Figure 5.23b shows a form in which the weights are written on top of the *connections* (arrows between units, although the arrowheads are often omitted). It is even more common to diagram nets as in Figure 5.23c, in which the weights (and bias) are completely omitted and assumed to be present.

Instead of being connected into an irregular computation graph as in Figure 5.24, neural networks are usually organized into consecutive *layers*, as shown in Figure 5.25. We can now think of all the units within a layer as being a vector, with the corresponding linear combinations written as

$$\mathbf{s}_l = \mathbf{W}_l \mathbf{x}_l, \quad (5.47)$$

where $\mathbf{x}_l$ are the inputs to layer l , $\mathbf{W}_l$ is a weight matrix, and $\mathbf{s}_l$ is the weighted sum, to which an element-wise non-linearity is applied using a set of activation functions,

$$\mathbf{x}_{l+1} = \mathbf{y}_l = \mathbf{h}(\mathbf{s}_l). \quad (5.48)$$

A layer in which a full (dense) weight matrix is used for the linear combination is called a *fully connected* (FC) layer, since all of the inputs to one layer are connected to all of its

¹⁷Note that while almost all feedforward neural networks use linear weighted summations of their inputs, the Neocognitron (Fukushima 1980) also included a divisive normalization stage inspired by the behavior of biological neurons. Some of the latest DNNs also support multiplicative interactions between activations using *conditional batch norm* (Section 5.3.3).

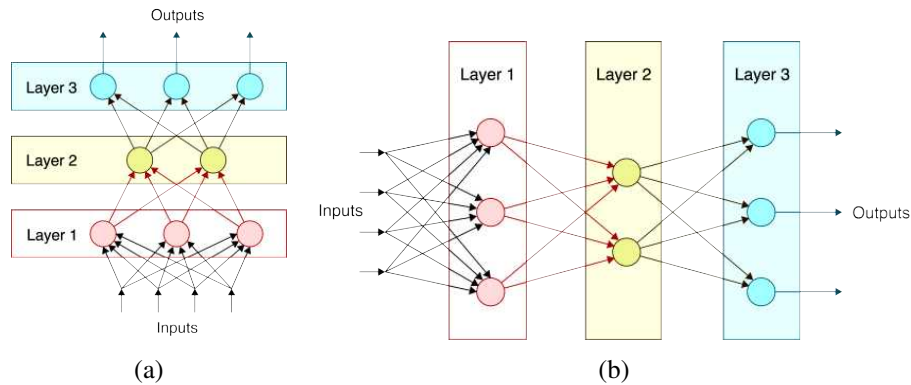


Figure 5.25 Two different ways to draw neural networks: (a) inputs at bottom, outputs at top, (b) inputs at left, outputs at right. © Glassner (2018)

outputs. As we will see in Section 5.4, when processing pixels (or other signals), early stages of processing use convolutions instead of dense connections for both spatial invariance and better efficiency.¹⁸ A network that consists only of fully connected (and no convolutional) layers is now often called a *multi-layer perceptron* (MLP).

5.3.2 Activation functions

Most early neural networks (Rumelhart, Hinton, and Williams 1986b; LeCun, Bottou *et al.* 1998) used sigmoidal functions similar to the ones used in logistic regression. Newer networks, starting with Nair and Hinton (2010) and Krizhevsky, Sutskever, and Hinton (2012), use Rectified Linear Units (ReLU) or variants. The ReLU activation function is defined as

$$h(y) = \max(0, y) \quad (5.49)$$

and is shown in the upper-left corner of Figure 5.26, along with some other popular functions, whose definitions can be found in a variety of publications (e.g., Goodfellow, Bengio, and Courville 2016, Section 6.3; Clevert, Unterthiner, and Hochreiter 2015; He, Zhang *et al.* 2015) and the Machine Learning Cheatsheet.¹⁹

While the ReLU is currently the most popular activation function, a widely cited observation in the CS231N course notes (Li, Johnson, and Yeung 2019) attributed to Andrej Karpathy

¹⁸Heads up for more confusing abbreviations: While a fully connected (dense) layer is often abbreviated as FC, a fully convolutional network, which is the opposite, i.e., sparsely connected with shared weights, is often abbreviated as FCN.

¹⁹https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html

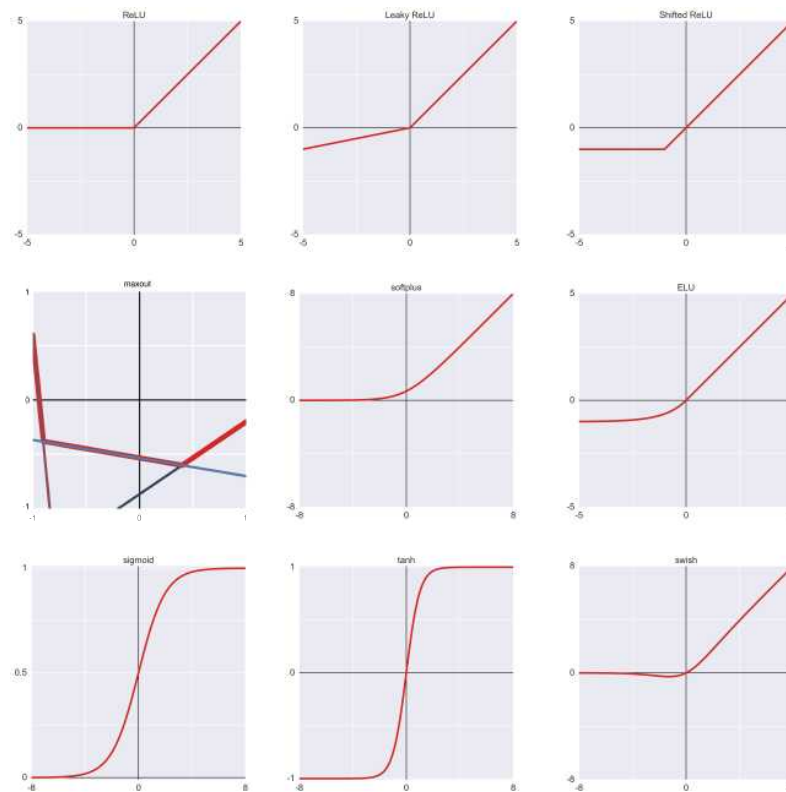


Figure 5.26 Some popular non-linear activation functions from © Glassner (2018): From top-left to bottom-right: ReLU, leaky ReLU, shifted ReLU, maxout, softplus, ELU, sigmoid, tanh, swish.

warns that²⁰

Unfortunately, ReLU units can be fragile during training and can “die”. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any data-point again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. ... With a proper setting of the learning rate this is less frequently an issue.

The CS231n course notes advocate trying some alternative non-clipping activation functions

²⁰<http://cs231n.github.io/neural-networks-1/#actfun>

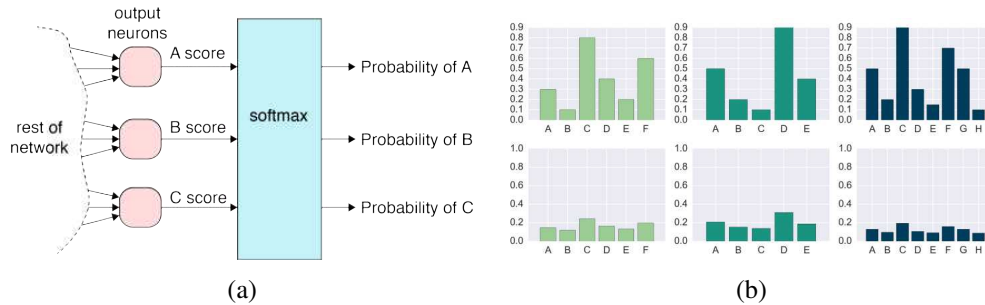


Figure 5.27 (a) A softmax layer used to convert from neural network activations (“score”) to class likelihoods (b) The top row shows the activations, while the bottom shows the result of running the scores through softmax to obtain properly normalized likelihoods. © Glassner (2018).

if this problem arises.

For the final layer in networks used for classification, the softmax function (5.3) is normally used to convert from real-valued activations to class likelihoods, as shown in Figure 5.27. We can thus think of the penultimate set of neurons as determining directions in activation space that most closely match the log likelihoods of their corresponding class, while minimizing the log likelihoods of alternative classes. Since the inputs flow forward to the final output classes and probabilities, feedforward networks are discriminative, i.e., they have no statistical model of the classes they are outputting, nor any straightforward way to generate samples from such classes (but see Section 5.5.4 for techniques to do this).

5.3.3 Regularization and normalization

As with other forms of machine learning, regularization and other techniques can be used to prevent neural networks from overfitting so they can better generalize to unseen data. In this section, we discuss traditional methods such as regularization and data augmentation that can be applied to most machine learning systems, as well as techniques such as dropout and batch normalization, which are specific to neural networks.

Regularization and weight decay

As we saw in Section 4.1.1, quadratic or p -norm penalties on the weights (4.9) can be used to improve the conditioning of the system and to reduce overfitting. Setting $p = 2$ results in the usual L_2 regularization and makes large weights smaller, whereas using $p = 1$ is called

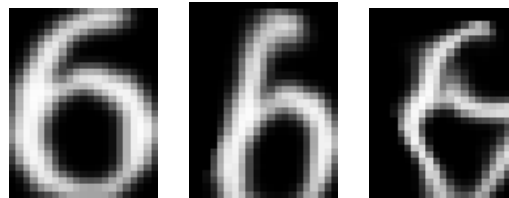


Figure 5.28 An original “6” digit from the MNIST database and two elastically distorted versions (Simard, Steinkraus, and Platt 2003) © 2003 IEEE.

lasso (least absolute shrinkage and selection operator) and can drive some weights all the way to zero. As the weights are being optimized inside a neural network, these terms make the weights smaller, so this kind of regularization is also known as *weight decay* (Bishop 2006, Section 3.1.4; Goodfellow, Bengio, and Courville 2016, Section 7.1; Zhang, Lipton *et al.* 2021, Section 4.5).²¹ Note that for more complex optimization algorithms such as Adam, L_2 regularization and weight decay are *not* equivalent, but the desirable properties of weight decay can be restored using a modified algorithm (Loshchilov and Hutter 2019).

Dataset augmentation

Another powerful technique to reduce over-fitting is to add more training samples by perturbing the inputs and/or outputs of the samples that have already been collected. This technique is known as *dataset augmentation* (Zhang, Lipton *et al.* 2021, Section 13.1) and can be particularly effective on image classification tasks, since it is expensive to obtain labeled examples, and also since image classes should not change under small local perturbations.

An early example of such work applied to a neural network classification task is the *elastic distortion* technique proposed by Simard, Steinkraus, and Platt (2003). In their approach, random low-frequency displacement (warp) fields are synthetically generated for each training example and applied to the inputs during training (Figure 5.28). Note how such distortions are *not* the same as simply adding pixel noise to the inputs. Instead, distortions move pixels around, and therefore introduce much larger changes in the input vector space, while still preserving the semantic meaning of the examples (in this case, MNIST digits (LeCun, Cortes, and Burges 1998)).

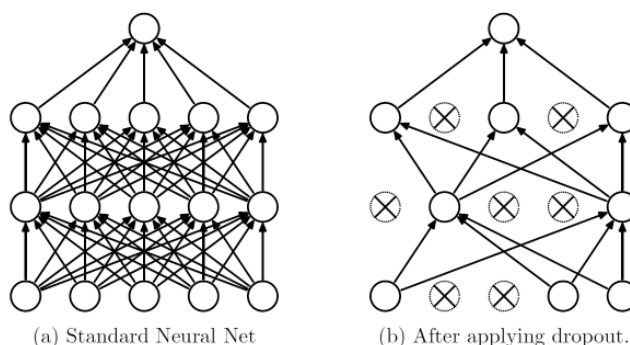


Figure 5.29 When using dropout, during training some fraction of units p is removed from the network (or, equivalently, clamped to zero) © Srivastava, Hinton *et al.* (2014). Doing this randomly for each mini-batch injects noise into the training process (at all levels of the network) and prevents the network from overly relying on particular units.

Dropout

Dropout is a regularization technique introduced by Srivastava, Hinton *et al.* (2014), where at each mini-batch during training (Section 5.3.6), some percentage p (say 50%) of the units in each layer are clamped to zero, as shown in Figure 5.29. Randomly setting units to zero injects noise into the training process and also prevents the network from overly specializing units to particular samples or tasks, both of which can help reduce overfitting and improve generalization.

Because dropping (zeroing out) p of the units reduces the expected value of any sum the unit contributes to by a fraction $(1 - p)$, the weighted sums s_i in each layer (5.45) are multiplied (during training) by $(1 - p)^{-1}$. At test time, the network is run with no dropout and no compensation on the sums. A more detailed description of dropout can be found in Zhang, Lipton *et al.* (2021, Section 4.6) and Johnson (2020, Lecture 10).

Batch normalization

Optimizing the weights in a deep neural network, which we discuss in more detail in Section 5.3.6, is a tricky process and may be slow to converge.

One of the classic problems with iterative optimization techniques is poor *conditioning*, where the components of the gradient vary greatly in magnitude. While it is sometimes

²¹From a Bayesian perspective, we can also think of this penalty as a Gaussian prior on the weight distribution (Appendix B.4).

possible to reduce these effects with preconditioning techniques that scale individual elements in a gradient before taking a step (Section 5.3.6 and Appendix A.5.2), it is usually preferable to control the condition number of the system during the problem formulation.

In deep networks, one way in which poor conditioning can manifest itself is if the sizes of the weights or activations in successive layers become imbalanced. Say we take a given network and scale all of the weights in one layer by $100\times$ and scale down the weights in the next layer by the same amount. Because the ReLU activation function is linear in both of its domains, the outputs of the second layer will still be the same, although the activations at the output of the first layer will be 100 times larger. During the gradient descent step, the derivatives with respect to the weights will be vastly different after this rescaling, and will in fact be opposite in magnitude to the weights themselves, requiring tiny gradient descent steps to prevent overshooting (see Exercise 5.4).²²

The idea behind batch normalization (Ioffe and Szegedy 2015) is to re-scale (and re-center) the activations at a given unit so that they have unit variance and zero mean (which, for a ReLU activation function, means that the unit will be active half the time). We perform this normalization by considering all of the training samples n in a given minibatch $\mathcal{B}$ (5.71) and computing the mean and variance statistics for unit i as

$$\mu_i = \frac{1}{|\mathcal{B}|} \sum_{n \in \mathcal{B}} s_i^{(n)} \quad (5.50)$$

$$\sigma_i^2 = \frac{1}{|\mathcal{B}|} \sum_{n \in \mathcal{B}} (s_i^{(n)} - \mu_i)^2 \quad (5.51)$$

$$\hat{s}_i^{(n)} = \frac{s_i^{(n)} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}, \quad (5.52)$$

where $s_i^{(n)}$ is the weighted sum of unit i for training sample n , $\hat{s}_i^{(n)}$ is the corresponding batch normalized sum, and ϵ (often 10^{-5}) is a small constant to prevent division by zero.

After batch normalization, the $\hat{s}_i^{(n)}$ activations now have zero mean and unit variance. However, this normalization may run at cross-purpose to the minimization of the loss function during training. For this reason, Ioffe and Szegedy (2015) add an extra gain γ_i and bias β_i parameter to each unit i and define the output of a batch normalization stage to be

$$y_i = \gamma_i \hat{s}_i + \beta_i. \quad (5.53)$$

²²This motivating paragraph is my own explanation of why batch normalization might be a good idea, and is related to the idea that batch normalization reduces *internal covariate shift*, used by (Ioffe and Szegedy 2015) to justify their technique. This hypothesis is now being questioned and alternative theories are being developed (Bjorck, Gomes *et al.* 2018; Santurkar, Tsipras *et al.* 2018; Kohler, Daneshmand *et al.* 2019).

These parameters act just like regular weights, i.e., they are modified using gradient descent during training to reduce the overall training loss.²³

One subtlety with batch normalization is that the μ_i and σ_i^2 quantities depend analytically on all of the activation for a given unit in a minibatch. For gradient descent to be properly defined, the derivatives of the loss function with respect to these variables, and the derivatives of the quantities $\hat{s}_i$ and y_i with respect to these variables, must be computed as part of the gradient computation step, using similar chain rule computations as the original backpropagation algorithm (5.65–5.68). These derivations can be found in Ioffe and Szegedy (2015) as well as several blogs.²⁴

When batch normalization is applied to convolutional layers (Section 5.4), one could in principle compute a normalization separately for each pixel, but this would add a tremendous number of extra learnable bias and gain parameters (β_i, γ_i) . Instead, batch normalization is usually implemented by computing the statistics as sums over all the pixels with the same convolution kernel, and then adding a single bias and gain parameter for each convolution kernel (Ioffe and Szegedy 2015; Johnson 2020, Lecture 10; Zhang, Lipton *et al.* 2021, Section 7.5).

Having described how batch normalization operates during training, we still need to decide what to do at test or inference time, i.e., when applying the trained network to new data. We cannot simply skip this stage, as the network was trained while removing common mean and variance estimates. For this reason, the mean and variance estimates are usually recomputed over the *whole* training set, or some running average of the per-batch statistics are used. Because of the linear form of (5.45) and (5.52–5.53), it is possible to fold the μ_i and σ_i estimates and learned (β_i, γ_i) parameters into the original weight and bias terms in (5.45).

Since the publication of the seminal paper by Ioffe and Szegedy (2015), a number of variants have been developed, some of which are illustrated in Figure 5.30. Instead of accumulating statistics over the samples in a minibatch $\mathcal{B}$, we can compute them over different subsets of activations in a layer. These subsets include:

- all the activations in a layer, which is called *layer normalization* (Ba, Kiros, and Hinton 2016);
- all the activations in a given convolutional output channel (see Section 5.4), which is called *instance normalization* (Ulyanov, Vedaldi, and Lempitsky 2017);

²³There is a trick used by those in the know, which relies on the observation that any bias term b_i in the original summation s_i (5.45) shows up in the mean μ_i and gets subtracted out. For this reason, the bias term is often omitted when using batch (or other kinds of) normalization.

²⁴<https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html>, https://kevinzakka.github.io/2016/09/14/batch_normalization, <https://deeppoints.io/batchnorm>

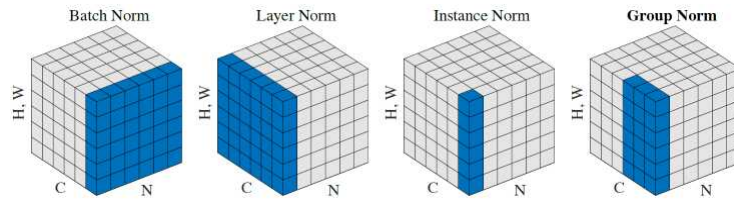


Figure 5.30 *Batch norm, layer norm, instance norm, and group norm, from Wu and He (2018) © 2018 Springer. The (H, W) dimension denotes pixels, C denotes channels, and N denotes training samples in a minibatch. The pixels in blue are normalized by the same mean and variance.*

- different sub-groups of output channels, which is called *group normalization* (Wu and He 2018).

The paper by Wu and He (2018) describes each of these in more detail and also compares them experimentally. More recent work by Qiao, Wang *et al.* (2019a) and Qiao, Wang *et al.* (2019b) discusses some of the disadvantages of these newer variants and proposes two new techniques called *weight standardization* and *batch channel normalization* to mitigate these problems.

Instead of modifying the activations in a layer using their statistics, it is also possible to modify the weights in a layer to explicitly make the weight norm and weight vector direction separate parameters, which is called *weight normalization* (Salimans and Kingma 2016). A related technique called *spectral normalization* (Miyato, Kataoka *et al.* 2018) constrains the largest singular value of the weight matrix in each layer to be 1.

The bias and gain parameters (β_i, γ_i) may also depend on the activations in some other layer in the network, e.g., derived from a guide image.²⁵ Such techniques are referred to as *conditional batch normalization* and have been used to select between different artistic styles (Dumoulin, Shlens, and Kudlur 2017) and to enable local semantic guidance in image synthesis (Park, Liu *et al.* 2019). Related techniques and applications are discussed in more detail in Section 14.6 on neural rendering.

The reasons why batch and other kinds of normalization help deep networks converge faster and generalize better are still being debated. Some recent papers on this topic include Bjorck, Gomes *et al.* (2018), Hoffer, Banner *et al.* (2018), Santurkar, Tsipras *et al.* (2018), and Kohler, Daneshmand *et al.* (2019).

²⁵Note that this gives neural networks the ability to multiply two layers in a network, which we used previously to perform locally (Section 3.5.5).

5.3.4 Loss functions

In order to optimize the weights in a neural network, we need to first define a *loss function* that we minimize over the training examples. We have already seen the main loss functions used in machine learning in previous parts of this chapter.

For classification, most neural networks use a final softmax layer (5.3), as shown in Figure 5.27. Since the outputs are meant to be class probabilities that sum up to 1, it is natural to use the cross-entropy loss given in (5.19) or (5.23–5.24) as the function to minimize during training. Since in our description of the feedforward networks we have used indices i and j to denote neural units, we will, in this section, use n to index a particular training example.

The multi-class cross-entropy loss can thus be re-written as

$$E(\mathbf{w}) = \sum_n E_n(\mathbf{w}) = - \sum_n \log p_{nt_n}, \quad (5.54)$$

where $\mathbf{w}$ is the vector of all weights, biases, and other model parameters, and p_{nk} is the network's current estimate of the probability of class k for sample n , and t_n is the integer denoting the correct class. Substituting the definition of p_{nk} from (5.20) with the appropriate replacement of l_{ik} with s_{nk} (the notation we use for neural nets), we get

$$E_n(\mathbf{w}) = \log Z_n - s_{nt_n} \quad (5.55)$$

with $Z_n = \sum_j \exp s_{nj}$. Gómez (2018) has a nice discussion of some of the losses widely used in deep learning.

For networks that perform regression, i.e., generate one or more continuous variables such as depth maps or denoised images, it is common to use an L_2 loss,

$$E(\mathbf{w}) = \sum_n E_n(\mathbf{w}) = - \sum_n \|\mathbf{y}_n - \mathbf{t}_n\|^2, \quad (5.56)$$

where $\mathbf{y}_n$ is the network output for sample n and $\mathbf{t}_n$ is the corresponding training (target) value, since this is a natural measure of error between continuous variables. However, if we believe there may be outliers in the training data, or if gross errors are not so harmful as to merit a quadratic penalty, more robust norms such as L_1 can be used (Barron 2019; Ranftl, Lasinger *et al.* 2020). (It is also possible to use robust norms for classification, e.g., adding an outlier probability to the class labels.)

As it is common to interpret the final outputs of a network as a probability distribution, we need to ask whether it is wise to use such probabilities as a measure of confidence in a particular answer. If a network is properly trained and predicting answers with good accuracy, it is tempting to make this assumption. The training losses we have presented so far, however,

only encourage the network to maximize the probability-weighted correct answers, and do not, in fact, encourage the network outputs to be properly *confidence calibrated*. Guo, Pleiss *et al.* (2017) discuss this issue, and present some simple measures, such as multiplying the log-likelihoods by a *temperature* (Platt 2000a), to improve the match between classifier probabilities and true reliability. The GrokNet image recognition system (Bell, Liu *et al.* 2020), which we discuss in Section 6.2.3, uses calibration to obtain better attribute probability estimates.

For networks that hallucinate new images, e.g., when introducing missing high-frequency details (Section 10.3) or doing image transfer tasks (Section 14.6), we may want to use a *perceptual loss* (Johnson, Alahi, and Fei-Fei 2016; Dosovitskiy and Brox 2016; Zhang, Isola *et al.* 2018), which uses intermediate layer neural network responses as the basis of comparison between target and output images. It is also possible to train a separate *discriminator* network to evaluate the quality (and plausibility) of synthesized images, as discussed in Section 5.5.4. More details on the application of loss functions to image synthesis can be found in Section 14.6 on neural rendering.

While loss functions are traditionally applied to supervised learning tasks, where the correct label or target value $\mathbf{t}_n$ is given for each input, it is also possible to use loss functions in an unsupervised setting. An early example of this was the *contrastive loss* function proposed by Hadsell, Chopra, and LeCun (2006) to cluster samples that are similar together while spreading dissimilar samples further apart. More formally, we are given a set of inputs $\{\mathbf{x}_i\}$ and pairwise indicator variables $\{t_{ij}\}$ that indicate whether two inputs are similar.²⁶ The goal is now to compute an embedding $\mathbf{v}_i$ for each input $\mathbf{x}_i$ such that similar input pairs have similar embeddings (low distances), while dissimilar inputs have large embedding distances. Finding mappings or embeddings that create useful distances between samples is known as (*distance*) *metric learning* (Köstinger, Hirzer *et al.* 2012; Kulis 2013) and is a commonly used tool in machine learning. The losses used to encourage the creation of such meaningful distances are collectively known as *ranking losses* (Gómez 2019) and can be used to relate features from different domains such as text and images (Karpathy, Joulin, and Fei-Fei 2014).

The contrastive loss from (Hadsell, Chopra, and LeCun 2006) is defined as

$$E_{\text{CL}} = \sum_{(i,j) \in \mathcal{P}} \{t_{ij} \log L_S(d_{ij}) + (1 - t_{ij}) \log L_D(d_{ij})\}, \quad (5.57)$$

where $\mathcal{P}$ is the set of all labeled input pairs, L_S and L_D are the similar and dissimilar loss functions, and $d_{ij} = \|\mathbf{v}_i - \mathbf{v}_j\|$ are the pairwise distance between paired embeddings.²⁷

²⁶Indicator variables are often denoted as y_{ij} , but we will stick to the t_{ij} notation to be consistent with Section 5.1.3.

²⁷In metric learning, the embeddings are very often normalized to unit length.

This has a form similar to the cross-entropy loss given in (5.19), except that we measure squared distances between encodings $\mathbf{v}_i$ and $\mathbf{v}_j$. In their paper, Hadsell, Chopra, and LeCun (2006) suggest using a quadratic function for L_S and a quadratic hinge loss (c.f. (5.30)) $L_D = [m - d_{ij}]_+^2$ for dissimilarity, where m is a margin beyond which there is no penalty.

To train with a contrastive loss, you can run both pairs of inputs through the neural network, compute the loss, and then backpropagate the gradients through both instantiations (activations) of the network. This can also be thought of as constructing a *Siamese network* consisting of two copies with shared weights (Bromley, Guyon *et al.* 1994; Chopra, Hadsell, and LeCun 2005). It is also possible to construct a *triplet loss* that takes as input a pair of matching samples and a third non-matching sample and ensures that the distance between non-matching samples is greater than the distance between matches plus some margin (Weinberger and Saul 2009; Weston, Bengio, and Usunier 2011; Schroff, Kalenichenko, and Philbin 2015; Rawat and Wang 2017).

Both pairwise contrastive and triplet losses can be used to learn embeddings for visual similarity search (Bell and Bala 2015; Wu, Manmatha *et al.* 2017; Bell, Liu *et al.* 2020), as discussed in more detail in Section 6.2.3. They have also been recently used for unsupervised pre-training of neural networks (Wu, Xiong *et al.* 2018; He, Fan *et al.* 2020; Chen, Kornblith *et al.* 2020), which we discuss in Section 5.4.7. In this case, it is more common to use a different contrastive loss function, inspired by softmax (5.3) and multi-class cross-entropy (5.20–5.22), which was first proposed by (Sohn 2016). Before computing the loss, the embeddings are all normalized to unit norm, $\|\hat{\mathbf{v}}_i\|^2 = 1$. Then, the following loss is summed over all matching embeddings,

$$l_{ij} = -\log \frac{\exp(\hat{\mathbf{v}}_i \cdot \hat{\mathbf{v}}_j / \tau)}{\sum_k \exp(\hat{\mathbf{v}}_i \cdot \hat{\mathbf{v}}_k / \tau)}, \quad (5.58)$$

with the denominator summed over non-matches as well. The τ variable denotes the “temperature” and controls how tight the clusters will be; it is sometimes replaced with an s multiplier parameterizing the hyper-sphere radius (Deng, Guo *et al.* 2019). The exact details of how the matches are computed vary by exact implementation.

This loss goes by several names, including InfoNCE (Oord, Li, and Vinyals 2018), and NT-Xent (normalized temperature cross-entropy loss) in Chen, Kornblith *et al.* (2020). Generalized versions of this loss called SphereFace, CosFace, and ArcFace are discussed and compared in the ArcFace paper (Deng, Guo *et al.* 2019) and used by Bell, Liu *et al.* (2020) as part of their visual similarity search system. The *smoothed average precision* loss recently proposed by Brown, Xie *et al.* (2020) can sometimes be used as an alternative to the metric losses discussed in this section. Some recent papers that compare and discuss deep metric learning approaches include (Jacob, Picard *et al.* 2019; Musgrave, Belongie, and Lim 2020).

Weight initialization

Before we can start optimizing the weights in our network, we must first initialize them. Early neural networks used small random weights to break the symmetry, i.e., to make sure that all of the gradients were not zero. It was observed, however, that in deeper layers, the activations would get progressively smaller.

To maintain a comparable variance in the activations of successive layers, we must take into account the *fan-in* of each layer, i.e., the number of incoming connections where activations get multiplied by weights. Glorot and Bengio (2010) did an initial analysis of this issue, and came up with a recommendation to set the random initial weight variance as the inverse of the fan-in. Their analysis, however, assumed a linear activation function (at least around the origin), such as a tanh function.

Since most modern deep neural networks use the ReLU activation function (5.49), He, Zhang *et al.* (2015) updated this analysis to take into account this asymmetric non-linearity. If we initialize the weights to have zero mean and variance V_l for layer l and set the original biases to zero, the linear summation in (5.45) will have a variance of

$$\text{Var}[s_l] = n_l V_l E[x_l^2], \quad (5.59)$$

where n_l is the number of incoming activations/weights and $E[x_l^2]$ is the expectation of the squared incoming activations. When the summations s_l , which have zero mean, are fed through the ReLU, the negative ones will get clamped to zero, so the expectation of the squared output $E[y_l^2]$ is half the variance of s_l , $\text{Var}[s_l]$.

In order to avoid decaying or increasing average activations in deeper layers, we want the magnitude of the activations in successive layers to stay about the same. Since we have

$$E[y_l^2] = \frac{1}{2} \text{Var}[s_l] = \frac{1}{2} n_l V_l E[x_l^2], \quad (5.60)$$

we conclude that the variance in the initial weights V_l should be set to

$$V_l = \frac{2}{n_l}, \quad (5.61)$$

i.e., the inverse of half the fan-in of a given unit or layer. This weight initialization rule is commonly called *He initialization*.

Neural network initialization continues to be an active research area, with publications that include Krähenbühl, Doersch *et al.* (2016), Mishkin and Matas (2016), Frankle and Carbin (2019), and Zhang, Dauphin, and Ma (2019)

5.3.5 Backpropagation

Once we have set up our neural network by deciding on the number of layers, their widths and depths, added some regularization terms, defined the loss function, and initialized the weights, we are ready to train the network with our sample data. To do this, we use gradient descent or one of its variants to iteratively modify the weights until the network has converged to a good set of values, i.e., an acceptable level of performance on the training and validation data.

To do this, we compute the derivatives (gradients) of the loss function E_n for training sample n with respect to the weights $\mathbf{w}$ using the chain rule, starting with the outputs and working our way back through the network towards the inputs, as shown in Figure 5.31. This procedure is known as *backpropagation* (Rumelhart, Hinton, and Williams 1986b) and stands for *backward propagation of errors*. You can find alternative descriptions of this technique in textbooks and course notes on deep learning, including Bishop (2006, Section 5.3.1), Goodfellow, Bengio, and Courville (2016, Section 6.5), Glassner (2018, Chapter 18), Johnson (2020, Lecture 6), and Zhang, Lipton *et al.* (2021).

Recall that in the forward (evaluation) pass of a neural network, activations (layer outputs) are computed layer-by-layer, starting with the first layer and finishing at the last. We will see in the next section that many newer DNNs have an acyclic graph structure, as shown in Figures 5.42–5.43, rather than just a single linear pipeline. In this case, any breadth-first traversal of the graph can be used. The reason for this evaluation order is computational efficiency. Activations need only be computed once for each input sample and can be re-used in succeeding stages of computation.

During backpropagation, we perform a similar breadth-first traversal of the reverse graph. However, instead of computing activations, we compute derivatives of the loss with respect to the weights and inputs, which we call *errors*. Let us look at this in more detail, starting with the loss function.

The derivative of the cross-entropy loss E_n (5.54) with respect to the output probability p_{nk} is simply $-\delta_{nt_n}/p_{nk}$. What is more interesting is the derivative of the loss with respect to the *scores* s_{nk} going into the softmax layer (5.55) shown in Figure 5.27,

$$\frac{\partial E_n}{\partial s_{nk}} = -\delta_{nt_n} + \frac{1}{Z_n} \exp s_{nk} = p_{nk} - \delta_{nt_n} = p_{nk} - \tilde{t}_{nk}. \quad (5.62)$$

(The last form is useful if we are using one-hot encoding or the targets have non-binary probabilities.) This has a satisfyingly intuitive explanation as the difference between the predicted class probability p_{nk} and the true class identity t_{nk} .

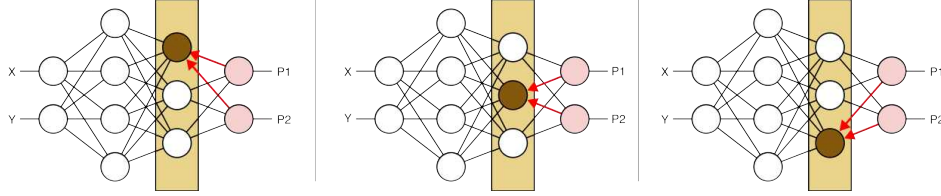


Figure 5.31 Backpropagating the derivatives (errors) through an intermediate layer of the deep network © Glassner (2018). The derivatives of the loss function applied to a single training example with respect to each of the pink unit inputs are summed together and the process is repeated chaining backward through the network.

For the L_2 loss in (5.56), we get a similar result,

$$\frac{\partial E_n}{\partial y_{nk}} = y_{nk} - t_{nk}, \quad (5.63)$$

which in this case denotes the real-valued difference between the predicted and target values.

In the rest of this section, we drop the sample index n from the activations x_{in} and y_{in} , since the derivatives for each sample n can typically be computed independently from other samples.²⁸

To compute the partial derivatives of the loss term with respect to earlier weights and activations, we work our way back through the network, as shown in Figure 5.31. Recall from (5.45–5.46) that we first compute a weighted sum s_i by taking a dot product between the input activations $\mathbf{x}_i$ and the unit's weight vector $\mathbf{w}_i$,

$$s_i = \mathbf{w}_i^T \mathbf{x}_i + b_i = \sum_j w_{ij} x_{ij} + b_i. \quad (5.64)$$

We then pass this weighted sum through an activation function h to obtain $y_i = h(s_i)$.

To compute the derivative of the loss E_n with respect to the weights, bias, and input

²⁸This is not the case if batch or other kinds of normalization (Section 5.3.3) are being used. For batch normalization, we have to accumulate the statistics across all the samples in the batch and then take their derivatives with respect to each weight (Ioffe and Szegedy 2015). For instance and group norm, we compute the statistics across all the pixels in a given channel or group, and then have to compute these additional derivatives as well.

activations, we use the chain rule,

$$e_i = \frac{\partial E_n}{\partial s_i} = h'(s_i) \frac{\partial E_n}{\partial y_i}, \quad (5.65)$$

$$\frac{\partial E_n}{\partial w_{ij}} = x_{ij} \frac{\partial E_n}{\partial s_i} = x_{ij} e_i, \quad (5.66)$$

$$\frac{\partial E_n}{\partial b_i} = \frac{\partial E_n}{\partial s_i} = e_i, \quad \text{and} \quad (5.67)$$

$$\frac{\partial E_n}{\partial x_{ij}} = w_{ij} \frac{\partial E_n}{\partial s_i} = w_{ij} e_i. \quad (5.68)$$

We call the term $e_i = \partial E_n / \partial s_i$, i.e., the partial derivative of the loss E_n with respect to the summed activation s_i , the *error*, as it gets propagated backward through the network.

Now, where do these errors come from, i.e., how do we obtain $\partial E_n / \partial y_i$? Recall from Figure 5.24 that the outputs from one unit or layer become the inputs for the next layer. In fact, for a simple network like the one in Figure 5.24, if we let x_{ij} be the activation that unit i receives from unit j (as opposed to just the j th input to unit i), we can simply set $x_{ij} = y_j$.

Since y_i , the output of unit i , now serves as input for the other units $k > i$ (assuming the units are ordered breadth first), we have

$$\frac{\partial E_n}{\partial y_i} = \sum_{k>i} \frac{\partial E_n}{\partial x_{ki}} = \sum_{k>i} w_{ki} e_k \quad (5.69)$$

and

$$e_i = h'(s_i) \frac{\partial E_n}{\partial y_i} = h'(s_i) \sum_{k>i} w_{ki} e_k. \quad (5.70)$$

In other words, to compute a unit's (backpropagation) error, we compute a weighted sum of the errors coming from the units it feeds into and then multiply this by the derivative of the current activation function $h'(s_i)$. This backward flow of errors is shown in Figure 5.31, where the errors for the three units in the shaded box are computed using weighted sums of the errors coming from later in the network.

This backpropagation rule has a very intuitive explanation. The error (derivative of the loss) for a given unit depends on the errors of the units that it feeds multiplied by the weights that couple them together. This is a simple application of the chain rule. The slope of the activation function $h'(s_i)$ modulates this interaction. If the unit's output is clamped to zero or small, e.g., with a negative-input ReLU or the "flat" part of a sigmoidal response, the unit's error is itself zero or small. The gradient of the weight, i.e., how much the weight should be perturbed to reduce the loss, is a signed product of the incoming activation and the unit's error, $x_{ij} e_i$. This is closely related to the Hebbian update rule (Hebb 1949), which observes that synaptic efficiency in biological neurons increases with correlated firing in the presynaptic

and postsynaptic cells. An easier way to remember this rule is “neurons wire together if they fire together” (Lowel and Singer 1992).

There are, of course, other computational elements in modern neural networks, including convolutions and pooling, which we cover in the next section. The derivatives and error propagation through such other units follows the same procedure as we sketched here, i.e., recursively apply the chain rule, taking analytic derivatives of the functions being applied, until you have the derivatives of the loss function with respect to all the parameters being optimized, i.e., the gradient of the loss.

As you may have noticed, the computation of the gradients with respect to the weights requires the unit activations computed in the forward pass. A typical implementation of neural network training stores the activations for a given sample and uses these during the backprop (backward error propagation) stage to compute the weight derivatives. Modern neural networks, however, may have millions of units and hence activations (Figure 5.44). The number of activations that need to be stored can be reduced by only storing them at certain layers and then re-computing the rest as needed, which goes under the name *gradient checkpointing* (Griewank and Walther 2000; Chen, Xu *et al.* 2016; Bulatov 2018).²⁹ A more extensive review of low-memory training can be found in the technical report by Sohoni, Aberger *et al.* (2019).

5.3.6 Training and optimization

At this point, we have all of the elements needed to train a neural network. We have defined the network’s topology in terms of the sizes and depths of each layer, specified our activation functions, added regularization terms, specified our loss function, and initialized the weights. We have even described how to compute the gradients, i.e., the derivatives of the regularized loss with respect to all of our weights. What we need at this point is some algorithm to turn these gradients into weight updates that will optimize the loss function and produce a network that generalizes well to new, unseen data.

In most computer vision algorithms such as optical flow (Section 9.1.3), 3D reconstruction using bundle adjustment (Section 11.4.2), and even in smaller-scale machine learning problems such as logistic regression (Section 5.1.3), the method of choice is linearized least squares (Appendix A.3). The optimization is performed using a second-order method such as Gauss-Newton, in which we evaluate all of the terms in our loss function and then take an optimally-sized downhill step using a direction derived from the gradients and the Hessian of the energy function.

²⁹This name seems a little weird, since it’s actually the activations that are saved instead of the gradients.

Unfortunately, deep learning problems are far too large (in terms of number of parameters and training samples; see Figure 5.44) to make this approach practical. Instead, practitioners have developed a series of optimization algorithms based on extensions to *stochastic gradient descent* (SGD) (Zhang, Lipton *et al.* 2021, Chapter 11). In SGD, instead of evaluating the loss function by summing over all the training samples, as in (5.54) or (5.56), we instead just evaluate a single training sample n and compute the derivatives of the associated loss $E_n(\mathbf{w})$. We then take a tiny downhill step along the direction of this gradient.

In practice, the directions obtained from just a single sample are incredibly noisy estimates of a good descent direction, so the losses and gradients are usually summed over a small subset of the training data,

$$E_{\mathcal{B}}(\mathbf{w}) = \sum_{n \in \mathcal{B}} E_n(\mathbf{w}), \quad (5.71)$$

where each subset $\mathcal{B}$ is called a *minibatch*. Before we start to train, we randomly assign the training samples into a fixed set of minibatches, each of which has a fixed size that commonly ranges from 32 at the low end to 8k at the higher end (Goyal, Dollár *et al.* 2017). The resulting algorithm is called *minibatch stochastic gradient descent*, although in practice, most people just call it SGD (omitting the reference to minibatches).³⁰

After evaluating the gradients $\mathbf{g} = \nabla_{\mathbf{w}} E_{\mathcal{B}}$ by summing over the samples in the minibatch, it is time to update the weights. The simplest way to do this is to take a small step in the gradient direction,

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \mathbf{g} \quad \text{or} \quad (5.72)$$

$$\mathbf{w}_{t+i} = \mathbf{w}_t - \alpha_t \mathbf{g}_t \quad (5.73)$$

where the first variant looks more like an assignment statement (see, e.g., Zhang, Lipton *et al.* 2021, Chapter 11; Loshchilov and Hutter 2019), while the second makes the temporal dependence explicit, using t to denote each successive step in the gradient descent.³¹

The step size parameter α is often called the *learning rate* and must be carefully adjusted to ensure good progress while avoiding overshooting and exploding gradients. In practice, it is common to start with a larger (but still small) learning rate α_t and to decrease it over time so that the optimization settles into a good minimum (Johnson 2020, Lecture 11; Zhang, Lipton *et al.* 2021, Chapter 11).

³⁰In the deep learning community, classic algorithms that sum over all the measurements are called *batch gradient descent*, although this term is not widely used elsewhere, as it is assumed that using all measurement at once is the preferred approach. In large-scale problems such as bundle adjustment, it's possible that using minibatches may result in better performance, but this has so far not been explored.

³¹I use the index k in discussing iterative algorithms in Appendix A.5.

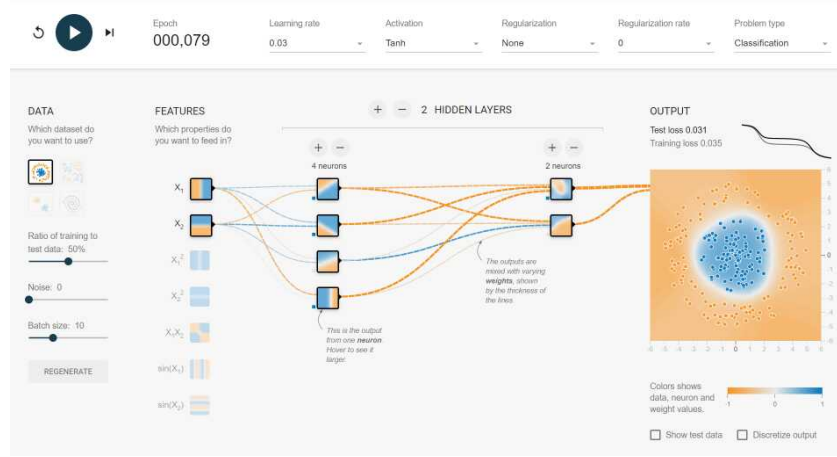


Figure 5.32 Screenshot from <http://playground.tensorflow.org>, where you can build and train your own small network in your web browser. Because the input space is two-dimensional, you can visualize the responses to all 2D inputs at each unit in the network.

Regular gradient descent is prone to stalling when the current solution reaches a “flat spot” in the search space, and stochastic gradient descent only pays attention to the errors in the current minibatch. For these reasons, the SGD algorithms may use the concept of *momentum*, where an exponentially decaying (“leaky”) running average of the gradients is accumulated and used as the update direction,

$$\mathbf{v}_{t+i} = \rho \mathbf{v}_t + \mathbf{g}_t \quad (5.74)$$

$$\mathbf{w}_{t+i} = \mathbf{w}_t - \alpha_t \mathbf{v}_t. \quad (5.75)$$

A relatively large value of $\rho \in [0.9, 0.99]$ is used to give the algorithm good memory, effectively averaging gradients over more batches.³²

Over the last decade, a number of more sophisticated optimization techniques have been applied to deep network training, as described in more detail in Johnson (2020, Lecture 11) and Zhang, Lipton *et al.* (2021, Chapter 11)). These algorithms include:

- *Nesterov momentum*, where the gradient is (effectively) computed at the state predicted from the velocity update;

³²Note that a recursive formula such as (5.74), which is the same as a temporal infinite impulse response filter (3.2.3) converges in the limit to a value of $\mathbf{g}/(1 - \rho)$, so α needs to be correspondingly adjusted.

- *AdaGrad* (Adaptive Gradient), where each component in the gradient is divided by the square root of the per-component summed squared gradients (Duchi, Hazan, and Singer 2011);
- *RMSProp*, where the running sum of squared gradients is replaced with a leaky (decay-ing) sum (Hinton 2012);
- *Adadelta*, which augments RMSProp with a leaky sum of the actual per-component changes in the parameters and uses these in the gradient re-scaling equation (Zeiler 2012);
- *Adam*, which combines elements of all the previous ideas into a single framework and also de-biases the initial leaky estimates (Kingma and Ba 2015); and
- *AdamW*, which is Adam with decoupled weight decay (Loshchilov and Hutter 2019).

Adam and AdamW are currently the most popular optimizers for deep networks, although even with all their sophistication, learning rates need to be set carefully (and probably decayed over time) to achieve good results. Setting the right *hyperparameters*, such as the learning rate initial value and decay rate, momentum terms such as ρ , and amount of regularization, so that the network achieves good performance within a reasonable training time is itself an open research area. The lecture notes by Johnson (2020, Lecture 11) provide some guidance, although in many cases, people perform a search over hyperparameters to find which ones produce the best performing network.

A simple two-input example

A great way to get some intuition on how deep networks update the weights and carve out a solution space during training is to play with the interactive visualization at <http://playground.tensorflow.org>.³³ As shown in Figure 5.32, just click the “run” (▶) button to get started, then reset the network to a new start (button to the left of run) and try single-stepping the network, using different numbers of units per hidden layer and different activation functions. Especially when using ReLUs, you can see how the network carves out different parts of the input space and then combines these sub-pieces together. Section 5.4.5 discusses visualization tools to get insights into the behavior of larger, deeper networks.

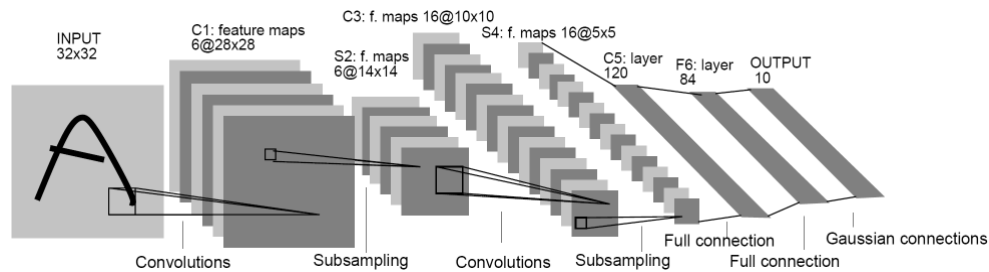


Figure 5.33 Architecture of LeNet-5, a convolutional neural network for digit recognition (LeCun, Bottou et al. 1998) © 1998 IEEE. This network uses multiple channels in each layer and alternates multi-channel convolutions with downsampling operations, followed by some fully connected layers that produce one activation for each of the 10 digits being classified.

5.4 Convolutional neural networks

The previous sections on deep learning have covered all of the essential elements of constructing and training deep networks. However, they have omitted what is likely the most crucial component of deep networks for image processing and computer vision, which is the use of trainable multi-layer convolutions. The idea of convolutional neural networks was popularized by LeCun, Bottou et al. (1998), where they introduced the LeNet-5 network for digit recognition shown in Figure 5.33.³⁴

Instead of connecting all of the units in a layer to all the units in a preceding layer, convolutional networks organize each layer into *feature maps* (LeCun, Bottou et al. 1998), which you can think of as parallel planes or *channels*, as shown in Figure 5.33. In a convolutional layer, the weighted sums are only performed within a small local window, and weights are identical for all pixels, just as in regular shift-invariant image convolution and correlation (3.12–3.15).

Unlike image convolution, however, where the same filter is applied to each (color) channel, neural network convolutions typically linearly combine the activations from each of the C_1 input channels in a previous layer and use different convolution kernels for each of the C_2 output channels, as shown in Figures 5.34–5.35.³⁵ This makes sense, as the main task in

³³Additional informative interactive demonstrations can be found at <https://cs.stanford.edu/people/karpathy/convnetjs>.

³⁴A similar convolutional architecture, but without the gradient descent training procedure, was earlier proposed by Fukushima (1980).

³⁵The number of channels in a given network layer is sometimes called its *depth*, but the number of layers in a deep network is also called its depth. So, be careful when reading network descriptions.

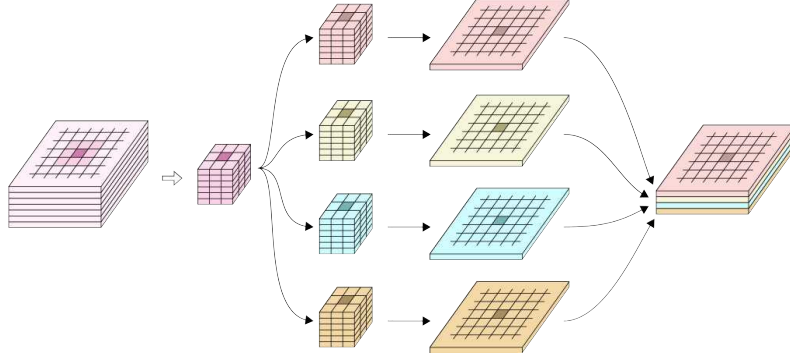


Figure 5.34 2D convolution with multiple input and output channels © Glassner (2018). Each 2D convolution kernel takes as input all of the C_1 channels in the preceding layer, windowed to a small area, and produces the values (after the activation function non-linearity) in one of the C_2 channels in the next layer. For each of the output channels, we have $S^2 \times C_1$ kernel weights, so the total number of learnable parameters in each convolutional layer is $S^2 \times C_1 \times C_2$. In this figure, we have $C_1 = 6$ input channels and $C_2 = 4$ output channels, with an $S = 3$ convolution window, for a total of $9 \times 6 \times 4$ learnable weights, shown in the middle column of the figure. Since the convolution is applied at each of the $W \times H$ pixels in a given layer, the amount of computation (multiply-adds) in each forward and backward pass over one sample in a given layer is $WHS^2C_1C_2$.

convolutional neural network layers is to construct local features (Figure 3.40c) and to then combine them in different ways to produce more discriminative and semantically meaningful features.³⁶ Visualizations of the kinds of features that deep networks extract are shown in Figure 5.47 in Section 5.4.5.

With these intuitions in place, we can write the weighted linear sums (5.45) performed in a convolutional layer as

$$s(i, j, c_2) = \sum_{c_1 \in \{C_1\}} \sum_{(k, l) \in \mathcal{N}} w(k, l, c_1, c_2) x(i + k, j + l, c_1) + b(c_2), \quad (5.76)$$

where the $x(i, j, c_1)$ are the activations in the previous layer, just as in (5.45), $\mathcal{N}$ are the S^2 signed offsets in the 2D spatial kernel, and the notation $c_1 \in \{C_1\}$ denotes $c_1 \in [0, C_1)$. Note that because the offsets (k, l) are added to (instead of subtracted from) the (i, j) pixel coordinates, this operation is actually a *correlation* (3.13), but this distinction is usually glossed

³⁶Note that pixels in different input and output channels (within the convolution window size) are fully connected, unless grouped convolutions, discussed below, are used.

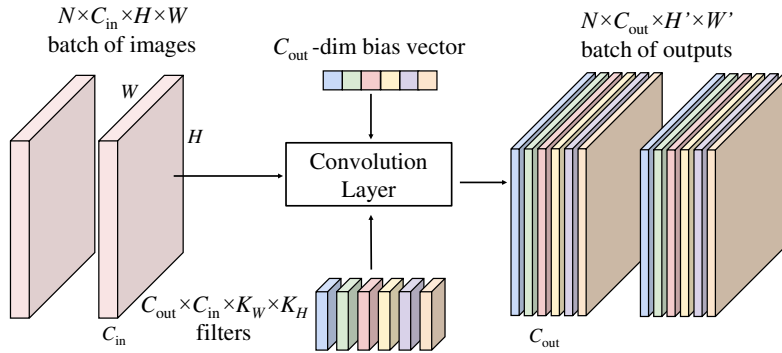


Figure 5.35 2D convolution with multiple batches, input, and output channels, © Johnson (2020). When doing mini-batch gradient descent, a whole batch of training images or features is passed into a convolutional layer, which takes as input all of the C_{in} channels in the preceding layer, windowed to a small area, and produces the values (after the activation function non-linearity) in one of the C_{out} channels in the next layer. As before, for each of the output channels, we have $K_w \times K_h \times C_{in}$ kernel weights, so the total number of learnable parameters in each convolutional layer is $K_w \times K_h \times C_{in} \times C_{in}$. In this figure, we have $C_{in} = 3$ input channels and $C_{out} = 6$ output channels.

over.³⁷

In neural network diagrams such as those shown in Figures 5.33 and 5.39–5.43, it is common to indicate the convolution kernel size S and the number of channels in a layer C , and only sometimes to show the image dimensions, as in Figures 5.33 and 5.39. Note that some neural networks such as the Inception module in GoogLeNet (Szegedy, Liu *et al.* 2015) shown in Figure 5.42 use 1×1 convolutions, which do not actually perform convolutions but rather combine various channels on a per-pixel basis, often with the goal of reducing the dimensionality of the feature space.

Because the weights in a convolution kernel are the same for all of the pixels within a given layer and channel, these weights are actually *shared* across what would result if we drew all of the connections between different pixels in different layers. This means that there are many fewer weights to learn than in fully connected layers. It also means that during backpropagation, kernel weight updates are summed over all of the pixels in a given layer/channel.

To fully determine the behavior of a convolutional layer, we still need to specify a few

³⁷Since the weights in a neural network are learned, this reversal does not really matter.

additional parameters.³⁸ These include:

- *Padding.* Early networks such as LeNet-5 did not pad the image, which therefore shrank after each convolution. Modern networks can optionally specify a padding width and mode, using one of the choices used with traditional image processing, such as zero padding or pixel replication, as shown in Figure 3.13.
- *Stride.* The default stride for convolution is 1 pixel, but it is also possible to only evaluate the convolution at every n th column and row. For example, the first convolution layer in AlexNet (Figure 5.39) uses a stride of 4. Traditional image pyramids (Figure 3.31) use a stride of 2 when constructing the coarser levels.
- *Dilation.* Extra “space” (skipped rows and column) can be inserted between pixel samples during convolution, also known as dilated or *à trous* (with holes, in French, or often just “atrous”) convolution (Yu and Koltun 2016; Chen, Papandreou *et al.* 2018). While in principle this can lead to aliasing, it can also be effective at pooling over a larger region while using fewer operations and learnable parameters.
- *Grouping.* While, by default, all input channels are used to produce each output channel, we can also group the input and output layers into G separate groups, each of which is convolved separately (Xie, Girshick *et al.* 2017). $G = 1$ corresponds to regular convolution, while $G = C_1$ means that each corresponding input channel is convolved independently from the others, which is known as *depthwise* or *channel-separated* convolution (Howard, Zhu *et al.* 2017; Tran, Wang *et al.* 2019).

A nice animation of the effects of these different parameters created by Vincent Dumoulin can be found at https://github.com/vdumoulin/conv_arithmetic as well as Dumoulin and Visin (2016).

In certain applications such as image inpainting (Section 10.5.1), the input image may come with an associated binary *mask*, indicating which pixels are valid and which need to be filled in. This is similar to the concept of alpha-matted images we studied in Section 3.1.3. In this case, one can use *partial convolutions* (Liu, Reda *et al.* 2018), where the input pixels are multiplied by the mask pixels and then normalized by the count of non-zero mask pixels. The mask channel output is set to 1 if any input mask pixels are non-zero. This resembles the pull-push algorithm of Gortler, Grzeszczuk *et al.* (1996) that we presented in Figure 4.2, except that the convolution weights are learned.

³⁸Most of the neural network building blocks we present in this chapter have corresponding functions in widely used deep learning frameworks, where you can get more detailed information about their operation. For example, the 2D convolution operator is called *Conv2d* in PyTorch and is documented at <https://pytorch.org/docs/stable/nn.html#convolution-layers>.

A more sophisticated version of partial convolutions is *gated convolutions* (Yu, Lin *et al.* 2019; Chang, Liu *et al.* 2019), where the per-pixel masks are derived from the previous layer using a learned convolution followed by a sigmoid non-linearity. This enables the network not only to learn a better measure of per-pixel confidence (weighting), but also to incorporate additional features such as user-drawn sketches or derived semantic information.

5.4.1 Pooling and unpooling

As we just saw in the discussion of convolution, strides of greater than 1 can be used to reduce the resolution of a given layer, as in the first convolutional layer of AlexNet (Figure 5.39). When the weights inside the convolution kernel are identical and sum up to 1, this is called *average pooling* and is typically applied in a channel-wise manner.

A widely used variant is to compute the maximum response within a square window, which is called *max pooling*. Common strides and window sizes for max pooling are a stride of 2 and 2×2 non-overlapping windows or 3×3 overlapping windows. Max pooling layers can be thought of as a “logical or”, since they only require one of the units in the pooling region to be turned on. They are also supposed to provide some shift invariance over the inputs. However, most deep networks are not all that shift-invariant, which degrades their performance. The paper by Zhang (2019) has a nice discussion of this issue and some simple suggestions to mitigate this problem.

One issue that commonly comes up is how to backpropagate through a max pooling layer. The max pool operator acts like a “switch” that shunts (connects) one of the input units to the output unit. Therefore, during backpropagation, we only need to pass the error and derivatives down to this maximally active unit, as long as we have remembered which unit has this response.

This same *max unpooling* mechanism can be used to create a “deconvolution network” when searching for the stimulus (Figure 5.47) that most strongly activates a particular unit (Zeiler and Fergus 2014).

If we want a more continuous behavior, we could construct a pooling unit that computes an L_p norm over its inputs, since the $L_{p \rightarrow \infty}$ effectively computes a maximum over its components (Springenberg, Dosovitskiy *et al.* 2015). However, such a unit requires more computation, so it is not widely used in practice, except sometimes at the final layer, where it is known as *generalized mean (GeM) pooling* (Dollár, Tu *et al.* 2009; Tolias, Sivic, and Jégou 2016; Gordo, Almazán *et al.* 2017; Radenović, Tolias, and Chum 2019) or *dynamic mean (DAME) pooling* (Yang, Kien Nguyen *et al.* 2019). In their paper, Springenberg, Dosovitskiy *et al.* (2015) also show that using strided convolution instead of max pooling can produce competitive results.

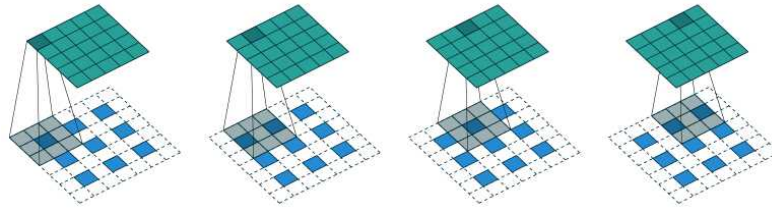


Figure 5.36 *Transposed convolution (© Dumoulin and Visin (2016)) can be used to upsample (increase the size of) an image. Before applying the convolution operator, $(s - 1)$ extra rows and columns of zeros are inserted between the input samples, where s is the upsampling stride.*

While unpooling can be used to (approximately) reverse the effect of max pooling operation, if we want to reverse a convolutional layer, we can look at learned variants of the interpolation operator we studied in Sections 3.5.1 and 3.5.3. The easiest way to visualize this operation is to add extra rows and columns of zeros between the pixels in the input layer, and to then run a regular convolution (Figure 5.36). This operation is sometimes called *backward convolution* with a *fractional stride* (Long, Shelhamer, and Darrell 2015), although it is more commonly known as *transposed convolution* (Dumoulin and Visin 2016), because when convolutions are written in matrix form, this operation is a multiplication with a transposed sparse weight matrix. Just as with regular convolution, padding, stride, dilation, and grouping parameters can be specified. However, in this case, the stride specifies the factor by which the image will be upsampled instead of downsampled.

U-Nets and Feature Pyramid Networks

When discussing the Laplacian pyramid in Section 3.5.3, we saw how image downsampling and upsampling can be combined to achieve a variety of multi-resolution image processing tasks (Figure 3.33). The same kinds of combinations can be used in deep convolutional networks, in particular, when we want the output to be a full-resolution image. Examples of such applications include pixel-wise semantic labeling (Section 6.4), image denoising and super-resolution (Section 10.3), monocular depth inference (Section 12.8), and neural style transfer (Section 14.6). The idea of reducing the resolution of a network and then expanding it again is sometimes called a *bottleneck* and is related to earlier self-supervised network training using *autoencoders* (Hinton and Zemel 1994; Goodfellow, Bengio, and Courville 2016, Chapter 14).

One of the earliest applications of this idea was the *fully convolutional network* developed

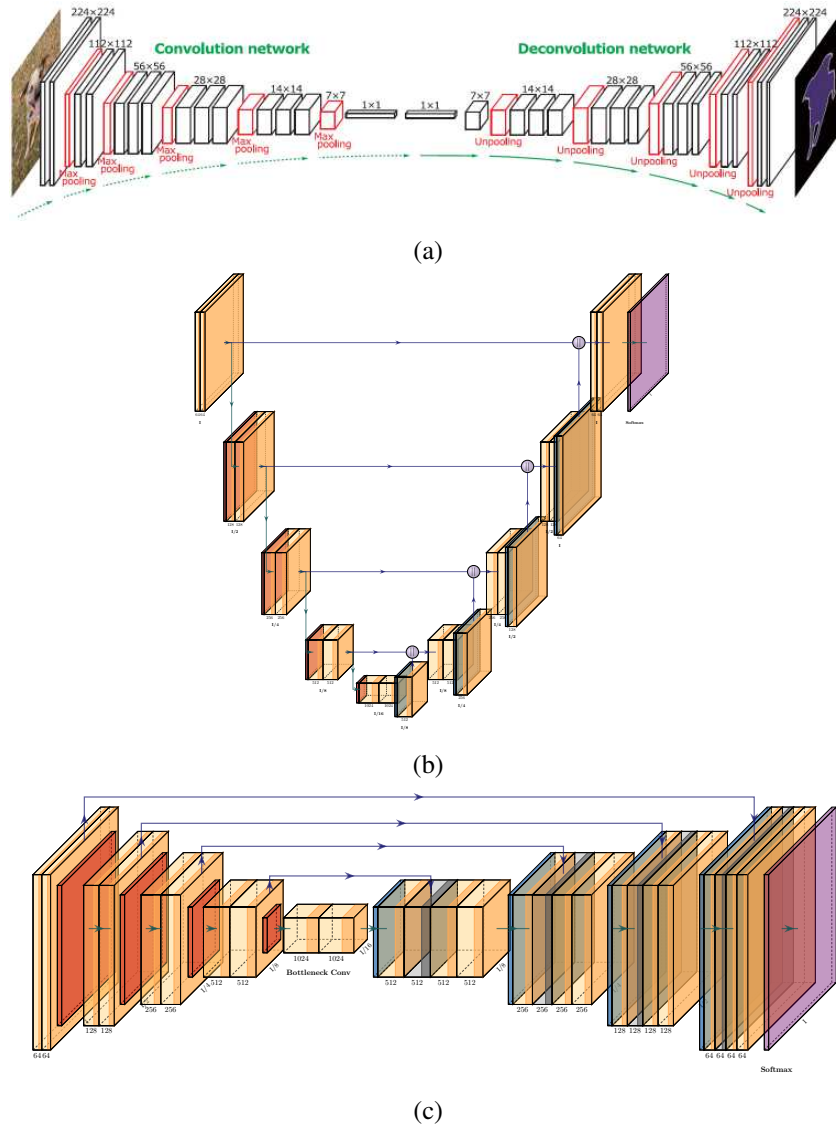


Figure 5.37 (a) The deconvolution network of Noh, Hong, and Han (2015) © 2015 IEEE and (b–c) the U-Net of Ronneberger, Fischer, and Brox (2015), drawn using the PlotNeural-Net LaTeX package. In addition to the fine-to-coarse-to-fine bottleneck used in (a), the U-Net also has skip connections between encoding and decoding layers at the same resolution.



Figure 5.38 Screenshot from Andrej Karpathy’s web browser demos at <https://cs.stanford.edu/people/karpathy/convnetjs>, where you can run a number of small neural networks, including CNNs for digit and tiny image classification.

by Long, Shelhamer, and Darrell (2015). This paper inspired myriad follow-on architectures, including the hourglass-shaped “deconvolution” network of Noh, Hong, and Han (2015), the U-Net of Ronneberger, Fischer, and Brox (2015), the atrous convolution network with CRF refinement layer of Chen, Papandreou *et al.* (2018), and the panoptic feature pyramid networks of Kirillov, Girshick *et al.* (2019). Figure 5.37 shows the general layout of two of these networks, which are discussed in more detail in Section 6.4 on semantic segmentation. We will see other uses of these kinds of *backbone networks* (He, Gkioxari *et al.* 2017) in later sections on image denoising and super-resolution (Section 10.3), monocular depth inference (Section 12.8), and neural style transfer (Section 14.6).

5.4.2 Application: Digit classification

One of the earliest commercial application of convolutional neural networks was the LeNet-5 system created by LeCun, Bottou *et al.* (1998) whose architecture is shown in Figure 5.33. This network contained most of the elements of modern CNNs, although it used sigmoid non-linearities, average pooling, and Gaussian RBF units instead of softmax at its output. If you want to experiment with this simple digit recognition CNN, you can visit the interactive JavaScript demo created by Andrej Karpathy at <https://cs.stanford.edu/people/karpathy/convnetjs> (Figure 5.38).

The network was initially deployed around 1995 by AT&T to automatically read checks deposited in NCR ATM machines to verify that the written and keyed check amounts were the same. The system was then incorporated into NCR’s high-speed check reading systems,

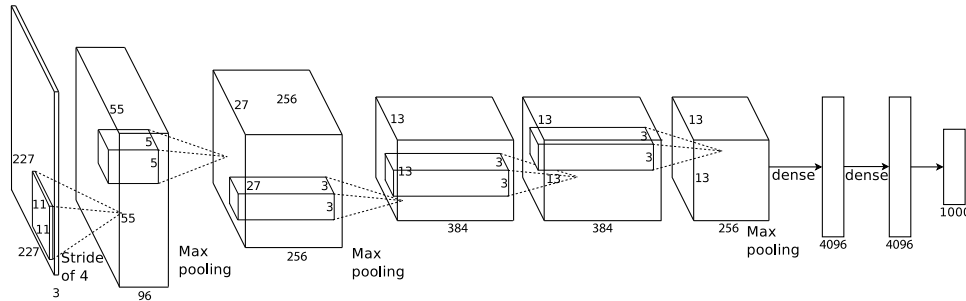


Figure 5.39 Architecture of the SuperVision deep neural network (more commonly known as “AlexNet”), courtesy of Matt Deitke (redrawn from (Krizhevsky, Sutskever, and Hinton 2012)). The network consists of multiple convolutional layers with ReLU activations, max pooling, some fully connected layers, and a softmax to produce the final class probabilities.

which at some point were processing somewhere between 10% and 20% of all the checks in the US.³⁹

Today, variants of the LeNet-5 architecture (Figure 5.33) are commonly used as the first convolutional neural network introduced in courses and tutorials on the subject.⁴⁰ Although the MNIST dataset (LeCun, Cortes, and Burges 1998) originally used to train LeNet-5 is still sometimes used, it is more common to use the more challenging CIFAR-10 (Krizhevsky 2009) or Fashion MNIST (Xiao, Rasul, and Vollgraf 2017) as datasets for training and testing.

5.4.3 Network architectures

While modern convolutional neural networks were first developed and deployed in the late 1990s, it was not until the breakthrough publication by Krizhevsky, Sutskever, and Hinton (2012) that they started outperforming more traditional techniques on natural image classification (Figure 5.40). As you can see in this figure, the AlexNet system (the more widely used name for their SuperVision network) led to a dramatic drop in error rates from 25.8% to 16.4%. This was rapidly followed in the next few years with additional dramatic performance improvements, due to further developments as well as the use of deeper networks, e.g., from the original 8-layer AlexNet to a 152-layer ResNet.

Figure 5.39 shows the architecture of the SuperVision network, which contains a series of convolutional layers with ReLU (rectified linear) non-linearities, max pooling, some fully

³⁹This information courtesy of Yann LeCun and Larry Jackel, who were two of the principals in the development of this system.

⁴⁰See, e.g., https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.

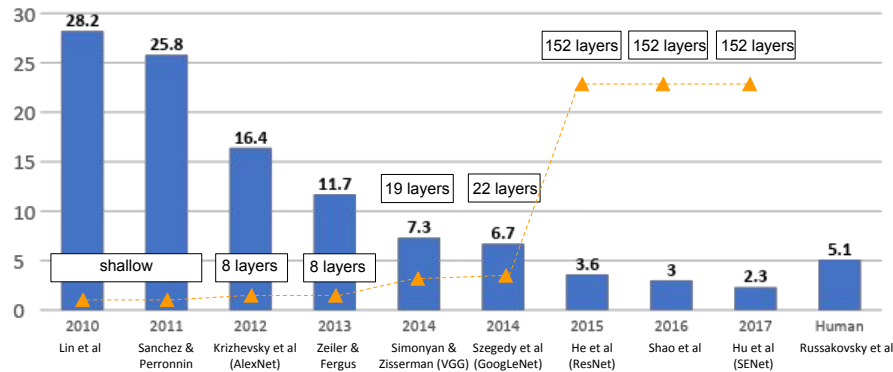


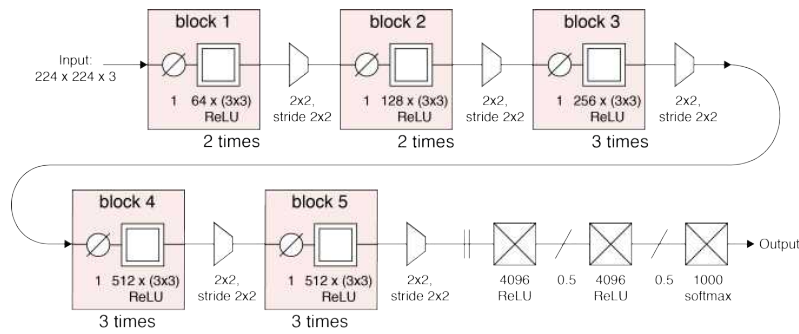
Figure 5.40 *Top-5 error rate and network depths of winning entries from the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) © Li, Johnson, and Yeung (2019).*

connected layers, and a final softmax layer, which is fed into a multi-class cross-entropy loss. Krizhevsky, Sutskever, and Hinton (2012) also used dropout (Figure 5.29), small translation and color manipulation for data augmentation, momentum, and weight decay (L_2 weight penalties).

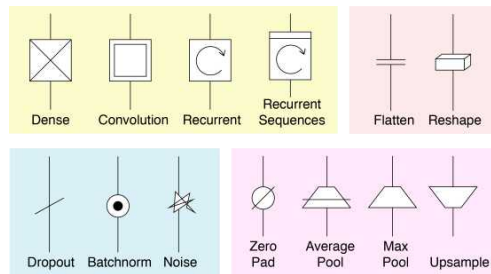
The next few years after the publication of this paper saw dramatic improvement in the classification performance on the ImageNet Large Scale Visual Recognition Challenge (Russakovsky, Deng *et al.* 2015), as shown in Figure 5.40. A nice description of the innovations in these various networks, as well as their capacities and computational cost, can be found in the lecture slides by Justin Johnson (2020, Lecture 8).

The winning entry from 2013 by Zeiler and Fergus (2014) used a larger version of AlexNet with more channels in the convolution stages and lowered the error rate by about 30%. The 2014 Oxford Visual Geometry Group (VGG) winning entry by Simonyan and Zisserman (2014b) used repeated 3×3 convolution/ReLU blocks interspersed with 2×2 max pooling and channel doubling (Figure 5.41), followed by some fully connected layers, to produce 16–19 layer networks that further reduced the error by 40%. However, as shown in Figure 5.44, this increased performance came at a greatly increased amount of computation.

The 2015 GoogLeNet of Szegedy, Liu *et al.* (2015) focused instead on efficiency. GoogLeNet begins with an aggressive stem network that uses a series of strided and regular convolutions and max pool layers to quickly reduce the image resolutions from 224^2 to 28^2 . It then uses a number of *Inception modules* (Figure 5.42), each of which is a small branching neural network whose features get concatenated at the end. One of the important characteristics of this module is that it uses 1×1 “bottleneck” convolutions to reduce the number of



(a)



(b)

Figure 5.41 The VGG16 network of *Simonyan and Zisserman (2014b)* © *Glassner (2018)*.
 (a) The network consists of repeated zero-pad, 3×3 convolution, ReLU blocks interspersed with 2×2 max pooling and a doubling in the number of channels. This is followed by some fully connected and dropout layers, with a final softmax into the 1,000 ImageNet categories.
 (b) Some of the schematic neural network symbols used by *Glassner (2018)*.

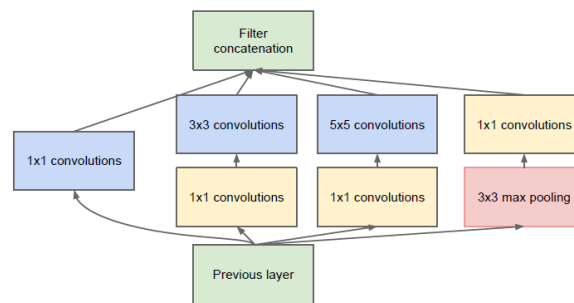


Figure 5.42 An Inception module from (*Szegedy, Liu et al. 2015*) © 2015 IEEE, which combines dimensionality reduction, multiple convolution sizes, and max pooling as different channels that get stacked together into a final feature map.

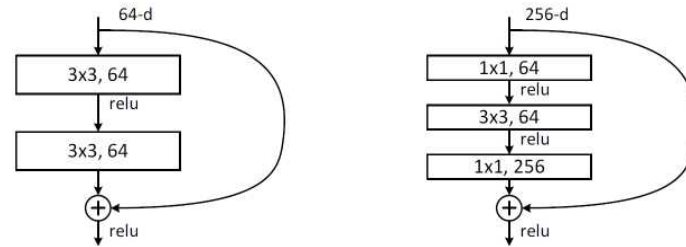


Figure 5.43 *ResNet residual networks (He, Zhang et al. 2016a) © 2016 IEEE, showing skip connections going around a series of convolutional layers. The figure on the right uses a bottleneck to reduce the number of channels before the convolution. Having direct connections that shortcut the convolutional layer allows gradients to more easily flow backward through the network during training.*

channels before performing larger 3×3 and 5×5 convolutions, thereby saving a significant amount of computation. This kind of projection followed by an additional convolution is similar in spirit to the approximation of filters as a sum of separable convolutions proposed by Perona (1995). GoogLeNet also removed the fully connected (MLP) layers at the end, relying instead on global average pooling followed by one linear layer before the softmax. Its performance was similar to that of VGG but at dramatically lower computation and model size costs (Figure 5.44).

The following year saw the introduction of Residual Networks (He, Zhang et al. 2016a), which dramatically expanded the number of layers that could be successfully trained (Figure 5.40). The main technical innovation was the introduction of *skip connections* (originally called “shortcut connections”), which allow information (and gradients) to flow around a set of convolutional layers, as shown in Figure 5.43. The networks are called *residual networks* because they allow the network to learn the residuals (differences) between a set of incoming and outgoing activations. A variant on the basic residual block is the “bottleneck block” shown on the right side of Figure 5.43, which reduces the number of channels before performing the 3×3 convolutional layer. A further extension, described in (He, Zhang et al. 2016b), moves the ReLU non-linearity to before the residual summation, thereby allowing true identity mappings to be modeled at no cost.

To build a ResNet, various residual blocks are interspersed with strided convolutions and channel doubling to achieve the desired number of layers. (Similar downsampling stems and average pooled softmax layers as in GoogLeNet are used at the beginning and end.) By combining various numbers of residual blocks, ResNets consisting of 18, 34, 50, 101, and 152 layers have been constructed and evaluated. The deeper networks have higher accuracy

but more computational cost (Figure 5.44). In 2015, ResNet not only took first place in the ILSVRC (ImageNet) classification, detection, and localization challenges, but also took first place in the detection and segmentation challenges on the newer COCO dataset and benchmark (Lin, Maire *et al.* 2014).

Since then, myriad extensions and variants have been constructed and evaluated. The ResNeXt system from Xie, Girshick *et al.* (2017) used grouped convolutions to slightly improve accuracy. Denseley connected CNNs (Huang, Liu *et al.* 2017) added skip connections between non-adjacent convolution and/or pool blocks. Finally, the Squeeze-and-Excitation network (SENet) by Hu, Shen, and Sun (2018) added global context (via global pooling) to each layer to obtain a noticeable increase in accuracy. More information about these and other CNN architectures can be found in both the original papers as well as class notes on this topic (Li, Johnson, and Yeung 2019; Johnson 2020).

Mobile networks

As deep neural networks were getting deeper and larger, a countervailing trend emerged in the construction of smaller, less computationally expensive networks that could be used in mobile and embedded applications. One of the earliest networks tailored for lighter-weight execution was MobileNets (Howard, Zhu *et al.* 2017), which used *depthwise convolutions*, a special case of grouped convolutions where the number of groups equals the number of channels. By varying two hyperparameters, namely a *width multiplier* and a *resolution multiplier*, the network architecture could be tuned along an accuracy vs. size vs. computational efficiency tradeoff. The follow-on MobileNetV2 system (Sandler, Howard *et al.* 2018) added an “inverted residual structure”, where the shortcut connections were between the bottleneck layers. ShuffleNet (Zhang, Zhou *et al.* 2018) added a “shuffle” stage between grouped convolutions to enable channels in different groups to co-mingle. ShuffleNet V2 (Ma, Zhang *et al.* 2018) added a channel split operator and tuned the network architectures using end-to-end performance measures. Two additional networks designed for computational efficiency are ESPNet (Mehta, Rastegari *et al.* 2018) and ESPNetv2 (Mehta, Rastegari *et al.* 2019), which use pyramids of (depth-wide) dilated separable convolutions.

The concepts of grouped, depthwise, and channel-separated convolutions continue to be a widely used tool for managing computational efficiency and model size (Choudhary, Mishra *et al.* 2020), not only in mobile networks, but also in video classification (Tran, Wang *et al.* 2019), which we discuss in more detail in Section 5.5.2.

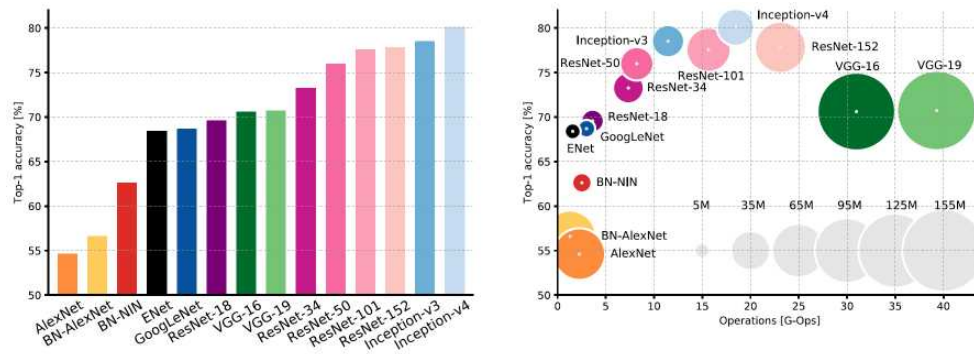


Figure 5.44 Network accuracy vs. size and operation counts (Canziani, Culurciello, and Paszke 2017) © 2017 IEEE: In the right figure, the network accuracy is plotted against operation count (1–40 G-Ops), while the size of the circle indicates the number of parameters (10–155 M). The initials BN indicate a batch normalized version of a network.

5.4.4 Model zoos

A great way to experiment with these various CNN architectures is to download pre-trained models from a *model zoo*⁴¹ such as the TorchVision library at <https://github.com/pytorch/vision>. If you look in the `torchvision/models` folder, you will find implementations of AlexNet, VGG, GoogleNet, Inception, ResNet, DenseNet, MobileNet, and ShuffleNet, along with other models for classification, object detection, and image segmentation. Even more recent models, some of which are discussed in the upcoming sections, can be found in the PyTorch Image Models library (timm), <https://github.com/rwightman/pytorch-image-models>. Similar collections of pre-trained models exist for other languages, e.g., <https://www.tensorflow.org/lite/models> for efficient (mobile) TensorFlow models.

While people often download and use pre-trained neural networks for their applications, it is more common to at least *fine-tune* such networks on data more characteristic of the application (as opposed to the public benchmark data on which most zoo models are trained).⁴² It is also quite common to replace the last few layers, i.e., the *head* of the network (so called because it lies at the top of a layer diagram when the layers are stacked bottom-to-top) while leaving the *backbone* intact. The terms backbone and head(s) are widely used and were popularized by the Mask-RCNN paper (He, Gkioxari *et al.* 2017). Some more recent papers

⁴¹The name “model zoo” itself is a fanciful invention of Evan Shelhamer, lead developer on Caffe (Jia, Shelhamer *et al.* 2014), who first used it on https://caffe.berkeleyvision.org/model_zoo.html to describe a collection of various pre-trained DNN models (personal communication).

⁴²See, e.g., https://classyvision.ai/tutorials/fine_tuning and (Zhang, Lipton *et al.* 2021, Section 13.2).

refer to the backbone and head as the *trunk* and its *branches* (Ding and Tao 2018; Kirillov, Girshick *et al.* 2019; Bell, Liu *et al.* 2020), with the term *neck* also being occasionally used (Chen, Wang *et al.* 2019).⁴³

When adding a new head, its parameters can be trained using the new data specific to the intended application. Depending on the amount and quality of new training data available, the head can be as simple as a linear model such as an SVM or logistic regression/softmax (Donahue, Jia *et al.* 2014; Sharif Razavian, Azizpour *et al.* 2014), or as complex as a fully connected or convolutional network (Xiao, Liu *et al.* 2018). Fine-tuning some of the layers in the backbone is also an option, but requires sufficient data and a slower learning rate so that the benefits of the pre-training are not lost. The process of pre-training a machine learning system on one dataset and then applying it to another domain is called *transfer learning* (Pan and Yang 2009). We will take a closer look at transfer learning in Section 5.4.7 on self-supervised learning.

Model size and efficiency

As you can tell from the previous discussion, neural network models come in a large variety of sizes (typically measured in number of parameters, i.e., weights and biases) and computational loads (often measured in FLOPs per forward inference pass). The evaluation by Canziani, Culurciello, and Paszke (2017), summarized in Figure 5.44, gives a snapshot of the performance (accuracy and size+operations) of the top-performing networks on the ImageNet challenge from 2012–2017. In addition to the networks we have already discussed, the study includes Inception-v3 (Szegedy, Vanhoucke *et al.* 2016) and Inception-v4 (Szegedy, Ioffe *et al.* 2017).

Because deep neural networks can be so memory- and compute-intensive, a number of researchers have investigated methods to reduce both, using lower precision (e.g., fixed-point) arithmetic and weight compression (Han, Mao, and Dally 2016; Iandola, Han *et al.* 2016). The XNOR-Net paper by Rastegari, Ordonez *et al.* (2016) investigates using binary weights (on-off connections) and optionally binary activations. It also has a nice review of previous binary networks and other compression techniques, as do more recent survey papers (Sze, Chen *et al.* 2017; Gu, Wang *et al.* 2018; Choudhary, Mishra *et al.* 2020).

Neural Architecture Search (NAS)

One of the most recent trends in neural network design is the use of *Neural Architecture Search* (NAS) algorithms to try different network topologies and parameterizations (Zoph

⁴³Classy Vision uses a trunk and heads terminology, https://classyvision.ai/tutorials/classy_model.

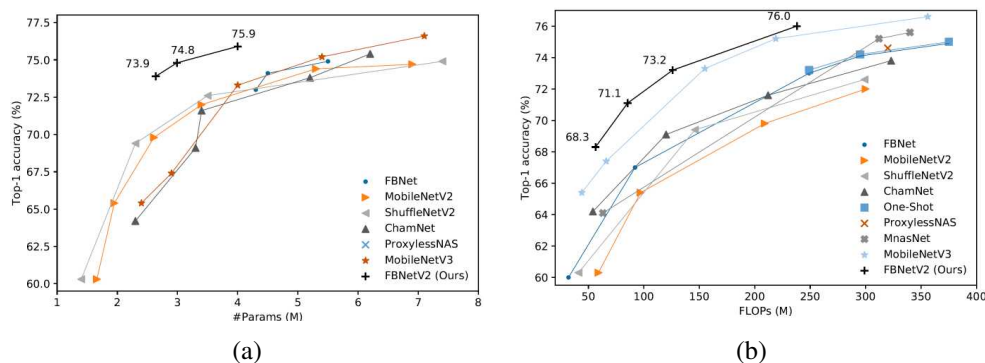


Figure 5.45 ImageNet accuracy vs. (a) size (# of parameters) and (b) operation counts for a number of recent efficient networks (Wan, Dai *et al.* 2020) © 2020 IEEE.

and Le 2017; Zoph, Vasudevan *et al.* 2018; Liu, Zoph *et al.* 2018; Pham, Guan *et al.* 2018; Liu, Simonyan, and Yang 2019; Hutter, Kotthoff, and Vanschoren 2019). This process is more efficient (in terms of a researcher’s time) than the trial-and-error approach that characterized earlier network design. Elsken, Metzen, and Hutter (2019) survey these and additional papers on this rapidly evolving topic. More recent publications include FBNet (Wu, Dai *et al.* 2019), RandomNets (Xie, Kirillov *et al.* 2019), EfficientNet (Tan and Le 2019), RegNet (Radosavovic, Kosaraju *et al.* 2020), FBNetV2 (Wan, Dai *et al.* 2020), and EfficientNetV2 (Tan and Le 2021). It is also possible to do unsupervised neural architecture search (Liu, Dollár *et al.* 2020). Figure 5.45 shows the top-1% accuracy on ImageNet vs. the network size (# of parameters) and forward inference operation counts for a number of recent network architectures (Wan, Dai *et al.* 2020). Compared to the earlier networks shown in Figure 5.44, the newer networks use $10\times$ (or more) fewer parameters.

Deep learning software

Over the last decade, a large number of deep learning software frameworks and programming language extensions have been developed. The Wikipedia entry on deep learning software lists over twenty such frameworks, about a half of which are still being actively developed.⁴⁴ While Caffe (Jia, Shelhamer *et al.* 2014) was one of the first to be developed and used for computer vision applications, it has mostly been supplanted by PyTorch and TensorFlow, at least if we judge by the open-source implementations that now accompany most computer vision research papers.

⁴⁴https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software

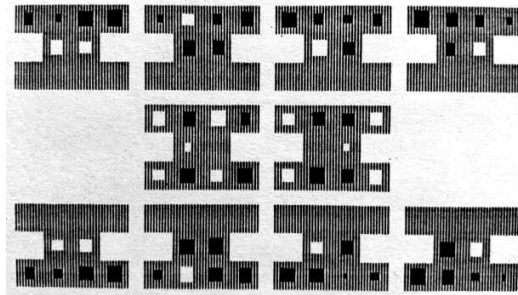


Figure 5.46 A Hinton diagram showing the weights connecting the units in a three layer neural network, courtesy of Geoffrey Hinton. The size of each small box indicates the magnitude of each weight and its color (black or white) indicates the sign.

Andrew Glassner’s (2018) introductory deep learning book uses the Keras library because of its simplicity. The Dive into Deep Learning book (Zhang, Lipton *et al.* 2021) and associated course (Smola and Li 2019) use MXNet for all the examples in the text, but they have recently released PyTorch and TensorFlow code samples as well. Stanford’s CS231n (Li, Johnson, and Yeung 2019) and Johnson (2020) include a lecture on the fundamentals of PyTorch and TensorFlow. Some classes also use simplified frameworks that require the students to implement more components, such as the Educational Framework (EDF) developed by McAllester (2020) and used in Geiger (2021).

In addition to software frameworks and libraries, deep learning code development usually benefits from good visualization libraries such as TensorBoard⁴⁵ and Visdom.⁴⁶ And in addition to the model zoos mentioned earlier in this section, there are even higher-level packages such as Classy Vision,⁴⁷ which allow you to train or fine-tune your own classifier with no or minimal programming. Andrej Karpathy also provides a useful guide for training neural networks at <http://karpathy.github.io/2019/04/25/recipe>, which may help avoid common issues.

5.4.5 Visualizing weights and activations

Visualizing intermediate and final results has always been an integral part of computer vision algorithm development (e.g., Figures 1.7–1.11) and is an excellent way to develop intuitions and debug or refine results. In this chapter, we have already seen examples of tools for simple

⁴⁵<https://www.tensorflow.org/tensorboard>

⁴⁶<https://github.com/fossasia/visdom>

⁴⁷<https://classyvision.ai>

two-input neural network visualizations, e.g., the TensorFlow Playground in Figure 5.32 and ConvNetJS in Figure 5.38. In this section, we discuss tools for visualizing network weights and, more importantly, the *response functions* of different units or layers in a network.

For a simple small network such as the one shown in Figure 5.32, we can indicate the strengths of connections using line widths and colors. What about networks with more units? A clever way to do this, called *Hinton diagrams* in honor of its inventor, is to indicate the strengths of the incoming and outgoing weights as black or white boxes of different sizes, as shown in Figure 5.46 (Ackley, Hinton, and Sejnowski 1985; Rumelhart, Hinton, and Williams 1986b).⁴⁸

If we wish to display the set of activations in a given layer, e.g., the response of the final 10-category layer in MNIST or CIFAR-10, across some or all of the inputs, we can use non-linear dimensionality reduction techniques such as t-SNE and UMap discussed in Section 5.2.4 and Figure 5.21.

How can we visualize what individual units (“neurons”) in a deep network respond to? For the first layer in a network (Figure 5.47, upper left corner), the response can be read directly from the incoming weights (grayish images) for a given channel. We can also find the patches in the validation set that produce the largest responses across the units in a given channel (colorful patches in the upper left corner of Figure 5.47). (Remember that in a convolutional neural network, different units in a particular channel respond similarly to shifted versions of the input, ignoring boundary and aliasing effects.)

For deeper layers in a network, we can again find maximally activating patches in the input images. Once these are found, Zeiler and Fergus (2014) pair a *deconvolution network* with the original network to backpropagate feature activations all the way back to the image patch, which results in the grayish images in layers 2–5 in Figure 5.47. A related technique called *guided backpropagation* developed by Springenberg, Dosovitskiy *et al.* (2015) produces slightly higher contrast results.

Another way to probe a CNN feature map is to determine how strongly parts of an input image activate units in a given channel. Zeiler and Fergus (2014) do this by masking sub-regions of the input image with a gray square, which not only produces activation maps, but can also show the most likely labels associated with each image region (Figure 5.48). Simonyan, Vedaldi, and Zisserman (2013) describe a related technique they call *saliency maps*, Nguyen, Yosinski, and Clune (2016) call their related technique *activation maximization*, and Selvaraju, Cogswell *et al.* (2017) call their visualization technique *gradient-weighted class activation mapping* (Grad-CAM).

⁴⁸In the early days of neural networks, bit-mapped displays and printers only supported 1-bit black and white images.

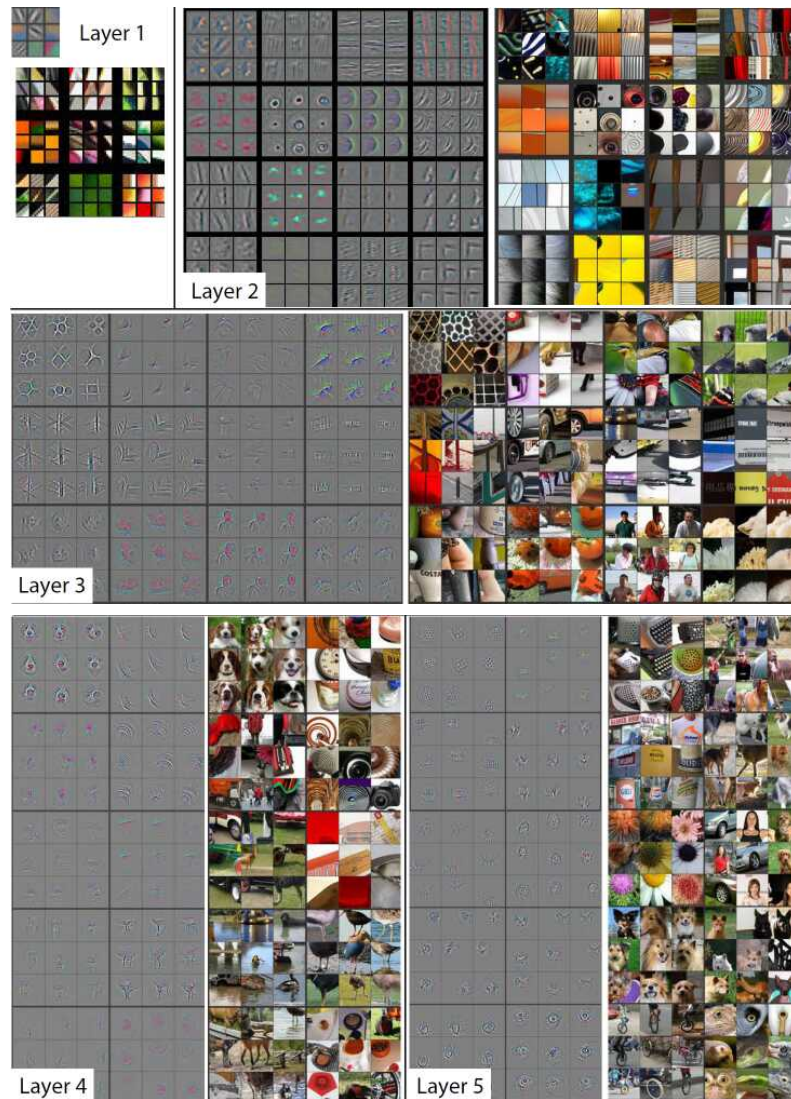


Figure 5.47 Visualizing network weights and features (Zeiler and Fergus 2014) © 2014 Springer. Each visualized convolutional layer is taken from a network adapted from the SuperVision net of Krizhevsky, Sutskever, and Hinton (2012). The 3×3 subimages denote the top nine responses in one feature map (channel in a given layer) projected back into pixel space (higher layers project to larger pixel patches), with the color images on the right showing the most responsive image patches from the validation set, and the grayish signed images on the left showing the corresponding maximum stimulus pre-images.

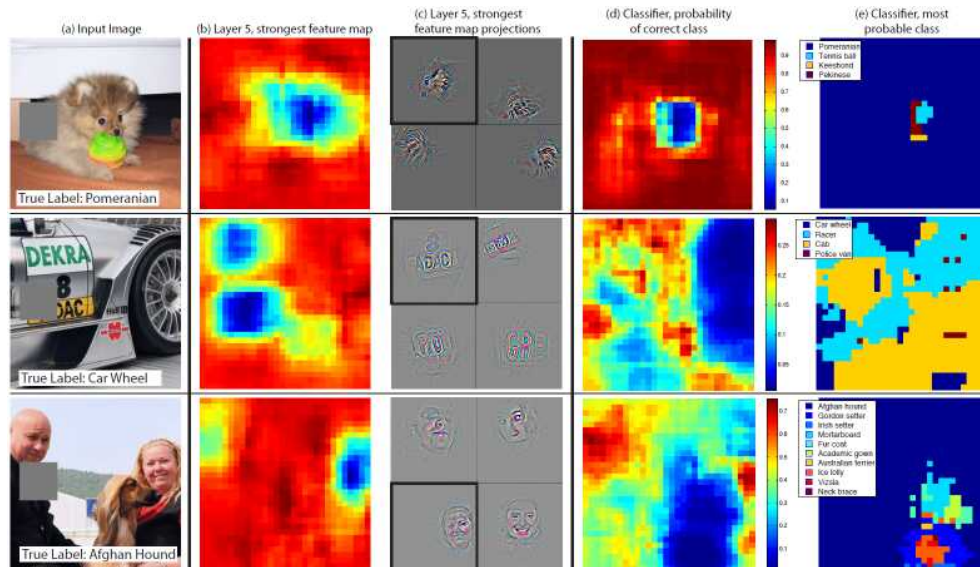


Figure 5.48 Heat map visualization from Zeiler and Fergus (2014) © 2014 Springer. By covering up portions of the input image with a small gray square, the response of a highly active channel in layer 5 can be visualized (second column), as can the feature map projections (third column), the likelihood of the correct class, and the most likely class per pixel.



Figure 5.49 Feature visualization of how GoogLeNet (Szegedy, Liu et al. 2015) trained on ImageNet builds up its representations over different layers, from Olah, Mordvintsev, and Schubert (2017).



Figure 5.50 Examples of adversarial images © Szegedy, Zaremba et al. (2013). For each original image in the left column, a small random perturbation (shown magnified by $10\times$ in the middle column) is added to obtain the image in the right column, which is always classified as an ostrich.

Many more techniques for visualizing neural network responses and behaviors have been described in various papers and blogs (Mordvintsev, Olah, and Tyka 2015; Zhou, Khosla et al. 2015; Nguyen, Yosinski, and Clune 2016; Bau, Zhou et al. 2017; Olah, Mordvintsev, and Schubert 2017; Olah, Satyanarayan et al. 2018; Cammarata, Carter et al. 2020), as well as the extensive lecture slides by Johnson (2020, Lecture 14). Figure 5.49 shows one example, visualizing different layers in a pre-trained GoogLeNet. OpenAI also recently released a great interactive tool called Microscope,⁴⁹ which allows people to visualize the significance of every neuron in many common neural networks.

5.4.6 Adversarial examples

While techniques such as guided backpropagation can help us better visualize neural network responses, they can also be used to “trick” deep networks into misclassifying inputs by subtly perturbing them, as shown in Figure 5.50. The key to creating such images is to take a set of final activations and to then backpropagate a gradient in the direction of the “fake” class, updating the input image until the fake class becomes the dominant activation. Szegedy, Zaremba et al. (2013) call such perturbed images *adversarial examples*.

Running this backpropagation requires access to the network and its weights, which means that this is a *white box attack*, as opposed to a *black box attack*, where nothing is

⁴⁹<https://microscope.openai.com/models>

known about the network. Surprisingly, however, the authors find “... that adversarial examples are relatively robust, and are shared by neural networks with varied number of layers, activations or trained on different subsets of the training data.”

The initial discovery of adversarial attacks spurred a flurry of additional investigations (Goodfellow, Shlens, and Szegedy 2015; Nguyen, Yosinski, and Clune 2015; Kurakin, Goodfellow, and Bengio 2016; Moosavi-Dezfooli, Fawzi, and Frossard 2016; Goodfellow, Papernot *et al.* 2017). Eykholt, Evtimov *et al.* (2018) show how adding simple stickers to real world objects (such as stop signs) can cause neural networks to misclassify photographs of such objects. Hendrycks, Zhao *et al.* (2021) have created a database of natural images that consistently fool popular deep classification networks trained on ImageNet. And while adversarial examples are mostly used to demonstrate the weaknesses of deep learning models, they can also be used to improve recognition (Xie, Tan *et al.* 2020).

Ilyas, Santurkar *et al.* (2019) try to demystify adversarial examples, finding that instead of making the anticipated large-scale perturbations that affect a human label, they are performing a type of shortcut learning (Lapuschkin, Wäldchen *et al.* 2019; Geirhos, Jacobsen *et al.* 2020). They find that optimizers are exploiting the *non-robust features* for an image label; that is, non-random correlations for an image class that exist in the dataset, but are not easily detected by humans. These non-robust features look merely like noise to a human observer, leaving images perturbed by them predominantly the same. Their claim is supported by training classifiers solely on non-robust features and finding that they correlate with image classification performance.

Are there ways to guard against adversarial attacks? The *cleverhans* software library (Papernot, Faghri *et al.* 2018) provides implementations of adversarial example construction techniques and adversarial training. There’s also an associated <http://www.cleverhans.io> blog on security and privacy in machine learning. Madry, Makelov *et al.* (2018) show how to train a network that is robust to bounded additive perturbations in known test images. There’s also recent work on detecting (Qin, Frosst *et al.* 2020b) and *deflecting* adversarial attacks (Qin, Frosst *et al.* 2020a) by forcing the perturbed images to visually resemble their (false) target class. This continues to be an evolving area, with profound implications for the robustness and safety of machine learning-based applications, as is the issue of *dataset bias* (Torralba and Efros 2011), which can be guarded against, to some extent, by testing *cross-dataset transfer* performance (Ranftl, Lasinger *et al.* 2020).

5.4.7 Self-supervised learning

As we mentioned previously, it is quite common to *pre-train* a *backbone* (or *trunk*) network for one task, e.g., whole image classification, and to then replace the final (few) layers with a

new *head* (or one or more *branches*), which are then trained for a different task, e.g., semantic image segmentation (He, Gkioxari *et al.* 2017). Optionally, the last few layers of the original backbone network can be *fine-tuned*.

The idea of training on one task and then using the learning on another is called *transfer learning*, while the process of modifying the final network to its intended application and statistics is called *domain adaptation*. While this idea was originally applied to backbones trained on labeled datasets such as ImageNet, i.e., in a *fully supervised* manner, the possibility of pre-training on the immensely larger set of unlabeled real-world images always remained a tantalizing possibility.

The central idea in *self-supervised* learning is to create a supervised *pretext task* where the labels can be automatically derived from unlabeled images, e.g., by asking the network to predict a subset of the information from the rest. Once pre-trained, the network can then be modified and fine-tuned on the final intended *downstream task*. Weng (2019) has a wonderful introductory blog post on this topic, and Zisserman (2018) has a great lecture, where the term *proxy task* is used. Additional good introductions can be found in the survey by Jing and Tian (2020) and the bibliography by Ren (2020).

Figure 5.51 shows some examples of pretext tasks that have been proposed for pre-training image classification networks. These include:

- Context prediction (Doersch, Gupta, and Efros 2015): take nearby image patches and predict their relative positions.
- Context encoders (Pathak, Krahenbuhl *et al.* 2016): inpaint one or more missing regions in an image.
- 9-tile jigsaw puzzle (Noroozi and Favaro 2016): rearrange the tiles into their correct positions.
- Colorizing black and white images (Zhang, Isola, and Efros 2016).
- Rotating images by multiples of 90° to make them upright (Gidaris, Singh, and Komodakis 2018). The paper compares itself against 11 previous self-supervised techniques.

In addition to using single-image pretext tasks, many researchers have used video clips, since successive frames contain semantically related content. One way to use this information is to order video frames correctly in time, i.e., to use a temporal version of context prediction and jigsaw puzzles (Misra, Zitnick, and Hebert 2016; Wei, Lim *et al.* 2018). Another is to extend colorization to video, with the colors in the first frame given (Vondrick,

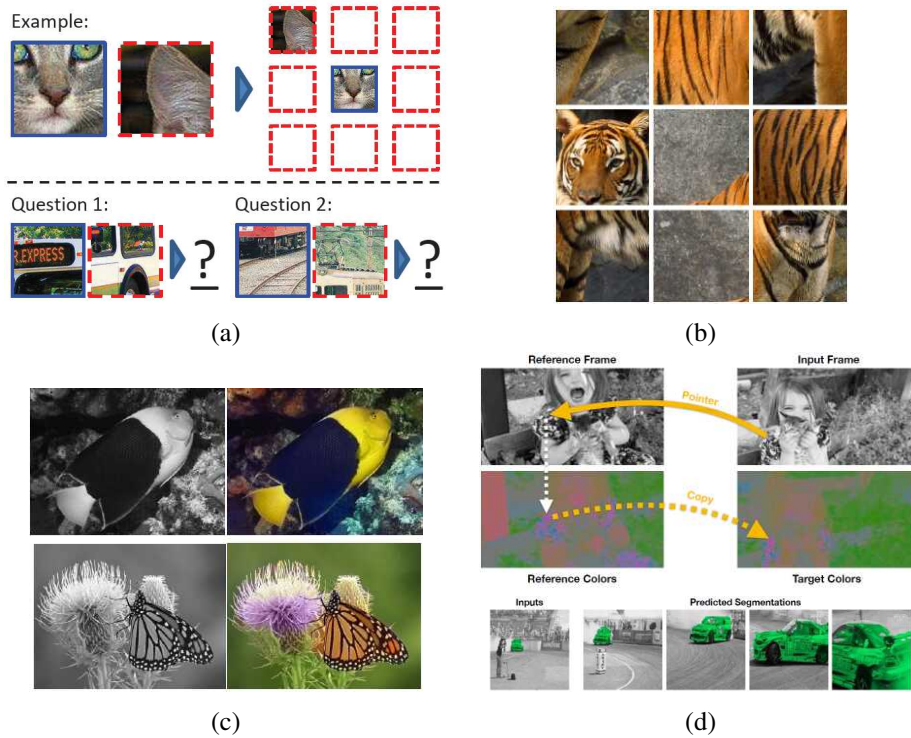


Figure 5.51 Examples of self-supervised learning tasks: (a) guessing the relative positions of image patches—can you guess the answers to Q1 and Q2? (Doersch, Gupta, and Efros 2015) © 2015 IEEE; (b) solving a nine-tile jigsaw puzzle (Noroozi and Favaro 2016) © 2016 Springer; (c) image colorization (Zhang, Isola, and Efros 2016) © 2016 Springer; (d) video color transfer for tracking (Vondrick, Shrivastava et al. 2018) © 2016 Springer.

Shrivastava *et al.* 2018), which encourages the network to learn semantic categories and correspondences. And since videos usually come with sounds, these can be used as additional cues in self-supervision, e.g., by asking a network to align visual and audio signals (Chung and Zisserman 2016; Arandjelovic and Zisserman 2018; Owens and Efros 2018), or in an unsupervised (contrastive) framework (Alwassel, Mahajan *et al.* 2020; Patrick, Asano *et al.* 2020).

Since self-supervised learning shows such great promise, an open question is whether such techniques could at some point surpass the performance of fully-supervised networks trained on smaller fully-labeled datasets.⁵⁰ Some impressive results have been shown using

⁵⁰https://people.eecs.berkeley.edu/~efros/gelato_bet.html

semi-supervised (weak) learning (Section 5.2.5) on very large (300M–3.5B) partially or noisily labeled datasets such as JFT-300M (Sun, Shrivastava *et al.* 2017) and Instagram hashtags (Mahajan, Girshick *et al.* 2018). Other researchers have tried simultaneously using supervised learning on labeled data and self-supervised pretext task learning on unlabeled data (Zhai, Oliver *et al.* 2019; Sun, Tzeng *et al.* 2019). It turns out that getting the most out of such approaches requires careful attention to dataset size, model architecture and capacity, and the extract details (and difficulty) of the pretext tasks (Kolesnikov, Zhai, and Beyer 2019; Goyal, Mahajan *et al.* 2019; Misra and Maaten 2020). At the same time, others are investigating how much real benefit pre-training actually gives in downstream tasks (He, Girshick, and Dollár 2019; Newell and Deng 2020; Feichtenhofer, Fan *et al.* 2021).

Semi-supervised training systems automatically generate ground truth labels for pretext tasks so that these can be used in a supervised manner (e.g. by minimizing classification errors). An alternative is to use unsupervised learning with a contrastive loss (Section 5.3.4) or other ranking loss (Gómez 2019) to encourage semantically similar inputs to produce similar encodings while spreading dissimilar inputs further apart. This is commonly now called *contrastive (metric) learning*.

Wu, Xiong *et al.* (2018) train a network to produce a separate embedding for each instance (training example), which they store in a moving average *memory bank* as new samples are fed through the neural network being trained. They then classify new images using nearest neighbors in the embedding space. Momentum Contrast (MoCo) replaces the memory bank with a fixed-length queue of encoded samples fed through a temporally adapted momentum encoder, which is separate from the actual network being trained (He, Fan *et al.* 2020). Pretext-invariant representation learning (PIRL) uses pretext tasks and “multi-crop” data augmentation, but then compares their outputs using a memory bank and contrastive loss (Misra and Maaten 2020). SimCLR (simple framework for contrastive learning) uses fixed mini-batches and applies a contrastive loss (normalized temperature cross-entropy, similar to (5.58)) between each sample in the batch and all the other samples, along with aggressive data augmentation (Chen, Kornblith *et al.* 2020). MoCo v2 combines ideas from MoCo and SimCLR to obtain even better results (Chen, Fan *et al.* 2020). Rather than directly comparing the generated embeddings, a fully connected network (MLP) is first applied.

Contrastive losses are a useful tool in metric learning, since they encourage distances in an embedding space to be small for semantically related inputs. An alternative is to use deep clustering to similarly encourage related inputs to produce similar outputs (Caron, Bojanowski *et al.* 2018; Ji, Henriques, and Vedaldi 2019; Asano, Rupprecht, and Vedaldi 2020; Gidaris, Bursuc *et al.* 2020; Yan, Misra *et al.* 2020). Some of the latest results using clustering for unsupervised learning now produce results competitive with contrastive metric learning

and also suggest that the kinds of data augmentation being used are even more important than the actual losses that are chosen (Caron, Misra *et al.* 2020; Tian, Chen, and Ganguli 2021). In the context of vision and language (Section 6.6), CLIP (Radford, Kim *et al.* 2021) has achieved remarkable generalization for image classification using contrastive learning and “natural-language supervision.” With a dataset of 400 million text and image pairs, their task is to take in a single image and a random sample of 32,768 text snippets and predict which text snippet is truly paired with the image.

Interestingly, it has recently been discovered that representation learning that *only* enforces similarity between semantically similar inputs also works well. This seems counter-intuitive, because without negative pairs as in contrastive learning, the representation can easily collapse to trivial solutions by predicting a constant for any input and maximizing similarity. To avoid this collapse, careful attention is often paid to the network design. Bootstrap Your Own Latent (BYOL) (Grill, Strub *et al.* 2020) shows that with a momentum encoder, an extra predictor MLP on the online network side, and a stop-gradient operation on the target network side, one can successfully remove the negatives from MoCo v2 training. SimSiam (Chen and He 2021) further shown that even the momentum encoder is not required and only a stop-gradient operation is sufficient for the network to learn meaningful representations. While both systems jointly train the predictor MLP and the encoder with gradient updates, it has been even more recently shown that the predictor weights can be directly set using statistics of the input right before the predictor layer (Tian, Chen, and Ganguli 2021). Feichtenhofer, Fan *et al.* (2021) compare a number of these unsupervised representation learning techniques on a variety of video understanding tasks and find that the learned spatiotemporal representations generalize well to different tasks.

Contrastive learning and related work rely on compositions of data augmentations (e.g. color jitters, random crops, etc.) to learn representations that are *invariant* to such changes (Chen and He 2021). An alternative attempt is to use generative modeling (Chen, Radford *et al.* 2020), where the representations are pre-trained by predicting pixels either in an auto-regressive (GPT- or other language model) manner or a de-noising (BERT-, masked auto-encoder) manner. Generative modeling has the potential to bridge self-supervised learning across domains from vision to NLP, where scalable pre-trained models are now dominant.

One final variant on self-supervised learning is using a student-teacher model, where the teacher network is used to provide training examples to a student network. These kinds of architectures were originally called *model compression* (Bucila, Caruana, and Niculescu-Mizil 2006) and *knowledge distillation* (Hinton, Vinyals, and Dean 2015) and were used to produce smaller models. However, when coupled with additional data and larger capacity networks, they can also be used to improve performance. Xie, Luong *et al.* (2020) train

an EfficientNet (Tan and Le 2019) on the labeled ImageNet training set, and then use this network to label an additional 300M unlabeled images. The true labels and pseudo-labeled images are then used to train a higher-capacity “student”, using regularization (e.g., dropout) and data augmentation to improve generalization. The process is then repeated to yield further improvements.

In all, self-supervised learning is currently one of the most exciting sub-areas in deep learning,⁵¹ and many leading researchers believe that it may hold the key to even better deep learning (LeCun and Bengio 2020). To explore implementations further, VISSL provides open-source PyTorch implementations of many state-of-the-art self-supervised learning models (with weights) that were described in this section.⁵²

5.5 More complex models

While deep neural networks started off being used in 2D image understanding and processing applications, they are now also widely used for 3D data such as medical images and video sequences. We can also chain a series of deep networks together in time by feeding the results from one time frame to the next (or even forward-backward). In this section, we look first at three-dimensional convolutions and then at recurrent models that propagate information forward or bi-directionally in time. We also look at *generative models* that can synthesize completely new images from semantic or related inputs.

5.5.1 Three-dimensional CNNs

As we just mentioned, deep neural networks in computer vision started off being used in the processing of regular two-dimensional images. However, as the amount of video being shared and analyzed increases, deep networks are also being applied to video understanding, which we discuss in more detail Section 6.5. We are also seeing applications in three-dimensional volumetric models such as occupancy maps created from range data (Section 13.5) and volumetric medical images (Section 6.4.1).

It may appear, at first glance, that the convolutional networks we have already studied, such as the ones illustrated in Figures 5.33, 5.34, and 5.39 already perform 3D convolutions, since their input receptive fields are 3D boxes in (x, y, c) , where c is the (feature) *channel* dimension. So, we could in principle fit a sliding window (say in time, or elevation) into a 2D network and be done. Or, we could use something like *grouped convolutions*. However,

⁵¹<https://sites.google.com/view/self-supervised-icml2019>

⁵²<https://github.com/facebookresearch/vissl>

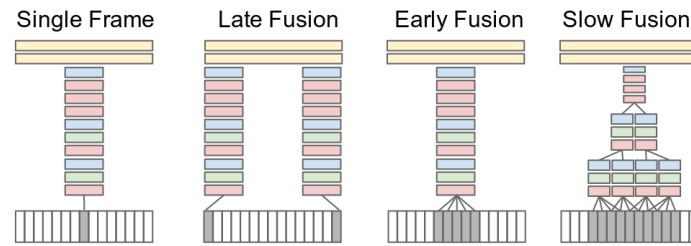


Figure 5.52 *Alternative approaches to information fusion over the temporal dimensions (Karpathy, Toderici et al. 2014) © 2014 IEEE.*

it's more convenient to operate on a complete 3D volume all at once, and to have *weight sharing* across the third dimension for all kernels, as well as multiple input and output feature channels at each layer.

One of the earliest applications of 3D convolutions was in the processing of video data to classify human actions (Kim, Lee, and Yang 2007; Ji, Xu *et al.* 2013; Baccouche, Mamalet *et al.* 2011). Karpathy, Toderici *et al.* (2014) describe a number of alternative architectures for fusing temporal information, as illustrated in Figure 5.52. The single frame approach classifies each frame independently, depending purely on that frame's content. Late fusion takes features generated from each frame and makes a per-clip classification. Early fusion groups small sets of adjacent frames into multiple channels in a 2D CNN. As mentioned before, the interactions across time do not have the convolutional aspects of weight sharing and temporal shift invariance. Finally, 3D CNNs (Ji, Xu *et al.* 2013) (not shown in this figure) learn 3D space and time-invariance kernels that are run over spatio-temporal windows and fused into a final score.

Tran, Bourdev *et al.* (2015) show how very simple $3 \times 3 \times 3$ convolutions combined with pooling in a deep network can be used to obtain even better performance. Their C3D network can be thought of as the “VGG of 3D CNNs” (Johnson 2020, Lecture 18). Carreira and Zisserman (2017) compare this architecture to alternatives that include *two-stream* models built by analyzing pixels and optical flows in parallel pathways (Figure 6.44b). Section 6.5 on video understanding discusses these and other architectures used for such problems, which have also been attacked using sequential models such as recurrent neural networks (RNNs) and LSTM, which we discuss in Section 5.5.2. Lecture 18 on video understanding by Johnson (2020) has a nice review of all these video understanding architectures.

In addition to video processing, 3D convolutional neural networks have been applied to volumetric image processing. Two examples of shape modeling and recognition from range data, i.e., 3D ShapeNets (Wu, Song *et al.* 2015) and VoxNet (Maturana and Scherer 2015)

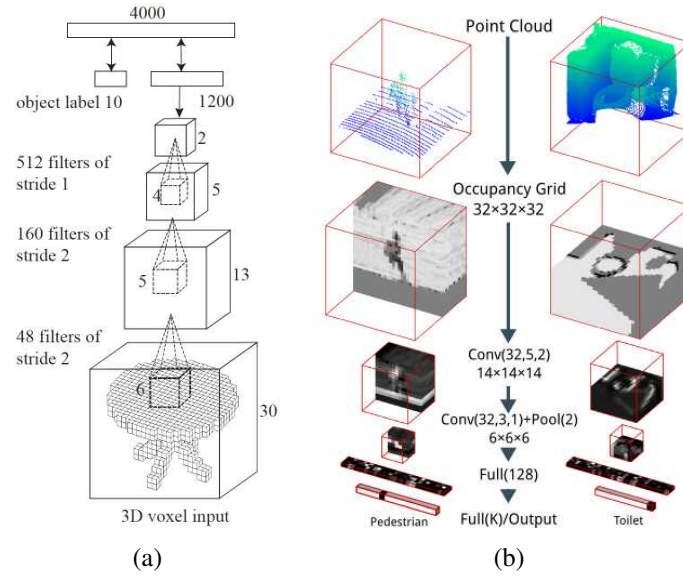


Figure 5.53 3D convolutional networks applied to volumetric data for object detection: (a) 3D ShapeNets (Wu, Song *et al.* 2015) © 2015 IEEE; (b) VoxNet (Maturana and Scherer 2015) © 2015 IEEE.

are shown in Figure 5.53. Examples of their application to medical image segmentation (Kamnitsas, Ferrante *et al.* 2016; Kamnitsas, Ledig *et al.* 2017) are discussed in Section 6.4.1. We discuss neural network approaches to 3D modeling in more detail in Sections 13.5.1 and 14.6.

Like regular 2D CNNs, 3D CNN architectures can exploit different spatial and temporal resolutions, striding, and channel depths, but they can be very computation and memory intensive. To counteract this, Feichtenhofer, Fan *et al.* (2019) develop a two-stream SlowFast architecture, where a slow pathway operates at a lower frame rate and is combined with features from a fast pathway with higher temporal sampling but fewer channels (Figure 6.44c). Video processing networks can also be made more efficient using channel-separated convolutions (Tran, Wang *et al.* 2019) and neural architecture search (Feichtenhofer 2020). Multigrid techniques (Appendix A.5.3) can also be used to accelerate the training of video recognition models (Wu, Girshick *et al.* 2020).

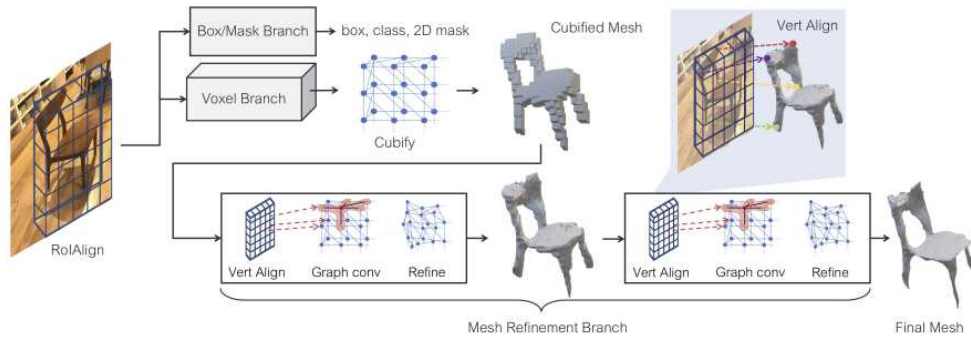


Figure 5.54 Overview of the Mesh R-CNN system (Gkioxari, Malik, and Johnson 2019) © 2019 IEEE. A Mask R-CNN backbone is augmented with two 3D shape inference branches. The voxel branch predicts a coarse shape for each detected object, which is further deformed with a sequence of refinement stages in the mesh refinement branch.

3D point clouds and meshes

In addition to processing 3D gridded data such as volumetric density, implicit distance functions, and video sequences, neural networks can be used to infer 3D models from single images. One approach is to predict per-pixel depth, which we study in Section 12.8. Another is to reconstruct full 3D models represented using volumetric density (Choy, Xu *et al.* 2016), which we study in Sections 13.5.1 and 14.6. Some more recent experiments, however, suggest that some of these 3D inference networks (Tatarchenko, Dosovitskiy, and Brox 2017; Groueix, Fisher *et al.* 2018; Richter and Roth 2018) may just be recognizing the general object category and doing a small amount of fitting (Tatarchenko, Richter *et al.* 2019).

Generating and processing 3D point clouds has also been extensively studied (Fan, Su, and Guibas 2017; Qi, Su *et al.* 2017; Wang, Sun *et al.* 2019). Guo, Wang *et al.* (2020) provide a comprehensive survey that reviews over 200 publications in this area.

A final alternative is to infer 3D triangulated meshes from either RGB-D (Wang, Zhang *et al.* 2018) or regular RGB (Gkioxari, Malik, and Johnson 2019; Wickramasinghe, Fua, and Knott 2021) images. Figure 5.54 illustrates the components of the Mesh R-CNN system, which detects images of 3D objects and turns each one into a triangulated mesh after first reconstructing a volumetric model. The primitive operations and representations needed to process such meshes using deep neural networks can be found in the PyTorch3D library.⁵³

⁵³<https://github.com/facebookresearch/pytorch3d>

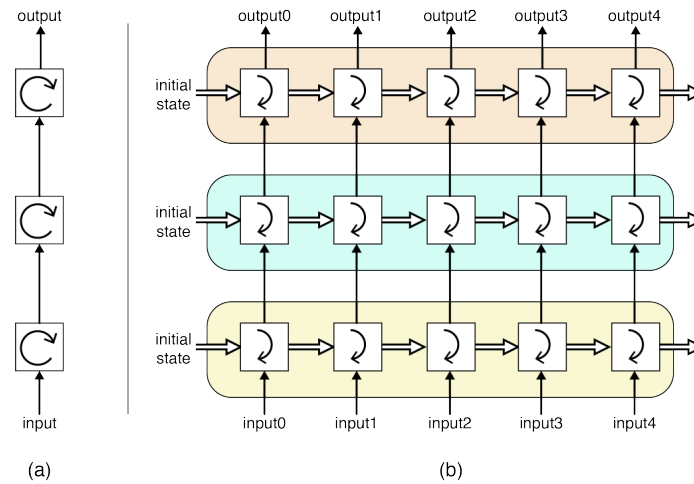


Figure 5.55 A deep recurrent neural network (RNN) uses multiple stages to process sequential data, with the output of one stage feeding the input of the next © Glassner (2018). Each stage maintains its own state and backpropagates its own gradients, although weights are shared between all stages. Column (a) shows a more compact rolled-up diagram, while column (b) shows the corresponding unrolled version.

5.5.2 Recurrent neural networks

While 2D and 3D convolutional networks are a good fit for images and volumes, sometimes we wish to process a *sequence* of images, audio signals, or text. A good way to exploit previously seen information is to pass features detected at one time instant (e.g., video frame) as input to the next frame’s processing. Such architectures are called Recurrent Neural Networks (RNNs) and are described in more detail in Goodfellow, Bengio, and Courville (2016, Chapter 10) and Zhang, Lipton *et al.* (2021, Chapter 8). Figure 5.55 shows a schematic sketch of such an architecture. Deep network layers not only pass information on to subsequent layers (and an eventual output), but also feed some of their information as input to the layer processing the next frame of data. Individual layers share weights across time (a bit like 3D convolution kernels), and backpropagation requires computing derivatives for all of the “unrolled” units (time instances) and summing these derivatives to obtain weight updates.

Because gradients can propagate for a long distance backward in time, and can therefore vanish or explode (just as in deep networks before the advent of residual networks), it is also possible to add extra *gating* units to modulate how information flows between frames. Such architectures are called *Gated Recurrent Units* (GRUs) and *Long short-term memory* (LSTM)

(Hochreiter and Schmidhuber 1997; Zhang, Lipton *et al.* 2021, Chapter 9).

RNNs and LSTMs are often used for video processing, since they can fuse information over time and model temporal dependencies (Baccouche, Mamalet *et al.* 2011; Donahue, Hendricks *et al.* 2015; Ng, Hausknecht *et al.* 2015; Srivastava, Mansimov, and Salakhudinov 2015; Ballas, Yao *et al.* 2016), as well as language modeling, image captioning, and visual question answering. We discuss these topics in more detail in Sections 6.5 and 6.6. They have also occasionally been used to merge multi-view information in stereo (Yao, Luo *et al.* 2019; Riegler and Koltun 2020a) and to simulate iterative low algorithms in a fully differentiable (and hence trainable) manner (Hur and Roth 2019; Teed and Deng 2020b).

To propagate information forward in time, RNNs, GRUs, and LSTMs need to encode *all* of the potentially useful previous information in the hidden state being passed between time steps. In some situations, it is useful for a sequence modeling network to look further back (or even forward) in time. This kind of capability is often called *attention* and is described in more detail in Zhang, Lipton *et al.* (2021, Chapter 10), Johnson (2020, Lecture 12), and Section 5.5.3 on transformers. In brief, networks with attention store lists of *keys* and *values*, which can be probed with a *query* to return a weighted blend of values depending on the alignment between the query and each key. In this sense, they are similar to *kernel regression* (4.12–4.14), which we studied in Section 4.1.1, except that the query and the keys are multiplied (with appropriate weights) before being passed through a softmax to determine the blending weights.

Attention can either be used to look backward at the hidden states in previous time instances (which is called *self-attention*), or to look at different parts of the image (*visual attention*, as illustrated in Figure 6.46). We discuss these topics in more detail in Section 6.6 on vision and language. When recognizing or generating sequences, such as the words in a sentence, attention modules often used to work in tandem with sequential models such as RNNs or LSTMs. However, more recent works have made it possible to apply attention to the entire input sequence in one parallel step, as described in Section 5.5.3 on transformers.

The brief descriptions in this section just barely skim the broad topic of deep sequence modeling, which is usually covered in several lectures in courses on deep learning (e.g., Johnson 2020, Lectures 12–13) and several chapters in deep learning textbooks (Zhang, Lipton *et al.* 2021, Chapters 8–10). Interested readers should consult these sources for more detailed information.

5.5.3 Transformers

Transformers, which are a novel architecture that adds attention mechanisms (which we describe below) to deep neural networks, were first introduced by Vaswani, Shazeer *et al.* (2017)

in the context of neural machine translation, where the task consists of translating text from one language to another (Mikolov, Sutskever *et al.* 2013). In contrast to RNNs and their variants (Section 5.5.2), which process input tokens one at a time, transformers can operate on the entire input sequence at once. In the years after first being introduced, transformers became the dominant paradigm for many tasks in natural language processing (NLP), enabling the impressive results produced by BERT (Devlin, Chang *et al.* 2018), RoBERTa (Liu, Ott *et al.* 2019), and GPT-3 (Brown, Mann *et al.* 2020), among many others. Transformers then began seeing success when processing the natural language component and later layers of many vision and language tasks (Section 6.6). More recently, they have gained traction in pure computer vision tasks, even outperforming CNNs on several popular benchmarks.

The motivation for applying transformers to computer vision is different than that of applying it to NLP. Whereas RNNs suffer from sequentially processing the input, convolutions do not have this problem, as their operations are already inherently parallel. Instead, the problem with convolutions has to do with their *inductive biases*, i.e., the default assumptions encoded into convolutional models.

A convolution operation assumes that nearby pixels are more important than far away pixels. Only after several convolutional layers are stacked together does the receptive field grow large enough to attend to the entire image (Araujo, Norris, and Sim 2019), unless the network is endowed with non-local operations (Wang, Girshick *et al.* 2018) similar to those used in some image denoising algorithms (Buades, Coll, and Morel 2005a). As we have seen in this chapter, convolution’s spatial locality bias has led to remarkable success across many aspects of computer vision. But as datasets, models, and computational power grow by orders of magnitude, these inductive biases may become a factor inhibiting further progress.⁵⁴

The fundamental component of a transformer is *self-attention*, which is itself built out of applying *attention* to each of N unit activations in a given layer in the network.⁵⁵ Attention is often described using an analogy to the concept of *associative maps* or *dictionaries* found as data structures in programming languages and databases. Given a set of *key-value pairs*, $\{(\mathbf{k}_i, \mathbf{v}_i)\}$ and a query $\mathbf{q}$, a dictionary returns the value $\mathbf{v}_i$ corresponding to the key $\mathbf{k}_i$ that exactly matches the query. In neural networks, the key and query values are real-valued vectors (e.g., linear projections of activations), so the corresponding operation returns a weighted sum of values where the weights depend on the pairwise distances between a query and the set of keys. This is basically the same as *scattered data interpolation*, which we studied in

⁵⁴Rich Sutton, a pioneer in reinforcement learning, believes that learning to leverage computation, instead of encoding human knowledge, is *the bitter lesson* to learn from the history of AI research (Sutton 2019). Others disagree with this view, believing that it is essential to be able to learn from small amounts of data (Lake, Salakhutdinov, and Tenenbaum 2015; Marcus 2020).

⁵⁵ N may indicate the number of words in a sentence or patches in an image

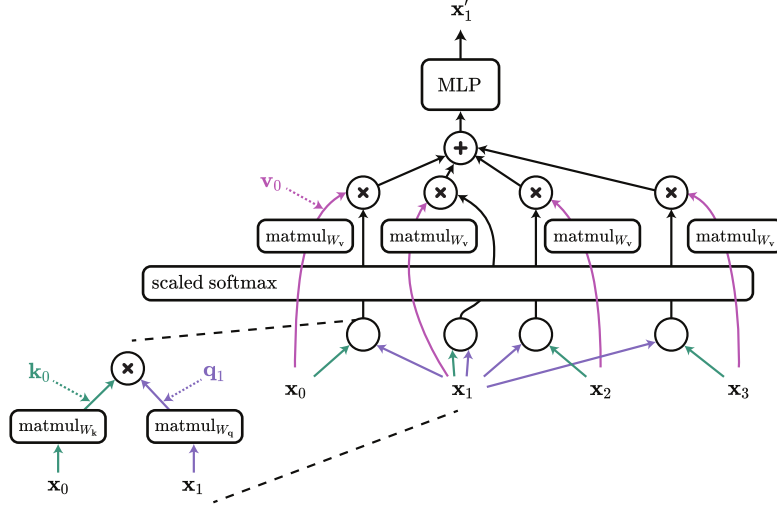


Figure 5.56 The self-attention computation graph to compute a single output vector x'_2 , courtesy of Matt Deitke, adapted from Vaswani, Huang, and Manning (2019). Note that the full self-attention operation also computes outputs for x'_1 , x'_3 , and x'_4 by shifting the input to the query (x_2 in this case) between x_1 , x_3 , and x_4 , respectively. For each of matmul_V , matmul_K , and matmul_Q , there is a single matrix of weights that gets reused with each call.

Section 4.1.1, as pointed out in Zhang, Lipton *et al.* (2021, Section 10.2). However, instead of using radial distances as in (4.14), attention mechanisms in neural networks more commonly use *scaled dot-product attention* (Vaswani, Shazeer *et al.* 2017; Zhang, Lipton *et al.* 2021, Section 10.3.3), which involves taking the dot product between the query and key vectors, scaling down by the square root of the dimension of these embeddings D ,⁵⁶ and then applying the softmax function of (5.5), i.e.,

$$\mathbf{y} = \sum_i \alpha(\mathbf{q} \cdot \mathbf{k}_i / D) \mathbf{v}_i = \text{softmax}(\mathbf{q}^T \mathbf{K} / D)^T \mathbf{V}, \quad (5.77)$$

where $\mathbf{K}$ and $\mathbf{V}$ are the row-stacked matrices composed of the key and value vectors, respectively, and $\mathbf{y}$ is the output of the attention operator.⁵⁷

Given a set of input vectors $\{x_0, x_1, \dots, x_{N-1}\}$, the self-attention operation produces a set of output vectors $\{x'_0, x'_1, \dots, x'_{N-1}\}$. Figure 5.56 shows the case for $N = 4$, where

⁵⁶We divide the dot product by $\sqrt{D}$ so that the variance of the scaled dot product does not increase for larger embedding dimensions, which could result in vanishing gradients.

⁵⁷The partition of unity function α notation is borrowed from Zhang, Lipton *et al.* (2021, Section 10.3).

the self-attention computation graph is used to obtain a single output vector $\mathbf{x}'_2$. As pictured, self-attention uses three learned weight matrices, $\mathbf{W}_q$, $\mathbf{W}_k$, and $\mathbf{W}_v$, which determine the

$$\mathbf{q}_i = \mathbf{W}_q \mathbf{x}_i, \mathbf{k}_i = \mathbf{W}_k \mathbf{x}_i, \text{ and } \mathbf{v}_i = \mathbf{W}_v \mathbf{x}_i \quad (5.78)$$

per-unit query, key, and value vectors going into each attention block. The weighted sum of values is then optionally passed through a multi-layer perceptron (MLP) to produce $\mathbf{x}'_2$.

In comparison to a fully connected or convolutional layer, self-attention computes each output (e.g., $\mathbf{x}'_i$) based on all of the input vectors $\{\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{N-1}\}$. In that sense, it is often compared to a fully connected layer, but instead of the weights being fixed for each input, the weights are adapted on the spot, based on the input (Khan, Naseer *et al.* 2021). Compared to convolutions, self-attention is able to attend to every part of the input from the start, instead of constraining itself to local regions of the input, which may help it introduce the kind of context information needed to disambiguate the objects shown in Figure 6.8.

There are several components that are combined with self-attention to produce a transformer block, as described in Vaswani, Shazeer *et al.* (2017). The full transformer consists of both an encoder and a decoder block, although both share many of the same components. In many applications, an encoder can be used without a decoder (Devlin, Chang *et al.* 2018; Dosovitskiy, Beyer *et al.* 2021) and vice versa (Razavi, van den Oord, and Vinyals 2019).

The right side of Figure 5.57 shows an example of a transformer encoder block. For both the encoder and decoder:

- Instead of modeling set-to-set operations, we can model sequence-to-sequence operations by adding a **positional encoding** to each input vector (Gehring, Auli *et al.* 2017). The positional encoding typically consists of a set of temporally shifted sine waves from which position information can be decoded. (Such position encodings have also recently been added to implicit neural shape representations, which we study in Sections 13.5.1 and 14.6.)
- In lieu of applying a single self-attention operation to the input, multiple self-attention operations, with different learned weight matrices to build different keys, values, and queries, are often joined together to form **multi-headed self-attention** (Vaswani, Shazeer *et al.* 2017). The result of each head is then concatenated together before everything is passed through an MLP.
- **Layer normalization** (Ba, Kiros, and Hinton 2016) is then applied to the output of the MLP. Each vector may then independently be passed through another MLP with shared weights before layer normalization is applied again.

- **Residual connections** (He, Zhang *et al.* 2016a) are employed after multi-headed attention and after the final MLP.

During training, the biggest difference in the decoder is that some of the input vectors to self-attention may be masked out, which helps support parallel training in autoregressive prediction tasks. Further exposition of the details and implementation of the transformer architecture is provided in Vaswani, Shazeer *et al.* (2017) and in the additional reading (Section 5.6).

A key challenge of applying transformers to the image domain has to do with the size of image input (Vaswani, Shazeer *et al.* 2017). Let N denote the length of the input, D denote the number of dimensions for each input entry, and K denote a convolution's (on side) kernel size.⁵⁸ The number of floating point operations (FLOPs) required for self-attention is on the order of $O(N^2D)$, whereas the FLOPs for a convolution operation is on the order of $O(ND^2K^2)$. For instance, with an ImageNet image scaled to size $224 \times 224 \times 3$, if each pixel is treated independently, $N = 224 \times 224 = 50176$ and $D = 3$. Here, a convolution is *significantly* more efficient than self-attention. In contrast, applications like neural machine translation may only have N as the number of words in a sentence and D as the dimension for each word embedding (Mikolov, Sutskever *et al.* 2013), which makes self-attention much more efficient.

The Image Transformer (Parmar, Vaswani *et al.* 2018) was the first attempt at applying the full transformer model to the image domain, with many of the same authors that introduced the transformer. It used both an encoder and decoder to try and build an autoregressive generative model that predicts the next pixel, given a sequence of input pixels and all the previously predicted pixels. (The earlier work on non-local networks by Wang, Girshick *et al.* (2018) also used ideas inspired by transformers, but with a simpler attention block and a fully two-dimensional setup.) Each vector input to the transformer corresponded to a single pixel, which ultimately constrained them to generate small images (i.e., 32×32), since the quadratic cost of self-attention was too expensive otherwise.

Dosovitskiy, Beyer *et al.* (2021) had a breakthrough that allowed transformers to process much larger images. Figure 5.57 shows the diagram of the model, named the Vision Transformer (ViT). For the task of image recognition, instead of treating each pixel as a separate input vector to the transformer, they divide an image (of size 224×224) into 196 distinct 16×16 gridded image patches. Each patch is then flattened, and passed through a shared embedding matrix, which is equivalent to a strided 16×16 convolution, and the results are combined with a positional encoding vector and then passed to the transformer. Earlier work

⁵⁸In Section 5.4 on convolutional architectures, we use C to denote the number of channels instead of D to denote the embedding dimensions.

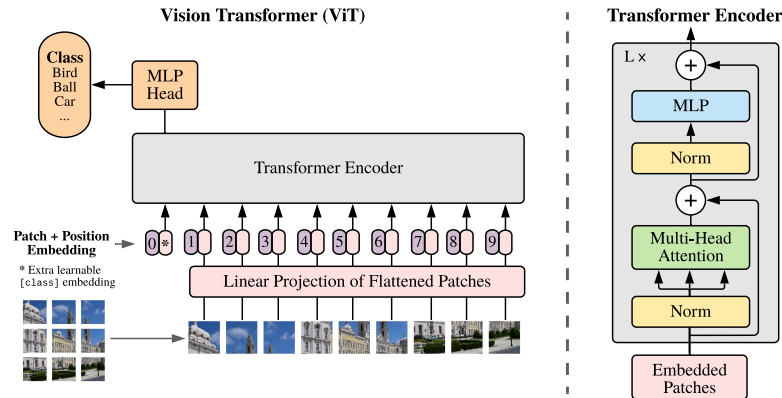


Figure 5.57 The Vision Transformer (ViT) model from (Dosovitskiy, Beyer et al. 2021) breaks an image into a 16×16 grid of patches. Each patch is then flattened, passed through a shared embedding matrix, and combined with a positional encoding vector. These inputs are then passed through a transformer encoder (right) several times before predicting an image's class.

from Cordonnier, Loukas, and Jaggi (2019) introduced a similar patching approach, but on a smaller scale with 2×2 patches.

ViT was only able to outperform their convolutional baseline BiT (Kolesnikov, Beyer et al. 2020) when using over 100 million training images from JFT-300M (Sun, Shrivastava et al. 2017). When using ImageNet alone, or a random subset of 10 or 30 million training samples from JPT-300, the ViT model typically performed much worse than the BiT baseline. Their results suggest that in low-data domains, the inductive biases present in convolutions are typically quite useful. But, with orders of magnitude of more data, a transformer model might discover even better representations that are not representable with a CNN.

Some works have also gone into combining the inductive biases of convolutions with transformers (Srinivas, Lin et al. 2021; Wu, Xiao et al. 2021; Lu, Batra et al. 2019; Yuan, Guo et al. 2021). An influential example of such a network is DETR (Carion, Massa et al. 2020), which is applied to the task of object detection. It first processes the image with a ResNet backbone, with the output getting passed to a transformer encoder-decoder architecture. They find that the addition of a transformer improves the ability to detect large objects, which is believed to be because of its ability to reason globally about correspondences between inputted encoding vectors.

The application and usefulness of transformers in the realm of computer vision is still being widely researched. Already, however, they have achieved impressive performance on

a wide range of tasks, with new papers being published rapidly.⁵⁹ Some more notable applications include image classification (Liu, Lin *et al.* 2021; Touvron, Cord *et al.* 2020), object detection (Dai, Cai *et al.* 2020; Liu, Lin *et al.* 2021), image pre-training (Chen, Radford *et al.* 2020), semantic segmentation (Zheng, Lu *et al.* 2020), pose recognition (Li, Wang *et al.* 2021), super-resolution (Zeng, Fu, and Chao 2020), colorization (Kumar, Weissenborn, and Kalchbrenner 2021), generative modeling (Jiang, Chang, and Wang 2021; Hudson and Zitnick 2021), and video classification (Arnab, Dehghani *et al.* 2021; Fan, Xiong *et al.* 2021; Li, Zhang *et al.* 2021). Recent works have also found success extending ViT’s patch embedding to pure MLP vision architectures (Tolstikhin, Houlsby *et al.* 2021; Liu, Dai *et al.* 2021; Touvron, Bojanowski *et al.* 2021). Applications to vision and language are discussed in Section 6.6.

5.5.4 Generative models

Throughout this chapter, I have mentioned that machine learning algorithms such as logistic regression, support vector machines, random trees, and feedforward deep neural networks are all examples of *discriminative* systems that never form an explicit *generative* model of the quantities they are trying to estimate (Bishop 2006, Section 1.5; Murphy 2012, Section 8.6). In addition to the potential benefits of generative models discussed in these two textbooks, Goodfellow (2016) and Kingma and Welling (2019) list some additional ones, such as the ability to visualize our assumptions about our unknowns, training with missing or incompletely labeled data, and the ability to generate multiple, alternative, results.

In computer graphics, which is sometimes called *image synthesis* (as opposed to the *image understanding* or *image analysis* we do in computer vision), the ability to easily generate realistic random images and models has long been an essential tool. Examples of such algorithms include texture synthesis and style transfer, which we study in more detail in Section 10.5, as well as fractal terrain (Fournier, Fussell, and Carpenter 1982) and tree generation (Prusinkiewicz and Lindenmayer 1996). Examples of deep neural networks being used to generate such novel images, often under user control, are shown in Figures 5.60 and 10.58. Related techniques are also used in the nascent field of *neural rendering*, which we discuss in Section 14.6.

How can we unlock the demonstrated power of deep neural networks to capture semantics in order to visualize sample images and generate new ones? One approach could be to use the visualization techniques introduced in Section 5.4.5. But as you can see from Figure 5.49, while such techniques can give us insights into individual units, they fail to create

⁵⁹<https://github.com/dk-liang/Awesome-Visual-Transformer>

fully realistic images.

Another approach might be to construct a *decoder* network to undo the classification performed by the original (*encoder*) network. This kind of “bottleneck” architecture is widely used, as shown in Figure 5.37a, to derive semantic per-pixel labels from images. Can we use a similar idea to generate realistic looking images?

Variational autoencoders

A network that encodes an image into small compact codes and then attempts to decode it back into the same image is called an *autoencoder*. The compact codes are typically represented as a vector, which is often called the *latent* vector to emphasize that it is hidden and unknown. Autoencoders have a long history of use in neural networks, even predating today’s feedforward networks (Kingma and Welling 2019). It was once believed that this might be a good way to pre-train networks, but the more challenging proxy tasks we studied in Section 5.4.7 have proven to be more effective.

At a high level, to train an autoencoder on a dataset of images, we can use an unsupervised objective that tries to have the output image of the decoder match the training image input to the encoder. To generate a new image, we can then randomly sample a latent vector and hope that from that vector, the decoder can generate a new image that looks like it came from the distribution of training images in our dataset.

With an autoencoder, there is a deterministic, one-to-one mapping from each input to its latent vector. Hence, the number of latent vectors that are generated exactly matches the number of input data points. If the encoder’s objective is to produce a latent vector that makes it easy to decode, one possible solution would be for every latent vector to be extremely far away from every other latent vector. Here, the decoder can overfit all the latent vectors it has seen since they would all be unique with little overlap. However, as our goal is to randomly generate latent vectors that can be passed to the decoder to generate realistic images, we want the latent space to both be well explored and to encode some meaning, such as nearby vectors being semantically similar. Ghosh, Sajjadi *et al.* (2019) propose one potential solution, where they inject noise into the latent vector and empirically find that it works quite well.

Another extension of the autoencoder is the *variational autoencoder* (VAE) (Kingma and Welling 2013; Rezende, Mohamed, and Wierstra 2014; Kingma and Welling 2019). Instead of generating a single latent vector for each input, it generates the mean and covariance that define a chosen distribution of latent vectors. The distribution can then be sampled from to produce a single latent vector, which gets passed into the decoder. To avoid having the covariance matrix become the zero matrix, making the sampling process deterministic, the objective function often includes a regularization term to penalize the distribution if it is far

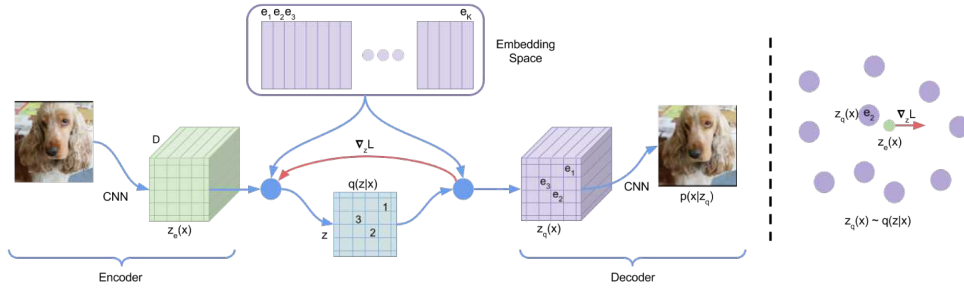


Figure 5.58 The VQ-VAE model. On the left, $z_e(x)$ represents the output of the encoder, the embedding space on top represents the codebook of K embedding vectors, and $q(z|x)$ represents the process of replacing each spatial (i.e., channel-wise) vector in the output of the encoder with its nearest vector in the codebook. On the right, we see how a $z_e(x)$ vector (green) may be rounded to e_2 , and that the gradient in the encoder network (red) may push the vector away from e_2 during backpropagation. © van den Oord, Vinyals, and Kavukcuoglu (2017)

from some chosen (e.g., Gaussian) distribution. Due to their probabilistic nature, VAEs can explore the space of possible latent vectors significantly better than autoencoders, making it harder for the decoder to overfit the training data.

Motivated by how natural language is discrete and by how images can typically be described in language (Section 6.6), the vector quantized VAE (VQ-VAE) of van den Oord, Vinyals, and Kavukcuoglu (2017) takes the approach of modeling the latent space with categorical variables. Figure 5.58 shows an outline of the VQ-VAE architecture. The encoder and decoder operate like a normal VAE, where the encoder predicts some latent representation from the input, and the decoder generates an image from the latent representation. However, in contrast to the normal VAE, the VQ-VAE replaces each spatial dimension of the predicted latent representation with its nearest vector from a discrete set of vectors (named the *codebook*). The discretized latent representation is then passed to the decoder. The vectors in the codebook are trained simultaneously with the VAE’s encoder and decoder. Here, the codebook vectors are optimized to move closer to the spatial vectors outputted by the encoder.

Although a VQ-VAE uses a discrete codebook of vectors, the number of possible images it can represent is still monstrously large. In some of their image experiments, they set the size of the codebook to $K = 512$ vectors and set the size of the latent variable to be $z = 32 \times 32 \times 1$. Here, they can represent $512^{32 \cdot 32 \cdot 1}$ possible images.

Compared to a VAE, which typically assumes a Gaussian latent distribution, the latent

distribution of a VQ-VAE is not as clearly defined, so a separate generative model is trained to sample latent variables z . The model is trained on the final latent variables outputted from the trained VQ-VAE encoder across the training data. For images, entries in z are often spatially dependent, e.g., an object may be encoded over many neighboring entries. With entries being chosen from a discrete codebook of vectors, we can use a PixelCNN (van den Oord, Kalchbrenner *et al.* 2016) to autoregressively sample new entries in the latent variable based on previously sampled neighboring entries. The PixelCNN can also be conditionally trained, which enables the ability to sample latent variables corresponding to a particular image class or feature.

A follow-up to the VQ-VAE model, named VQ-VAE-2 (Razavi, van den Oord, and Vinyals 2019), uses a two-level approach to decoding images, where with both a small and large latent vector, they can get much higher fidelity reconstructed and generated images. Section 6.6 discusses Dall-E (Ramesh, Pavlov *et al.* 2021), a model that applies VQ-VAE-2 to text-to-image generation and achieves remarkable results.

Generative adversarial networks

Another possibility for image synthesis is to use the multi-resolution features computed by pre-trained networks to match the statistics of a given texture or style image, as described in Figure 10.57. While such networks are useful for matching the *style* of a given artist and the high-level *content* (layout) of a photograph, they are not sufficient to generate completely photorealistic images.

In order to create truly photorealistic synthetic images, we want to determine if an image is real(istic) or fake. If such a loss function existed, we could use it to train networks to generate synthetic images. But, since such a loss function is incredibly difficult to write by hand, why not train a separate neural network to play the critic role? This is the main insight behind the *generative adversarial networks* introduced by Goodfellow, Pouget-Abadie *et al.* (2014). In their system, the output of the *generator* network G is fed into a separate *discriminator* network D , whose task is to tell “fake” synthetically generated images apart from real ones, as shown in Figure 5.59a. The goal of the generator is to create images that “fool” the discriminator into accepting them as real, while the goal of the discriminator is to catch the “forger” in their act. Both networks are co-trained simultaneously, using a blend of loss functions that encourage each network to do its job. The joint loss function can be written as

$$E_{\text{GAN}}(\mathbf{w}_G, \mathbf{w}_D) = \sum_n \log D(\mathbf{x}_n) + \log (1 - D(G(\mathbf{z}_n))), \quad (5.79)$$

where the $\{\mathbf{x}_n\}$ are the real-world training images, $\{\mathbf{z}_n\}$ are random vectors, which are

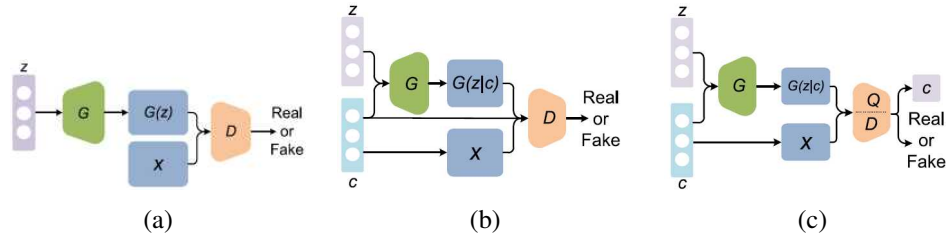


Figure 5.59 Generative adversarial network (GAN) architectures from Pan, Yu et al. (2019) © 2019 IEEE. (a) In a regular GAN, random “latent” noise vectors z are fed into a generator network G , which produces synthetic “fake” images $x' = G(z)$. The job of the discriminator D is to tell the fake images apart from real samples x . (b) In a conditional GAN (cGAN), the network iterates (during training) over all the classes that we wish to synthesize. The generator G gets both a class id c and a random noise vector z as input, and the discriminator D gets the class id as well and needs to determine if its input is a real member of the given class. (c) The discriminator in an InfoGAN does not have access to the class id, but must instead infer it from the samples it is given.

passed through the generator G to produce synthetic images x'_n , and the $\{w_G, w_D\}$ are the weights (parameters) in the generator and discriminator.

Instead of minimizing this loss, we adjust the weights of the generator to minimize the second term (they do not affect the first), and adjust the weights of the discriminator to *maximize* both terms, i.e., minimize the discriminator’s error. This process is often called a *min-imax game*.⁶⁰ More details about the formulation and how to optimize it can be found in the original paper by Goodfellow, Pouget-Abadie et al. (2014), as well as deep learning textbooks (Zhang, Lipton et al. 2021, Chapter 17), lectures (Johnson 2020, Lecture 20), tutorials (Goodfellow, Isola et al. 2018), and review articles (Creswell, White et al. 2018; Pan, Yu et al. 2019).

The original paper by Goodfellow, Pouget-Abadie et al. (2014) used a small, fully connected network to demonstrate the basic idea, so it could only generate 32×32 images such as MNIST digits and low-resolution faces. The *Deep Convolutional GAN* (DCGAN) introduced by Radford, Metz, and Chintala (2015) uses the second half of the deconvolution network shown in Figure 5.37a to map from the random latent vectors z to arbitrary size images and can therefore generate a much wider variety of outputs, while LAPGAN uses

⁶⁰Note that the term *adversarial* in GANs refers to this adversarial game between the generator and the discriminator, which helps the generator create better pictures. This is distinct from the *adversarial examples* we discussed in Section 5.4.6, which are images designed to fool recognition systems.

a Laplacian pyramid of adversarial networks (Denton, Chintala *et al.* 2015). Blending between different latent vectors (or perturbing them in certain directions) generates in-between synthetic images.

GANs and DCGANs can be trained to generate new samples from a given class, but it is even more useful to generate samples from different classes using the same trained network. The *conditional GAN* (cGAN) proposed by Mirza and Osindero (2014) achieves this by feeding a class vector into both the generator, which *conditions* its output on this second input, as well as the discriminator, as shown in Figure 5.59b. It is also possible to make the discriminator predict classes that correlate with the class vector using an extra mutual information term, as shown in Figure 5.59c (Chen, Duan *et al.* 2016). This allows the resulting InfoGAN network to learn disentangled representations, such as the digit shapes and writing styles in MNIST, or pose and lighting.

While generating random images can have many useful graphics applications, such as generating textures, filling holes, and stylizing photographs, as discussed in Section 10.5, it becomes even more useful when it can be done under a person's artistic control (Lee, Zitnick, and Cohen 2011). The iGAN interactive image editing system developed by Zhu, Krähenbühl *et al.* (2016) does this by learning a manifold of photorealistic images using a generative adversarial network and then constraining user edits (or even sketches) to produce images that lie on this manifold.

This approach was generalized by Isola, Zhu *et al.* (2017) to all kinds of other image-to-image translation tasks, as shown in Figure 5.60a. In their pix2pix system, images, which can just be sketches or semantic labels, are fed into a modified U-Net, which converts them to images with different semantic meanings or styles (e.g., photographs or road maps). When the input is a semantic label map and the output is a photorealistic image, this process is often called *semantic image synthesis*. The translation network is trained with a conditional GAN, which takes paired images from the two domains at training time and has the discriminator decide if the synthesized (translated) image together with the input image are a real or fake pair. Referring back to Figure 5.59b, the class c is now a complete image, which is fed into both G and the discriminator D , along with its paired or synthesized output. Instead of making a decision for the whole image, the discriminator looks at overlapping patches and makes decisions on a patch-by-patch basis, which requires fewer parameters and provides more training data and more discriminative feedback. In their implementation, there is no random vector z ; instead, dropout is used during both training and “test” (translation) time, which is equivalent to injecting noise at different levels in the network.

In many situations, paired images are not available, e.g., when you have collections of paintings and photographs from different locations, or pictures of animals in two different

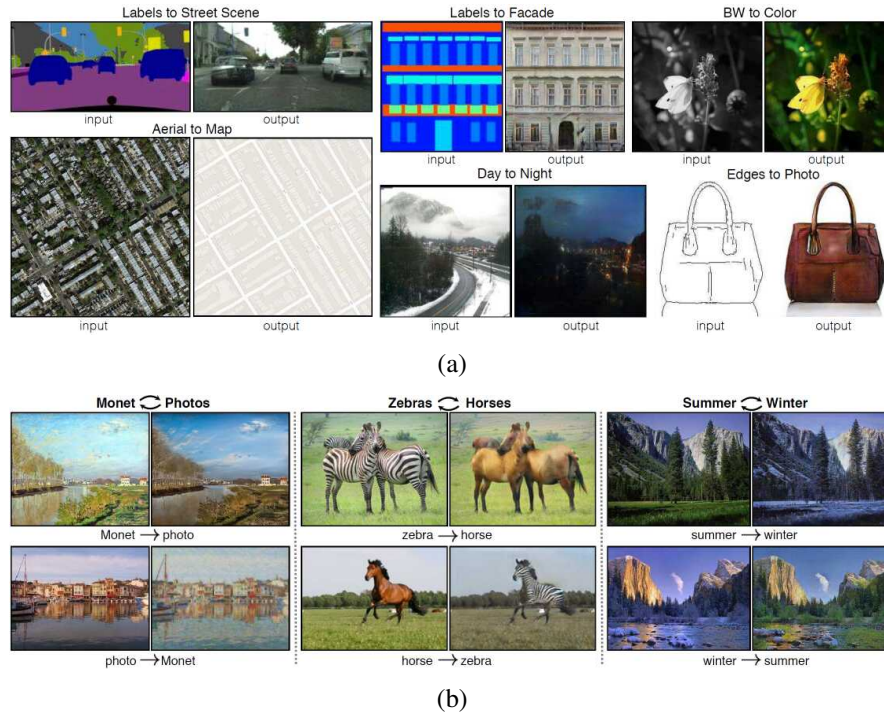


Figure 5.60 Image-to-image translation. (a) Given paired training images, the original pix2pix system learns how to turn sketches into photos, semantic maps to images, and other pixel remapping tasks (Isola, Zhu et al. 2017) © 2017 IEEE. (b) CycleGAN does not require paired training images, just collections coming from different sources, such as painting and photographs or horses and zebras (Zhu, Park et al. 2017) © 2017 IEEE.

classes, as shown in Figure 5.60b. In this case, a cycle-consistent adversarial network (CycleGAN) can be used to require the mappings between the two domains to encourage identity, while also ensuring that generated images are perceptually similar to the training images (Zhu, Park et al. 2017). DualGAN (Yi, Zhang et al. 2017) and DiscoGAN (Kim, Cha et al. 2017) use related ideas. The BicycleGAN system of Zhu, Zhang et al. (2017) uses a similar idea of transformation cycles to encourage encoded latent vectors to correspond to different modes in the outputs for better interpretability and control.

Since the publication of the original GAN paper, the number of extensions, applications, and follow-on papers has exploded. The GAN Zoo website⁶¹ lists over 500 GAN papers published between 2014 and mid-2018, at which point it stopped being updated. Large number

⁶¹<https://github.com/hindupuravinash/the-gan-zoo>

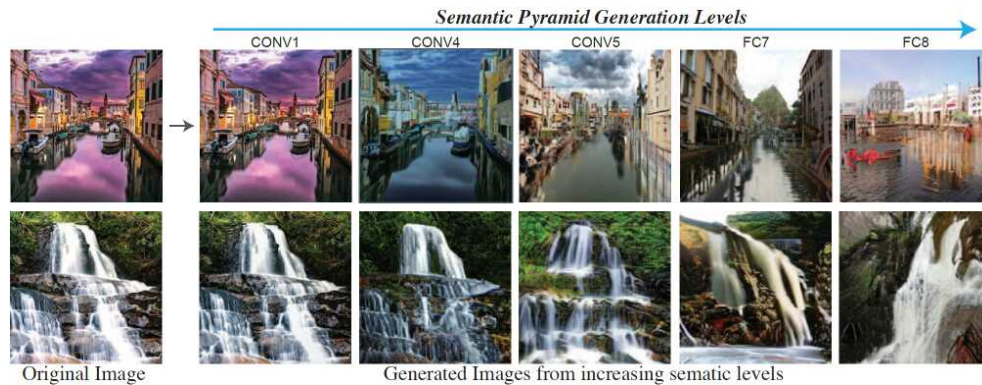


Figure 5.61 The Semantic Image Pyramid can be used to choose which semantic level in a deep network to modify when editing an image (Shocher, Gandelsman *et al.* 2020) © 2020 IEEE.

of papers continue to appear each year in vision, machine learning, and graphics conferences.

Some of the more important papers since 2017 include Wasserstein GANs (Arjovsky, Chintala, and Bottou 2017), Progressive GANs (Karras, Aila *et al.* 2018), UNIT (Liu, Breuel, and Kautz 2017) and MUNIT (Huang, Liu *et al.* 2018), spectral normalization (Miyato, Kataoka *et al.* 2018), SAGAN (Zhang, Goodfellow *et al.* 2019), BigGAN (Brock, Donahue, and Simonyan 2019), StarGAN (Choi, Choi *et al.* 2018) and StyleGAN (Karras, Laine, and Aila 2019) and follow-on papers (Choi, Uh *et al.* 2020; Karras, Laine *et al.* 2020; Viazovetskyi, Ivashkin, and Kashin 2020), SPADE (Park, Liu *et al.* 2019), GANSpace (Härkönen, Hertzmann *et al.* 2020), and VQGAN (Esser, Rombach, and Ommer 2020). You can find more detailed explanations and references to many more papers in the lectures by Johnson (2020, Lecture 20), tutorials by Goodfellow, Isola *et al.* (2018), and review articles by Creswell, White *et al.* (2018), Pan, Yu *et al.* (2019), and Tewari, Fried *et al.* (2020).

In summary, generative adversarial networks and their myriad extensions continue to be an extremely vibrant and useful research area, with applications such as image super-resolution (Section 10.3), photorealistic image synthesis (Section 10.5.3), image-to-image translation, and interactive image editing. Two very recent examples of this last application are the Semantic Pyramid for Image Generation by Shocher, Gandelsman *et al.* (2020), in which the semantic manipulation level can be controlled (from small texture changes to higher-level layout changes), as shown in Figure 5.61, and the Swapping Autoencoder by Park, Zhu *et al.* (2020), where structure and texture can be independently edited.

5.6 Additional reading

Machine learning and deep learning are rich, broad subjects which properly deserve their own course of study to master. Fortunately, there are a large number of good textbooks and online courses available to learn this material.

My own favorite for machine learning is the book by Bishop (2006), since it provides a broad treatment with a Bayesian flavor and excellent figures, which I have re-used in this book. The books by Glassner (2018, 2021) provide an even gentler introduction to both classic machine learning and deep learning, as well as additional figures I reference in this book. Two additional widely used textbooks for machine learning are Hastie, Tibshirani, and Friedman (2009) and Murphy (2012). Deisenroth, Faisal, and Ong (2020) provide a nice compact treatment of mathematics for machine learning, including linear and matrix algebra, probability theory, model fitting, regression, PCA, and SVMs, with a more in-depth exposition than the terse summaries I provide in this book. The book on Automated Machine Learning edited by Hutter, Kotthoff, and Vanschoren (2019) surveys automated techniques for designing and optimizing machine learning algorithms.

For deep learning, Goodfellow, Bengio, and Courville (2016) were the first to provide a comprehensive treatment, but it has not recently been revised. Glassner (2018, 2021) provides a wonderful introduction to deep learning, with lots of figures and no equations. I recommend it even to experienced practitioners since it helps develop and solidify intuitions about how learning works. An up-to-date reference on deep learning is the *Dive into Deep Learning* online textbook by Zhang, Lipton *et al.* (2021), which comes with interactive Python notebooks sprinkled throughout the text, as well as an associated course (Smola and Li 2019). Some introductory courses to deep learning use Charniak (2019).

Rawat and Wang (2017) provide a nice review article on deep learning, including a history of early and later neural networks, as well in-depth discussion of many deep learning components, such as pooling, activation functions, losses, regularization, and optimization. Additional surveys related to advances in deep learning include Sze, Chen *et al.* (2017), Elsken, Metzen, and Hutter (2019), Gu, Wang *et al.* (2018), and Choudhary, Mishra *et al.* (2020). Sejnowski (2018) provides an in-depth history of the early days of neural networks.

The Deep Learning for Computer Vision course slides by Johnson (2020) are an outstanding reference and a great way to learn the material, both for the depth of their information and how up-to-date the presentations are kept. They are based on Stanford's CS231n course (Li, Johnson, and Yeung 2019), which is also a great up-to-date source. Additional classes on deep learning with slides and/or video lectures include Grosse and Ba (2019), McAllester (2020), Leal-Taixé and Nießner (2020), Leal-Taixé and Nießner (2021), and Geiger (2021)

For transformers, [Bloem \(2019\)](#) provides a nice starting tutorial on implementing the standard transformer encoder and decoder block in PyTorch, from scratch. More comprehensive surveys of transformers applied to computer vision include [Khan, Naseer *et al.* \(2021\)](#) and [Han, Wang *et al.* \(2020\)](#). [Tay, Dehghani *et al.* \(2020\)](#) provides an overview of many attempts to reduce the quadratic cost of self-attention. [Wightman \(2021\)](#) makes available a fantastic collection of computer vision transformer implementations in PyTorch, with pre-trained weights and great documentation. Additional course lectures introducing transformers with videos and slides include [Johnson \(2020, Lecture 13\)](#), [Vaswani, Huang, and Manning \(2019, Lecture 14\)](#) and [LeCun and Canziani \(2020, Week 12\)](#).

For GANs, the new deep learning textbook by [Zhang, Lipton *et al.* \(2021, Chapter 17\)](#), lectures by [Johnson \(2020, Lecture 20\)](#), tutorials by [Goodfellow, Isola *et al.* \(2018\)](#), and review articles by [Creswell, White *et al.* \(2018\)](#), [Pan, Yu *et al.* \(2019\)](#), and [Tewari, Fried *et al.* \(2020\)](#) are all good sources. For a survey of the latest visual recognition techniques, the tutorials presented at ICCV ([Xie, Girshick *et al.* 2019](#)), CVPR ([Girshick, Kirillov *et al.* 2020](#)), and ECCV ([Xie, Girshick *et al.* 2020](#)) are excellent up-to-date sources.

5.7 Exercises

Ex 5.1: Backpropagation and weight updates. Implement the forward activation, backward gradient and error propagation, and weight update steps in a simple neural network. You can find examples of such code in HW3 of the 2020 UW CSE 576 class⁶² or the Educational Framework (EDF) developed by [McAllester \(2020\)](#) and used in [Geiger \(2021\)](#).

Ex 5.2: LeNet. Download, train, and test a simple “LeNet” ([LeCun, Bottou *et al.* 1998](#)) convolutional neural network on the CIFAR-10 ([Krizhevsky 2009](#)) or Fashion MNIST ([Xiao, Rasul, and Vollgraf 2017](#)) datasets. You can find such code in numerous places on the web, including HW4 of the 2020 UW CSE 576 class or the PyTorch beginner tutorial on Neural Networks.⁶³

Modify the network to remove the non-linearities. How does the performance change? Can you improve the performance of the original network by increasing the number of channels, layers, or convolution sizes? Do the training and testing accuracies move in the same or different directions as you modify your network?

Ex 5.3: Deep learning textbooks. Both the *Deep Learning: From Basics to Practice* book by [Glassner \(2018, Chapters 15, 23, and 24\)](#) and the *Dive into Deep Learning* book by [Zhang,](#)

⁶²<https://courses.cs.washington.edu/courses/cse576/20sp/calendar/>

⁶³https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html

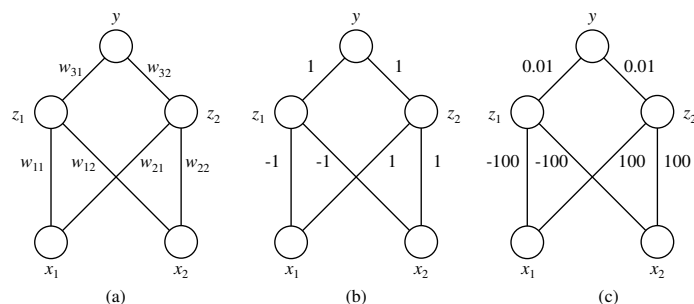


Figure 5.62 Simple two hidden unit network with a ReLU activation function and no bias parameters for regressing the function $y = |x_1 + 1.1x_2|$: (a) can you guess a set of weights that would fit this function?; (b) a reasonable set of starting weights; (c) a poorly scaled set of weights.

Lipton *et al.* (2021) contain myriad graded exercises with code samples to develop your understanding of deep neural networks. If you have the time, try to work through most of these.

Ex 5.4: Activation and weight scaling. Consider the two hidden unit network shown in Figure 5.62, which uses ReLU activation functions and has no additive bias parameters. Your task is to find a set of weights that will fit the function

$$y = |x_1 + 1.1x_2|. \quad (5.80)$$

1. Can you guess a set of weights that will fit this function?
2. Starting with the weights shown in column b, compute the activations for the hidden and final units as well as the regression loss for the nine input values $(x_1, x_2) \in \{-1, 0, 1\} \times \{-1, 0, 1\}$.
3. Now compute the gradients of the squared loss with respect to all six weights using the backpropagation chain rule equations (5.65–5.68) and sum them up across the training samples to get a final gradient.
4. What step size should you take in the gradient direction, and what would your update squared loss become?
5. Repeat this exercise for the initial weights in column (c) of Figure 5.62.
6. Given this new set of weights, how much worse is your error decrease, and how many iterations would you expect it to take to achieve a reasonable solution?

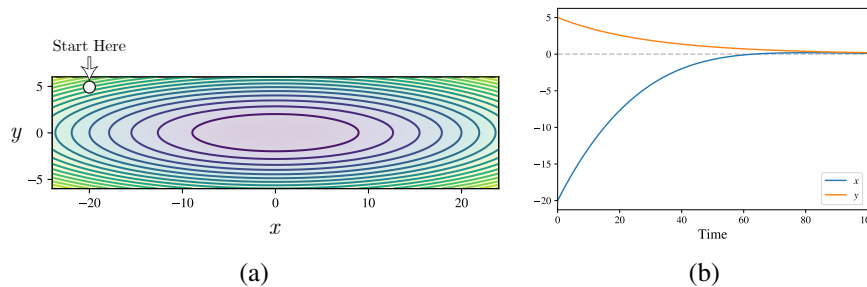


Figure 5.63 Function optimization: (a) the contour plot of $f(x, y) = x^2 + 20y^2$ with the function being minimized at $(0, 0)$; (b) ideal gradient descent optimization that quickly converges towards the minimum at $x = 0, y = 0$.

7. Would batch normalization help in this case?

Note: the following exercises were suggested by Matt Deitke.

Ex 5.5: Function optimization. Consider the function $f(x, y) = x^2 + 20y^2$ shown in Figure 5.63a. Begin by solving for the following:

1. Calculate ∇f , i.e., the gradient of f .
2. Evaluate the gradient at $x = -20, y = 5$.

Implement some of the common gradient descent optimizers, which should take you from the starting point $x = -20, y = 5$ to near the minimum at $x = 0, y = 0$. Try each of the following optimizers:

1. Standard gradient descent.
2. Gradient descent with momentum, starting with the momentum term as $\rho = 0.99$.
3. Adam, starting with decay rates of $\beta_1 = 0.9$ and $b_2 = 0.999$.

Play around with the learning rate α . For each experiment, plot how x and y change over time, as shown in Figure 5.63b.

How do the optimizers behave differently? Is there a single learning rate that makes all the optimizers converge towards $x = 0, y = 0$ in under 200 steps? Does each optimizer monotonically trend towards $x = 0, y = 0$?

Ex 5.6: Weight initialization. For an arbitrary neural network, is it possible to initialize the weights of a neural network such that it will never train on any non-trivial task, such as image classification or object detection? Explain why or why not.

Ex 5.7: Convolutions. Consider convolving a $256 \times 256 \times 3$ image with 64 separate convolution kernels. For kernels with heights and widths of $\{(3 \times 3), (5 \times 5), (7 \times 7), \text{ and } (9 \times 9)\}$, answer each of the following:

1. How many parameters (i.e., weights) make up the convolution operation?
2. What is the output size after convolving the image with the kernels?

Ex 5.8: Data augmentation. The figure below shows image augmentations that translate and scale an image.



Let CONV denote a convolution operation, f denote an arbitrary function (such as scaling or translating an image), and IMAGE denote the input image. A function f has invariance, with respect to a convolution, when $\text{CONV}(\text{IMAGE}) = \text{CONV}(f(\text{IMAGE}))$, and equivariance when $\text{CONV}(f(\text{IMAGE})) = f(\text{CONV}(\text{IMAGE}))$. Answer and explain each of the following:

1. Are convolutions translation invariant?
2. Are convolutions translation equivariant?
3. Are convolutions scale invariant?
4. Are convolutions scale equivariant?

Ex 5.9: Training vs. validation. Suppose your model is performing significantly better on the training data than it is on the validation data. What changes might be made to the loss function, training data, and network architecture to prevent such overfitting?

Ex 5.10: Cascaded convolutions. With only a single matrix multiplication, how can multiple convolutional kernel's convolve over an entire input image? Here, let the input image be of size $256 \times 256 \times 3$ and each of the 64 kernels be of size $3 \times 3 \times 3$.⁶⁴

Ex 5.11: Pooling vs. 1×1 convolutions. Pooling layers and 1×1 convolutions are both commonly used to shrink the size of the proceeding layer. When would you use one over the other?

⁶⁴Hint: You will need to reshape the input and each convolution's kernel size.

Ex 5.12: Inception. Why is an inception module more efficient than a residual block? What are the comparative disadvantages of using an inception module?

Ex 5.13: ResNets. Why is it easier to train a ResNet with 100 layers than a VGG network with 100 layers?

Ex 5.14: U-Nets. An alternative to the U-Net architecture is to not change the size of the height and width intermediate activations throughout the network. The final layer would then be able to output the same transformed pixel-wise representation of the input image. What is the disadvantage of this approach?

Ex 5.15: Early vs. late fusion in video processing. What are two advantages of early fusion compared to late fusion?

Ex 5.16: Video-to-video translation. Independently pass each frame in a video through a pix2pix model. For instance, if the video is of the day, then the output might be each frame at night. Stitch the output frames together to form a video. What do you notice? Does the video look plausible?

Ex 5.17: Vision Transformer. Using a Vision Transformer (ViT) model, pass several images through it and create a histogram of the activations after each layer normalization operation. Do the histograms tend to form of a normal distribution?

Ex 5.18: GAN training. In the GAN loss formulation, suppose the discriminator D is near-perfect, such that it correctly outputs near 1 for real images $\mathbf{x}_n$ and near 0 for synthetically generated images $G(\mathbf{z}_n)$.

1. For both the discriminator and the generator, compute its approximate loss with

$$\mathcal{L}_{\text{GAN}}(\mathbf{x}_n, \mathbf{z}_n) = \log D(\mathbf{x}_n) + \log(1 - D(G(\mathbf{z}_n))), \quad (5.81)$$

where the discriminator tries to minimize $\mathcal{L}_{\text{GAN}}$ and the generator tries to maximize $\mathcal{L}_{\text{GAN}}$.

2. How well can this discriminator be used to train the generator?
3. Can you modify the generator's loss function, $\min \log(1 - D(G(\mathbf{z}_n)))$, such that it is easier to train with both a great discriminator and a discriminator that is no better than random?⁶⁵

⁶⁵*Hint:* The loss function should suggest a relatively large change to fool a great discriminator and a relatively small change with a discriminator that is no better than random.

Ex 5.19: Colorization. Even though large amounts of unsupervised data can be collected for image colorization, it often does not train well using a pixel-wise regression loss between an image's predicted colors and its true colors. Why is that? Is there another loss function that may be better suited for the problem?

Chapter 6

Recognition

6.1	Instance recognition	346
6.2	Image classification	349
6.2.1	Feature-based methods	350
6.2.2	Deep networks	358
6.2.3	<i>Application: Visual similarity search</i>	360
6.2.4	Face recognition	363
6.3	Object detection	370
6.3.1	Face detection	371
6.3.2	Pedestrian detection	376
6.3.3	General object detection	379
6.4	Semantic segmentation	387
6.4.1	<i>Application: Medical image segmentation</i>	390
6.4.2	Instance segmentation	391
6.4.3	Panoptic segmentation	392
6.4.4	<i>Application: Intelligent photo editing</i>	394
6.4.5	Pose estimation	395
6.5	Video understanding	396
6.6	Vision and language	400
6.7	Additional reading	409
6.8	Exercises	413

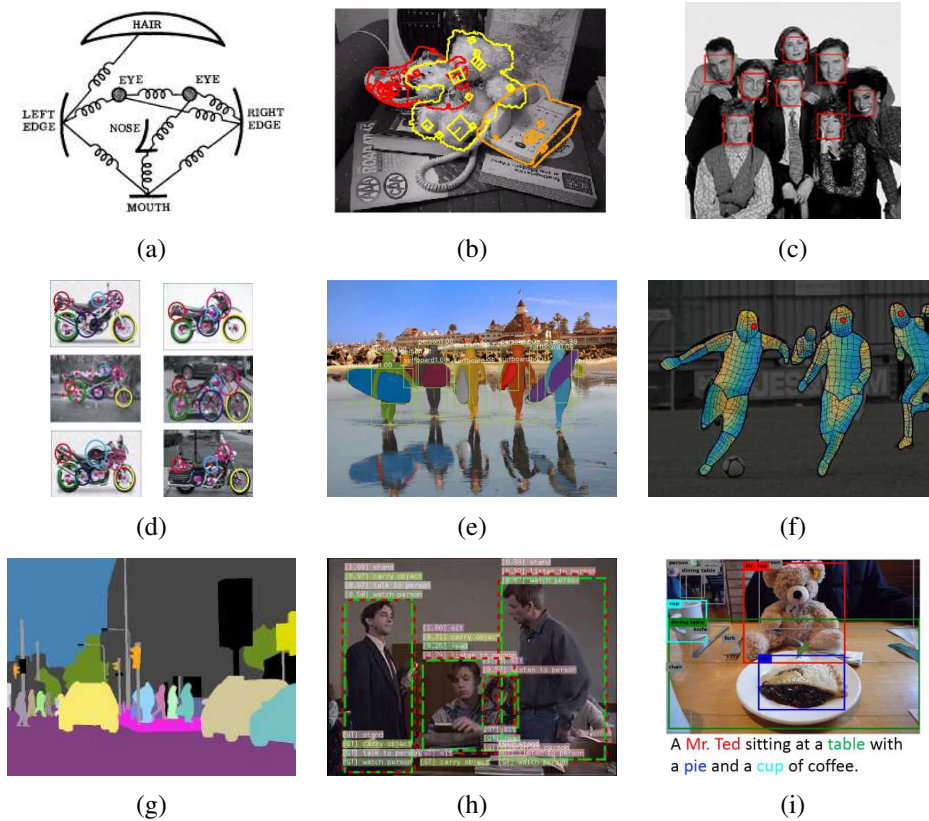


Figure 6.1 Various kinds of recognition: (a) face recognition with pictorial structures (Fischler and Elschlager 1973) © 1973 IEEE; (b) instance (known object) recognition (Lowe 1999) © 1999 IEEE; (c) real-time face detection (Viola and Jones 2004) © 2004 Springer; (d) feature-based recognition (Fergus, Perona, and Zisserman 2007) © 2007 Springer; (e) instance segmentation using Mask R-CNN (He, Gkioxari et al. 2017) © 2017 IEEE; (f) pose estimation (Güler, Neverova, and Kokkinos 2018) © 2018 IEEE; (g) panoptic segmentation (Kirillov, He et al. 2019) © 2019 IEEE; (h) video action recognition (Feichtenhofer, Fan et al. 2019); (i) image captioning (Lu, Yang et al. 2018) © 2018 IEEE.

Of all the computer vision topics covered in this book, visual recognition has undergone the largest changes and fastest development in the last decade, due in part to the availability of much larger labeled datasets as well as breakthroughs in deep learning (Figure 5.40). In the first edition of this book (Szeliski 2010), recognition was the last chapter, since it was considered a “high-level task” to be layered on top of lower-level components such as feature detection and matching. In fact, many introductory vision courses still teach recognition at the end, often covering “classic” (non-learning) vision algorithms and applications first, and then shifting to deep learning and recognition.

As I mentioned in the preface and introduction, I have now moved machine and deep learning to early in the book, since it is foundational technology widely used in other parts of computer vision. I also decided to move the recognition chapter right after deep learning, since most of the modern techniques for recognition are natural applications of deep neural networks. The majority of the old recognition chapter has been replaced with newer deep learning techniques, so you will sometimes find terse descriptions of classical recognition techniques along with pointers to the first edition and relevant surveys or seminal papers.

A good example of the classic approach is *instance recognition*, where we are trying to find exemplars of a particular manufactured object such as a stop sign or sneaker (Figure 6.1b). (An even earlier example is face recognition using relative feature locations, as shown in Figure 6.1a.) The general approach of finding distinctive features while dealing with local appearance variation (Section 7.1.2), and then checking for their co-occurrence and relative positions in an image, is still widely used for manufactured 3D object detection (Figure 6.3), 3D structure and pose recovery (Chapter 11), and location recognition (Section 11.2.3). Highly accurate and widely used feature-based approaches to instance recognition were developed in the 2000s (Figure 7.27) and, despite more recent deep learning-based alternatives, are often still the preferred method (Sattler, Zhou *et al.* 2019). We review instance recognition in Section 6.1, although some of the needed components, such as feature detection, description, and matching (Chapter 7), as well as 3D pose estimation and verification (Chapter 11), will not be introduced until later.

The more difficult problem of *category* or *class recognition* (e.g., recognizing members of highly variable categories such as cats, dogs, or motorcycles) was also initially attacked using feature-based approaches and relative locations (*part-based models*), such as the one depicted in Figure 6.1d. We begin our discussion of *image classification* (another name for whole-image category recognition) in Section 6.2 with a review of such “classic” (though now rarely used) techniques. We then show how the deep neural networks described in the previous chapter are ideally suited to these kinds of classification problems. Next, we cover visual similarity search, where instead of categorizing an image into a predefined number of

categories, we retrieve other images that are semantically similar. Finally, we focus on face recognition, which is one of the longest studied topics in computer vision.

In Section 6.3, we turn to the topic of *object detection*, where we categorize not just whole images but delineate (with bounding boxes) where various objects are located. This topic includes more specialized variants such as *face detection* and *pedestrian detection*, as well as the detection of objects in generic categories. In Section 6.4, we study *semantic segmentation*, where the task is now to delineate various objects and materials in a pixel-accurate manner, i.e., to label each pixel with an object identity and class. Variants on this include *instance segmentation*, where each separate object gets a unique label, *panoptic segmentation*, where both objects and stuff (e.g., grass, sky) get labeled, and *pose estimation*, where pixels get labeled with people's body parts and orientations. The last two sections of this chapter briefly touch on *video understanding* (Section 6.5) and *vision and language* (Section 6.6).

Before starting to describe individual recognition algorithms and variants, I should briefly mention the critical role that large-scale datasets and benchmarks have played in the rapid advancement of recognition systems. While small datasets such as Xerox 10 (Csurka, Dance *et al.* 2006) and Caltech-101 (Fei-Fei, Fergus, and Perona 2006) played an early role in evaluating object recognition systems, the PASCAL Visual Object Class (VOC) challenge (Everingham, Van Gool *et al.* 2010; Everingham, Eslami *et al.* 2015) was the first dataset large and challenging enough to significantly propel the field forward. However, PASCAL VOC only contained 20 classes. The introduction of the ImageNet dataset (Deng, Dong *et al.* 2009; Russakovsky, Deng *et al.* 2015), which had 1,000 classes and over one million labeled images, finally provided enough data to enable end-to-end learning systems to break through. The Microsoft COCO (Common Objects in Context) dataset spurred further development (Lin, Maire *et al.* 2014), especially in accurate per-object segmentation, which we study in Section 6.4. A nice review of crowdsourcing methods to construct such datasets is presented in (Kovashka, Russakovsky *et al.* 2016). We will mention additional, sometimes more specialized, datasets throughout this chapter. A listing of the most popular and active datasets and benchmarks is provided in Tables 6.1–6.4.

6.1 Instance recognition

General object recognition falls into two broad categories, namely *instance recognition* and *class recognition*. The former involves re-recognizing a known 2D or 3D rigid object, potentially being viewed from a novel viewpoint, against a cluttered background, and with partial



Figure 6.2 *Recognizing objects in a cluttered scene (Lowe 2004) © 2004 Springer. Two of the training images in the database are shown on the left. They are matched to the cluttered scene in the middle using SIFT features, shown as small squares in the right image. The affine warp of each recognized database image onto the scene is shown as a larger parallelogram in the right image.*

occlusions.¹ The latter, which is also known as *category-level* or *generic* object recognition (Ponce, Hebert *et al.* 2006), is the much more challenging problem of recognizing any instance of a particular general class, such as “cat”, “car”, or “bicycle”.

Over the years, many different algorithms have been developed for instance recognition. Mundy (2006) surveys earlier approaches, which focused on extracting lines, contours, or 3D surfaces from images and matching them to known 3D object models. Another popular approach was to acquire images from a large set of viewpoints and illuminations and to represent them using an eigenspace decomposition (Murase and Nayar 1995). More recent approaches (Lowe 2004; Lepetit and Fua 2005; Rothganger, Lazebnik *et al.* 2006; Ferrari, Tuytelaars, and Van Gool 2006b; Gordon and Lowe 2006; Obdržálek and Matas 2006; Sivic and Zisserman 2009; Zheng, Yang, and Tian 2018) tend to use viewpoint-invariant 2D features, such as those we will discuss in Section 7.1.2. After extracting informative sparse 2D features from both the new image and the images in the database, image features are matched against the object database, using one of the sparse feature matching strategies described in Section 7.1.3. Whenever a sufficient number of matches have been found, they are verified by finding a geometric transformation that aligns the two sets of features (Figure 6.2).

¹The Microsoft COCO dataset paper (Lin, Maire *et al.* 2014) introduced the newer concept of *instance segmentation*, which is the pixel-accurate delineation of different objects drawn from a set of generic classes (Section 6.4.2). This now sometimes leads to confusion, unless you look at these two terms (instance recognition vs. segmentation) carefully.

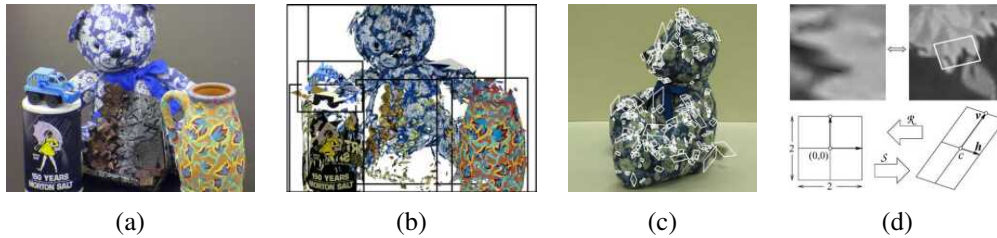


Figure 6.3 3D object recognition with affine regions (Rothganger, Lazebnik et al. 2006) © 2006 Springer: (a) sample input image; (b) five of the recognized (reprojected) objects along with their bounding boxes; (c) a few of the local affine regions; (d) local affine region (patch) reprojected into a canonical (square) frame, along with its geometric affine transformations.

Geometric alignment

To recognize one or more instances of some known objects, such as those shown in the left column of Figure 6.2, the recognition system first extracts a set of interest points in each database image and stores the associated descriptors (and original positions) in an indexing structure such as a search tree (Section 7.1.3). At recognition time, features are extracted from the new image and compared against the stored object features. Whenever a sufficient number of matching features (say, three or more) are found for a given object, the system then invokes a *match verification* stage, whose job is to determine whether the spatial arrangement of matching features is consistent with those in the database image.

Because images can be highly cluttered and similar features may belong to several objects, the original set of feature matches can have a large number of outliers. For this reason, Lowe (2004) suggests using a Hough transform (Section 7.4.2) to accumulate votes for likely geometric transformations. In his system, he uses an affine transformation between the database object and the collection of scene features, which works well for objects that are mostly planar, or where at least several corresponding features share a quasi-planar geometry.²

Another system that uses local affine frames is the one developed by Rothganger, Lazebnik et al. (2006). In their system, the affine region detector of Mikolajczyk and Schmid (2004) is used to rectify local image patches (Figure 6.3d), from which both a SIFT descriptor and a 10×10 UV color histogram are computed and used for matching and recognition. Corresponding patches in different views of the same object, along with their local affine deformations, are used to compute a 3D affine model for the object using an extension of

²When a larger number of features is available, a full fundamental matrix can be used (Brown and Lowe 2002; Gordon and Lowe 2006). When image stitching is being performed (Brown and Lowe 2007), the motion models discussed in Section 8.2.1 can be used instead.

the factorization algorithm of Section 11.4.1, which can then be upgraded to a Euclidean reconstruction (Tomasi and Kanade 1992). At recognition time, local Euclidean neighborhood constraints are used to filter potential matches, in a manner analogous to the affine geometric constraints used by Lowe (2004) and Obdržálek and Matas (2006). Figure 6.3 shows the results of recognizing five objects in a cluttered scene using this approach.

While feature-based approaches are normally used to detect and localize known objects in scenes, it is also possible to get pixel-level segmentations of the scene based on such matches. Ferrari, Tuytelaars, and Van Gool (2006b) describe such a system for simultaneously recognizing objects and segmenting scenes, while Kannala, Rahtu *et al.* (2008) extend this approach to non-rigid deformations. Section 6.4 re-visits this topic of joint recognition and segmentation in the context of generic class (category) recognition.

While instance recognition in the early to mid-2000s focused on the problem of locating a known 3D object in an image, as shown in Figures 6.2–6.3, attention shifted to the more challenging problem of *instance retrieval* (also known as *content-based image retrieval*), in which the number of images being searched can be very large. Section 7.1.4 reviews such techniques, a snapshot of which can be seen in Figure 7.27 and the survey by Zheng, Yang, and Tian (2018). This topic is also related to visual similarity search (Section 6.2.3 and 3D pose estimation (Section 11.2).

6.2 Image classification

While instance recognition techniques are relatively mature and are used in commercial applications such as traffic sign recognition (Stallkamp, Schlipsing *et al.* 2012), generic category (class) recognition is still a rapidly evolving research area. Consider for example the set of photographs in Figure 6.4a, which shows objects taken from 10 different visual categories. (I’ll leave it up to you to name each of the categories.) How would you go about writing a program to categorize each of these images into the appropriate class, especially if you were also given the choice “none of the above”?

As you can tell from this example, visual category recognition is an extremely challenging problem. However, the progress in the field has been quite dramatic, if judged by how much better today’s algorithms are compared to those of a decade ago.

In this section, we review the main classes of algorithms used for whole-image classification. We begin with classic feature-based approaches that rely on handcrafted features and their statistics, optionally using machine learning to do the final classification (Figure 5.2b). Since such techniques are no longer widely used, we present a fairly terse description of the most important techniques. More details can be found in the first edition of this book

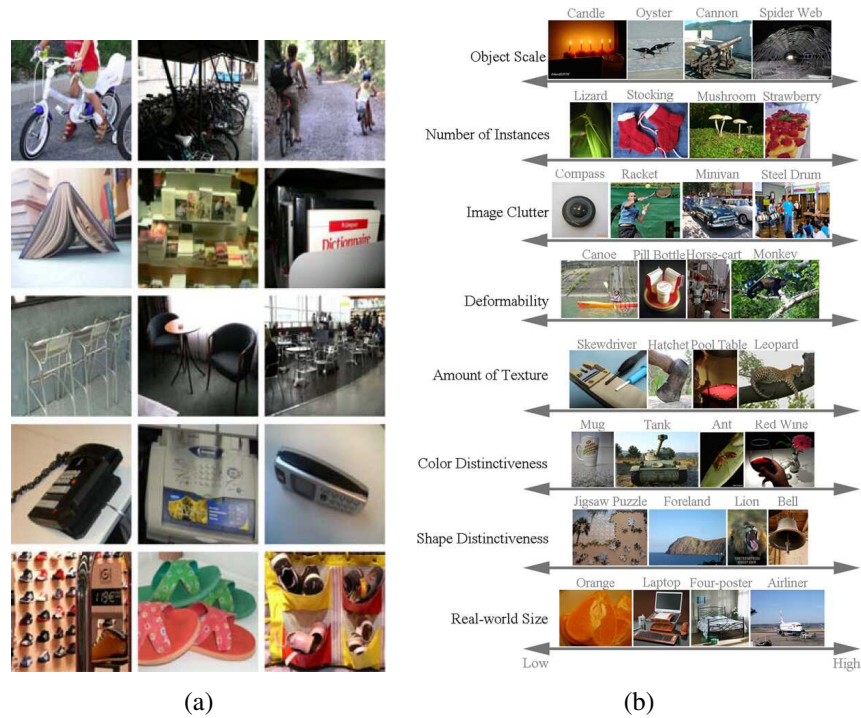


Figure 6.4 Challenges in image recognition: (a) sample images from the Xerox 10 class dataset (Csurka, Dance et al. 2006) © 2007 Springer; (b) axes of difficulty and variation from the ImageNet dataset (Russakovsky, Deng et al. 2015) © 2015 Springer.

(Szeliski 2010, Chapter 14) and in the cited journal papers and surveys. Next, we describe modern image classification systems, which are based on the deep neural networks we introduced in the previous chapter. We then describe visual similarity search, where the task is to find visually and semantically similar images, rather than classification into a fixed set of categories. Finally, we look at face recognition, since this topic has its own long history and set of techniques.

6.2.1 Feature-based methods

In this section, we review “classic” feature-based approaches to category recognition (image classification). While, historically, *part-based* representations and recognition algorithms (Section 6.2.1) were the preferred approach (Fischler and Elschlager 1973; Felzenszwalb and Huttenlocher 2005; Fergus, Perona, and Zisserman 2007), we begin by describing sim-

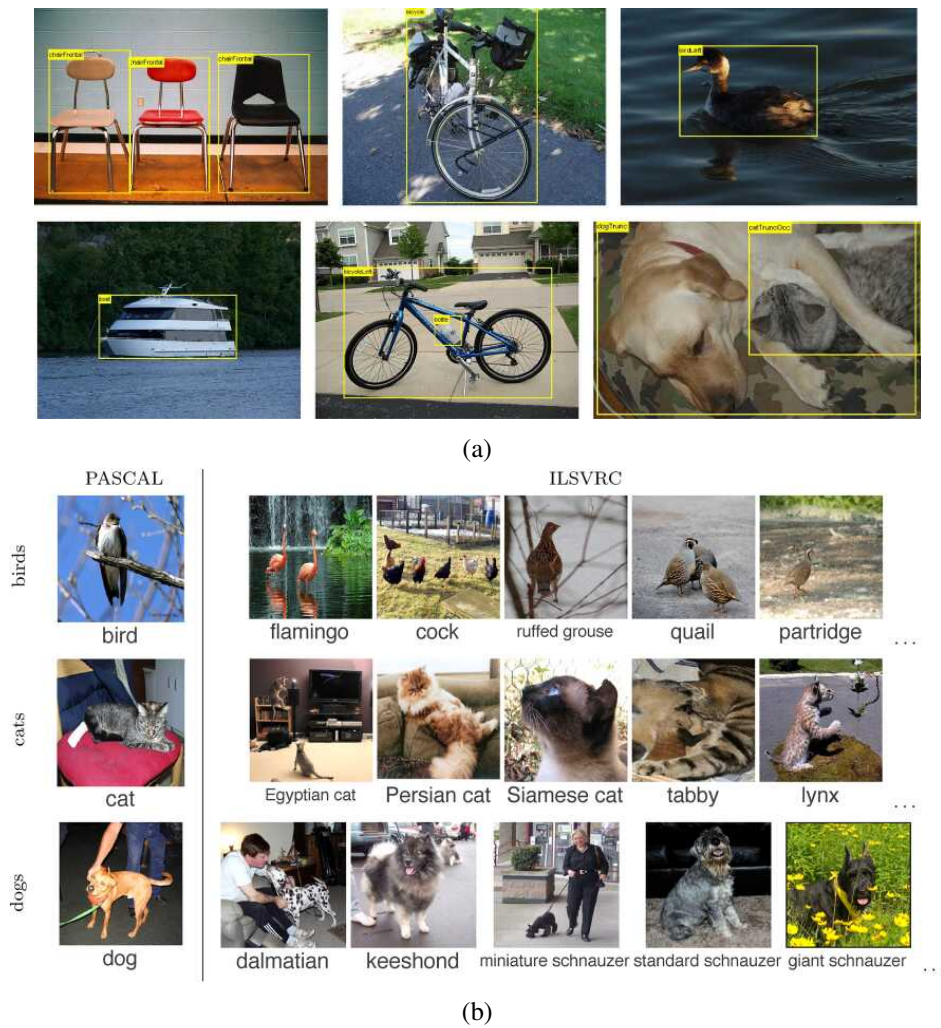


Figure 6.5 Sample images from two widely used image classification datasets: (a) *Pascal Visual Object Categories (VOC)* (Everingham, Eslami et al. 2015) © 2015 Springer; (b) *ImageNet* (Russakovsky, Deng et al. 2015) © 2015 Springer.

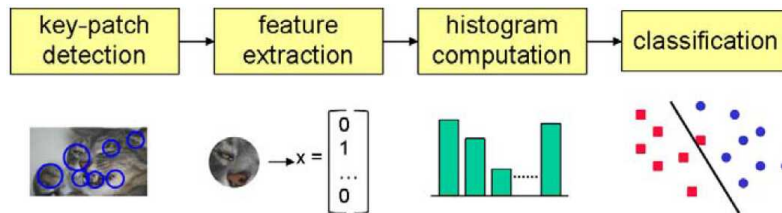


Figure 6.6 A typical processing pipeline for a bag-of-words category recognition system (Csurka, Dance *et al.* 2006) © 2007 Springer. Features are first extracted at keypoints and then quantized to get a distribution (histogram) over the learned visual words (feature cluster centers). The feature distribution histogram is used to learn a decision surface using a classification algorithm, such as a support vector machine.

pler *bag-of-features* approaches that represent objects and images as unordered collections of feature descriptors. We then review more complex systems constructed with part-based models, and then look at how context and scene understanding, as well as machine learning, can improve overall recognition results. Additional details on the techniques presented in this section can be found in older survey articles, paper collections, and courses (Pinz 2005; Ponce, Hebert *et al.* 2006; Dickinson, Leonardis *et al.* 2007; Fei-Fei, Fergus, and Torralba 2009), as well as two review articles on the PASCAL and ImageNet recognition challenges (Everingham, Van Gool *et al.* 2010; Everingham, Eslami *et al.* 2015; Russakovsky, Deng *et al.* 2015) and the first edition of this book (Szeliski 2010, Chapter 14).

Bag of words

One of the simplest algorithms for category recognition is the *bag of words* (also known as *bag of features* or *bag of keypoints*) approach (Csurka, Dance *et al.* 2004; Lazebnik, Schmid, and Ponce 2006; Csurka, Dance *et al.* 2006; Zhang, Marszalek *et al.* 2007). As shown in Figure 6.6, this algorithm simply computes the distribution (histogram) of visual words found in the query image and compares this distribution to those found in the training images. We will give more details of this approach in Section 7.1.4. The biggest difference from instance recognition is the absence of a geometric verification stage (Section 6.1), since individual instances of generic visual categories, such as those shown in Figure 6.4a, have relatively little spatial coherence to their features (but see the work by Lazebnik, Schmid, and Ponce (2006)).

Csurka, Dance *et al.* (2004) were the first to use the term *bag of keypoints* to describe such approaches and among the first to demonstrate the utility of frequency-based techniques for

category recognition. Their original system used affine covariant regions and SIFT descriptors, k-means visual vocabulary construction, and both a naïve Bayesian classifier and support vector machines for classification. (The latter was found to perform better.) Their newer system (Csurka, Dance *et al.* 2006) uses regular (non-affine) SIFT patches and boosting instead of SVMs and incorporates a small amount of geometric consistency information.

Zhang, Marszalek *et al.* (2007) perform a more detailed study of such bag of features systems. They compare a number of feature detectors (Harris–Laplace (Mikolajczyk and Schmid 2004) and Laplacian (Lindeberg 1998b)), descriptors (SIFT, RIFT, and SPIN (Lazebnik, Schmid, and Ponce 2005)), and SVM kernel functions.

Instead of quantizing feature vectors to visual words, Grauman and Darrell (2007b) develop a technique for directly computing an approximate distance between two variably sized collections of feature vectors. Their approach is to bin the feature vectors into a multi-resolution pyramid defined in feature space and count the number of features that land in corresponding bins B_{il} and B'_{il} . The distance between the two sets of feature vectors (which can be thought of as points in a high-dimensional space) is computed using histogram intersection between corresponding bins, while discounting matches already found at finer levels and weighting finer matches more heavily. In follow-on work, Grauman and Darrell (2007a) show how an explicit construction of the pyramid can be avoided using hashing techniques.

Inspired by this work, Lazebnik, Schmid, and Ponce (2006) show how a similar idea can be employed to augment bags of keypoints with loose notions of 2D spatial location analogous to the pooling performed by SIFT (Lowe 2004) and “gist” (Torralba, Murphy *et al.* 2003). In their work, they extract affine region descriptors (Lazebnik, Schmid, and Ponce 2005) and quantize them into visual words. (Based on previous results by Fei-Fei and Perona (2005), the feature descriptors are extracted densely (on a regular grid) over the image, which can be helpful in describing textureless regions such as the sky.) They then form a spatial pyramid of bins containing word counts (histograms) and use a similar pyramid match kernel to combine histogram intersection counts in a hierarchical fashion.

The debate about whether to use quantized feature descriptors or continuous descriptors and also whether to use sparse or dense features went on for many years. Boiman, Shechtman, and Irani (2008) show that if query images are compared to *all* the features representing a given class, rather than just each class image individually, nearest-neighbor matching followed by a naïve Bayes classifier outperforms quantized visual words. Instead of using generic feature detectors and descriptors, some authors have been investigating *learning* class-specific features (Ferencz, Learned-Miller, and Malik 2008), often using randomized forests (Philbin, Chum *et al.* 2007; Moosmann, Nowak, and Jurie 2008; Shotton, Johnson, and Cipolla 2008) or combining the feature generation and image classification stages (Yang,

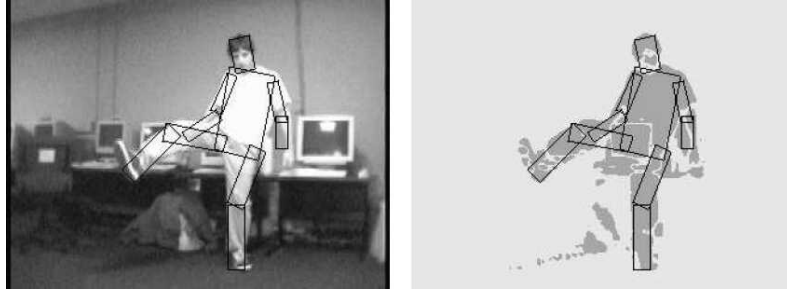


Figure 6.7 Using pictorial structures to locate and track a person (Felzenszwalb and Huttenlocher 2005) © 2005 Springer. The structure consists of articulated rectangular body parts (torso, head, and limbs) connected in a tree topology that encodes relative part positions and orientations. To fit a pictorial structure model, a binary silhouette image is first computed using background subtraction.

Jin *et al.* 2008). Others, such as Serre, Wolf, and Poggio (2005) and Mutch and Lowe (2008) use hierarchies of dense feature transforms inspired by biological (visual cortical) processing combined with SVMs for final classification.

Part-based models

Recognizing an object by finding its constituent parts and measuring their geometric relationships is one of the oldest approaches to object recognition (Fischler and Elschlager 1973; Kanade 1977; Yuille 1991). Part-based approaches were often used for face recognition (Moghaddam and Pentland 1997; Heisele, Ho *et al.* 2003; Heisele, Serre, and Poggio 2007) and continue being used for pedestrian detection (Figure 6.24) (Felzenszwalb, McAllester, and Ramanan 2008) and pose estimation (Güler, Neverova, and Kokkinos 2018).

In this overview, we discuss some of the central issues in part-based recognition, namely, the representation of geometric relationships, the representation of individual parts, and algorithms for learning such descriptions and recognizing them at run time. More details on part-based models for recognition can be found in the course notes by Fergus (2009).

The earliest approaches to representing geometric relationships were dubbed *pictorial structures* by Fischler and Elschlager (1973) and consisted of spring-like connections between different feature locations (Figure 6.1a). To fit a pictorial structure to an image, an energy function of the form

$$E = \sum_i V_i(\mathbf{l}_i) + \sum_{ij \in E} V_{ij}(\mathbf{l}_i, \mathbf{l}_j) \quad (6.1)$$

is minimized over all potential part locations or poses $\{\mathbf{l}_i\}$ and pairs of parts (i, j) for which an edge (geometric relationship) exists in E . Note how this energy is closely related to that used with Markov random fields (4.35–4.38), which can be used to embed pictorial structures in a probabilistic framework that makes parameter learning easier (Felzenszwalb and Huttenlocher 2005).

Part-based models can have different topologies for the geometric connections between the parts (Carneiro and Lowe 2006). For example, Felzenszwalb and Huttenlocher (2005) restrict the connections to a tree, which makes learning and inference more tractable. A tree topology enables the use of a recursive Viterbi (dynamic programming) algorithm (Pearl 1988; Bishop 2006), in which leaf nodes are first optimized as a function of their parents, and the resulting values are then plugged in and eliminated from the energy function. To further increase the efficiency of the inference algorithm, Felzenszwalb and Huttenlocher (2005) restrict the pairwise energy functions $V_{ij}(\mathbf{l}_i, \mathbf{l}_j)$ to be Mahalanobis distances on functions of location variables and then use fast distance transform algorithms to minimize each pairwise interaction in time that is closer to linear in N .

Figure 6.7 shows the results of using their pictorial structures algorithm to fit an articulated body model to a binary image obtained by background segmentation. In this application of pictorial structures, parts are parameterized by the locations, sizes, and orientations of their approximating rectangles. Unary matching potentials $V_i(\mathbf{l}_i)$ are determined by counting the percentage of foreground and background pixels inside and just outside the tilted rectangle representing each part.

A large number of different graphical models have been proposed for part-based recognition. Carneiro and Lowe (2006) discuss a number of these models and propose one of their own, which they call a *sparse flexible model*; it involves ordering the parts and having each part's location depend on at most k of its ancestor locations.

The simplest models are bags of words, where there are no geometric relationships between different parts or features. While such models can be very efficient, they have a very limited capacity to express the spatial arrangement of parts. Trees and stars (a special case of trees where all leaf nodes are directly connected to a common root) are the most efficient in terms of inference and hence also learning (Felzenszwalb and Huttenlocher 2005; Fergus, Perona, and Zisserman 2005; Felzenszwalb, McAllester, and Ramanan 2008). Directed acyclic graphs come next in terms of complexity and can still support efficient inference, although at the cost of imposing a causal structure on the part model (Bouchard and Triggs 2005; Carneiro and Lowe 2006). k -fans, in which a clique of size k forms the root of a star-shaped model have inference complexity $O(N^{k+1})$, although with distance transforms and Gaussian priors, this can be lowered to $O(N^k)$ (Crandall, Felzenszwalb, and Huttenlocher

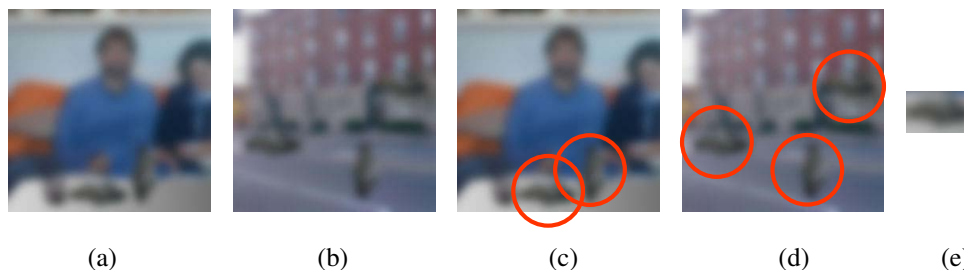


Figure 6.8 *The importance of context (images courtesy of Antonio Torralba). Can you name all of the objects in images (a–b), especially those that are circled in (c–d). Look carefully at the circled objects. Did you notice that they all have the same shape (after being rotated), as shown in column (e)?*

2005; Crandall and Huttenlocher 2006). Finally, fully connected *constellation* models are the most general, but the assignment of features to parts becomes intractable for moderate numbers of parts P , since the complexity of such an assignment is $O(N^P)$ (Fergus, Perona, and Zisserman 2007).

The original constellation model was developed by Burl, Weber, and Perona (1998) and consists of a number of parts whose relative positions are encoded by their mean locations and a full covariance matrix, which is used to denote not only positional uncertainty but also potential correlations between different parts. Weber, Welling, and Perona (2000) extended this technique to a weakly supervised setting, where both the appearance of each part and its locations are automatically learned given whole image labels. Fergus, Perona, and Zisserman (2007) further extend this approach to simultaneous learning of appearance and shape models from scale-invariant keypoint detections.

The part-based approach to recognition has also been extended to learning new categories from small numbers of examples, building on recognition components developed for other classes (Fei-Fei, Fergus, and Perona 2006). More complex hierarchical part-based models can be developed using the concept of grammars (Bouchard and Triggs 2005; Zhu and Mumford 2006). A simpler way to use parts is to have keypoints that are recognized as being part of a class vote for the estimated part locations (Leibe, Leonardis, and Schiele 2008). Parts can also be a useful component of fine-grained category recognition systems, as shown in Figure 6.9.

Context and scene understanding

Thus far, we have mostly considered the task of recognizing and localizing objects in isolation from that of understanding the scene (context) in which the object occur. This is a big

limitation, as context plays a very important role in human object recognition (Oliva and Torralba 2007). Context can greatly improve the performance of object recognition algorithms (Divvala, Hoiem *et al.* 2009), as well as providing useful semantic clues for general scene understanding (Torralba 2008).

Consider the two photographs in Figure 6.8a–b. Can you name all of the objects, especially those circled in images (c–d)? Now have a closer look at the circled objects. Do you see any similarity in their shapes? In fact, if you rotate them by 90°, they are all the same as the “blob” shown in Figure 6.8e. So much for our ability to recognize object by their shape!

Even though we have not addressed context explicitly earlier in this chapter, we have already seen several instances of this general idea being used. A simple way to incorporate spatial information into a recognition algorithm is to compute feature statistics over different regions, as in the spatial pyramid system of Lazebnik, Schmid, and Ponce (2006). Part-based models (Figure 6.7) use a kind of local context, where various parts need to be arranged in a proper geometric relationship to constitute an object.

The biggest difference between part-based and context models is that the latter combine objects into scenes and the number of constituent objects from each class is not known in advance. In fact, it is possible to combine part-based and context models into the same recognition architecture (Murphy, Torralba, and Freeman 2003; Sudderth, Torralba *et al.* 2008; Crandall and Huttenlocher 2007).

Consider an image database consisting of street and office scenes. If we have enough training images with labeled regions, such as buildings, cars, and roads, or monitors, keyboards, and mice, we can develop a geometric model for describing their relative positions. Sudderth, Torralba *et al.* (2008) develop such a model, which can be thought of as a two-level constellation model. At the top level, the distributions of objects relative to each other (say, buildings with respect to cars) is modeled as a Gaussian. At the bottom level, the distribution of parts (affine covariant features) with respect to the object center is modeled using a mixture of Gaussians. However, since the number of objects in the scene and parts in each object are unknown, a *latent Dirichlet process* (LDP) is used to model object and part creation in a generative framework. The distributions for all of the objects and parts are learned from a large labeled database and then later used during inference (recognition) to label the elements of a scene.

Another example of context is in simultaneous segmentation and recognition (Section 6.4 and Figure 6.33), where the arrangements of various objects in a scene are used as part of the labeling process. Torralba, Murphy, and Freeman (2004) describe a conditional random field where the estimated locations of building and roads influence the detection of cars, and where boosting is used to learn the structure of the CRF. Rabinovich, Vedaldi *et al.* (2007)

use context to improve the results of CRF segmentation by noting that certain adjacencies (relationships) are more likely than others, e.g., a person is more likely to be on a horse than on a dog. Galleguillos and Belongie (2010) review various approaches proposed for adding context to object categorization, while Yao and Fei-Fei (2012) study human-object interactions. (For a more recent take on this problem, see Gkioxari, Girshick *et al.* (2018).)

Context also plays an important role in 3D inference from single images (Figure 6.41), using computer vision techniques for labeling pixels as belonging to the ground, vertical surfaces, or sky (Hoiem, Efros, and Hebert 2005a). This line of work has been extended to a more holistic approach that simultaneously reasons about object identity, location, surface orientations, occlusions, and camera viewing parameters (Hoiem, Efros, and Hebert 2008).

A number of approaches use the *gist* of a scene (Torralba 2003; Torralba, Murphy *et al.* 2003) to determine where instances of particular objects are likely to occur. For example, Murphy, Torralba, and Freeman (2003) train a regressor to predict the vertical locations of objects such as pedestrians, cars, and buildings (or screens and keyboards for indoor office scenes) based on the gist of an image. These location distributions are then used with classic object detectors to improve the performance of the detectors. Gists can also be used to directly match complete images, as we saw in the scene completion work of Hays and Efros (2007).

Finally, some of the work in scene understanding exploits the existence of large numbers of labeled (or even unlabeled) images to perform matching directly against whole images, where the images themselves implicitly encode the expected relationships between objects (Russell, Torralba *et al.* 2007; Malisiewicz and Efros 2008; Galleguillos and Belongie 2010). This, of course, is one of the central benefits of using deep neural networks, which we discuss in the next section.

6.2.2 Deep networks

As we saw in Section 5.4.3, deep networks started outperforming “shallow” learning-based approaches on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) with the introduction of the “AlexNet” SuperVision system of Krizhevsky, Sutskever, and Hinton (2012). Since that time, recognition accuracy has continued to improve dramatically (Figure 5.40) driven to a large degree by deeper networks and better training algorithms. More recently, more efficient networks have become the focus of research (Figure 5.45) as well as larger (unlabeled) training datasets (Section 5.4.7). There are now open-source frameworks such as Classy Vision³ for training and fine tuning your own image and video classification models. Users can also upload custom images on the web to the Computer Vision Explorer⁴

³<https://classyvision.ai>

⁴<https://vision-explorer.allenai.org>

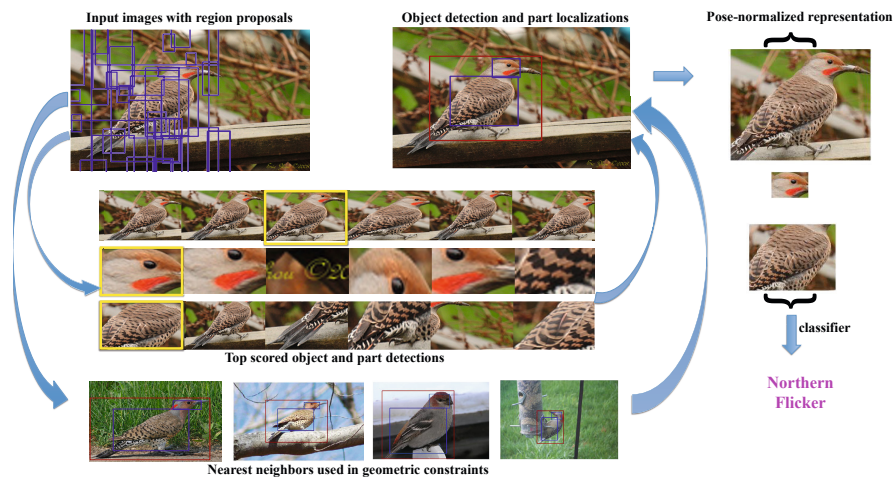


Figure 6.9 *Fine-grained category recognition using parts (Zhang, Donahue et al. 2014) © 2014 Springer. Deep neural network object and part detectors are trained and their outputs are combined using geometric constraints. A classifier trained on features from the extracted parts is used for the final categorization.*

to see how well many popular computer vision models perform on their own images.

In addition to recognizing commonly occurring categories such as those found in the ImageNet and COCO datasets, researchers have studied the problem of *fine-grained* category recognition (Duan, Parikh et al. 2012; Zhang, Donahue et al. 2014; Krause, Jin et al. 2015), where the differences between sub-categories can be subtle and the number of exemplars is quite low (Figure 6.9). Examples of categories with fine-grained sub-classes include flowers (Nilsback and Zisserman 2006), cats and dogs (Parkhi, Vedaldi et al. 2012), birds (Wah, Branson et al. 2011; Van Horn, Branson et al. 2015), and cars (Yang, Luo et al. 2015). A recent example of fine-grained categorization is the iNaturalist system (Van Horn, Mac Aodha et al. 2018),⁵ which allows both specialists and citizen scientists to photograph and label biological species, using a fine-grained category recognition system to label new images (Figure 6.10a).

Fine-grained categorization is often attacked using *attributes* of images and classes (Lampert, Nickisch, and Harmeling 2009; Parikh and Grauman 2011; Lampert, Nickisch, and Harmeling 2014), as shown in Figure 6.10b. Extracting attributes can enable *zero-shot learning* (Xian, Lampert et al. 2019), where previously unseen categories can be described using combinations of such attributes. However, some caution must be used in order not to

⁵<https://www.inaturalist.org>

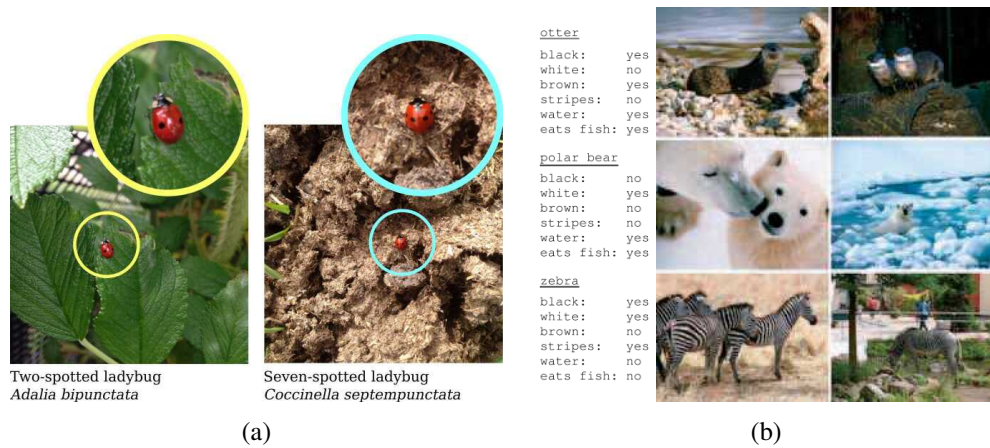


Figure 6.10 Fine-grained category recognition. (a) The iNaturalist website and app allows citizen scientists to collect and classify images on their phones (Van Horn, Mac Aodha *et al.* 2018) © 2018 IEEE. (b) Attributes can be used for fine-grained categorization and zero-shot learning (Lampert, Nickisch, and Harmeling 2014) © 2014 Springer. These images are part of the Animals with Attributes dataset.

learn spurious correlations between different attributes (Jayaraman, Sha, and Grauman 2014) or between objects and their common contexts (Singh, Mahajan *et al.* 2020). Fine-grained recognition can also be tackled using metric learning (Wu, Manmatha *et al.* 2017) or nearest-neighbor visual similarity search (Touvron, Sablayrolles *et al.* 2020), which we discuss next.

6.2.3 Application: Visual similarity search

Automatically classifying images into categories and tagging them with attributes using computer vision algorithms makes it easier to find them in catalogues and on the web. This is commonly used in *image search* or *image retrieval* engines, which find likely images based on keywords, just as regular web search engines find relevant documents and pages.

Sometimes, however, it's easier to find the information you need from an image, i.e., using *visual search*. Examples of this include fine-grained categorization, which we have just seen, as well as instance retrieval, i.e., finding the exact same object (Section 6.1) or location (Section 11.2.3). Another variant is finding visually similar images (often called *visual similarity search* or *reverse image search*), which is useful when the search intent cannot be succinctly captured in words.⁶

⁶Some authors use the term image retrieval to denote visual similarity search, (e.g., Jégou, Perronnin *et al.* 2012;



Figure 6.11 The GrokNet product recognition service is used for product tagging, visual search, and recommendations © Bell, Liu *et al.* (2020): (a) recognizing all the products in a photo; (b) automatically sourcing data for metric learning using weakly supervised data augmentation.

The topic of searching by visual similarity has a long history and goes by a variety of names, including query by image content (QBIC) (Flickner, Sawhney *et al.* 1995) and content-based image retrieval (CBIR) (Smeulders, Worring *et al.* 2000; Lew, Sebe *et al.* 2006; Vasconcelos 2007; Datta, Joshi *et al.* 2008). Early publications in these fields were based primarily on simple whole-image similarity metrics, such as color and texture (Swain and Ballard 1991; Jacobs, Finkelstein, and Salesin 1995; Manjunathi and Ma 1996).

Later architectures, such as that by Fergus, Perona, and Zisserman (2004), use a feature-based learning and recognition algorithm to re-rank the outputs from a traditional keyword-based image search engine. In follow-on work, Fergus, Fei-Fei *et al.* (2005) cluster the results returned by image search using an extension of probabilistic latent semantic analysis (PLSA) (Hofmann 1999) and then select the clusters associated with the highest ranked results as the representative images for that category. Other approaches rely on carefully annotated image databases such as LabelMe (Russell, Torralba *et al.* 2008). For example, Malisiewicz and Efros (2008) describe a system that, given a query image, can find similar LabelMe images, whereas Liu, Yuen, and Torralba (2009) combine feature-based correspondence algorithms with the labeled database to perform simultaneous recognition and segmentation.

Newer approaches to visual similarity search use whole-image descriptors such as Fisher kernels and the Vector of Locally Aggregated Descriptors (VLAD) (Jégou, Perronnin *et al.* 2012) or pooled CNN activations (Babenko and Lempitsky 2015a; Tolias, Sivic, and Jégou 2016; Cao, Araujo, and Sim 2020; Ng, Balntas *et al.* 2020; Tolias, Jenicek, and Chum 2020) combined with metric learning (Bell and Bala 2015; Song, Xiang *et al.* 2016; Gordo, Almazán *et al.* 2017; Wu, Manmatha *et al.* 2017; Berman, Jégou *et al.* 2019) to represent each image with a compact descriptor that can be used to measure similarity in large databases (Johnson, Douze, and Jégou 2021). It is also possible to combine several techniques, such as deep networks with VLAD (Arandjelovic, Gronat *et al.* 2016), generalized mean (GeM)

(Radenović, Tolias, and Chum 2019).

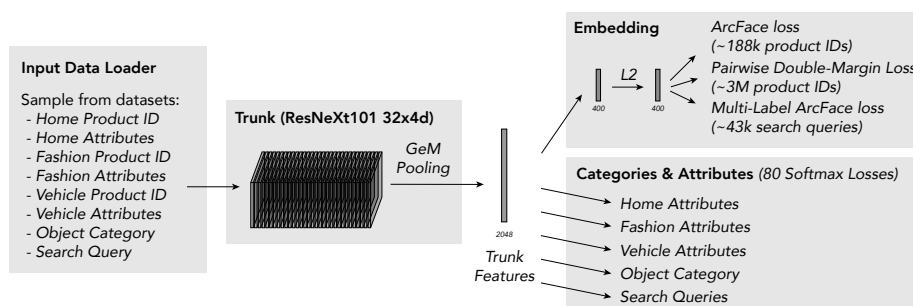


Figure 6.12 The GrokNet training architecture uses seven datasets, a common DNN trunk, two branches, and 83 loss functions (80 categorical losses + 3 embedding losses) © Bell, Liu *et al.* (2020).

pooling (Radenović, Tolias, and Chum 2019), or dynamic mean (DAME) pooling (Yang, Kien Nguyen *et al.* 2019) into complete systems that are end-to-end tunable. Gordo, Almazán *et al.* (2017) provide a comprehensive review and experimental comparison of many of these techniques, which we also discuss in Section 7.1.4 on large-scale matching and retrieval. Some of the latest techniques for image retrieval use combinations of local and global descriptors to obtain state-of-the-art performance on the landmark recognition tasks (Cao, Araujo, and Sim 2020; Ng, Balntas *et al.* 2020; Tolias, Jeníček, and Chum 2020). The ECCV 2020 Workshop on Instance-Level Recognition⁷ has pointers to some of the latest work in this area, while the upcoming NeurIPS’21 Image Similarity Challenge⁸ has new datasets for detecting content manipulation.

A recent example of a commercial system that uses visual similarity search, in addition to category recognition, is the GrokNet product recognition service described by Bell, Liu *et al.* (2020). GrokNet takes as input user images and shopping queries and returns indexed items similar to the ones in the query image (Figure 6.11a). The reason for needing a similarity search component is that the world contains too many “long-tail” items such as “a fur sink, an electric dog polisher, or a gasoline powered turtleneck sweater”,⁹ to make full categorization practical.

At training time, GrokNet takes both weakly labeled images, with category and/or attribute labels, and unlabeled images, where features in objects are detected and then used for metric learning, using a modification of ArcFace loss (Deng, Guo *et al.* 2019) and a novel

⁷<https://ilr-workshop.github.io/ECCVW2020>

⁸<https://www.drivendata.org/competitions/79/>

⁹<https://www.google.com/search?q=gasoline+powered+turtleneck+sweater>



Figure 6.13 Humans can recognize low-resolution faces of familiar people (Sinha, Balas *et al.* 2006) © 2006 IEEE.

pairwise margin loss (Figure 6.11b). The overall system takes in large collections of unlabeled and weakly labeled images and trains a ResNeXt101 trunk using a combination of category and attribute softmax losses and three different metric losses on the embeddings (Figure 6.12). GrokNet is just one example of a large number of commercial visual product search systems that have recently been developed. Others include systems from Amazon (Wu, Manmatha *et al.* 2017), Pinterest (Zhai, Wu *et al.* 2019), and Facebook (Tang, Borisjuk *et al.* 2019). In addition to helping people find items they may wish to purchase, large-scale similarity search can also speed the search for harmful content on the web, as exemplified in Facebook’s SimSearchNet.¹⁰

6.2.4 Face recognition

Among the various recognition tasks that computers are asked to perform, face recognition is the one where they have arguably had the most success.¹¹ While even people cannot readily distinguish between similar people with whom they are not familiar (O’Toole, Jiang *et al.* 2006; O’Toole, Phillips *et al.* 2009), computers’ ability to distinguish among a small number of family members and friends has found its way into consumer-level photo applications. Face recognition can be used in a variety of additional applications, including human–computer interaction (HCI), identity verification (Kirovski, Jojic, and Jancke 2004), desktop login, parental controls, and patient monitoring (Zhao, Chellappa *et al.* 2003), but it also has the potential for misuse (Chokshi 2019; Ovide 2020).

Face recognizers work best when they are given images of faces under a wide variety of

¹⁰<https://ai.facebook.com/blog/using-ai-to-detect-covid-19-misinformation-and-exploitative-content>

¹¹Instance recognition, i.e., the re-recognition of known objects such as locations or planar objects, is the other most successful application of general image recognition. In the general domain of *biometrics*, i.e., identity recognition, specialized images such as irises and fingerprints perform even better (Jain, Bolle, and Pankanti 1999; Daugman 2004).

Name/URL	Contents/Reference
CMU Multi-PIE database http://www.cs.cmu.edu/afs/cs/project/PIE/MultiPie	337 people's faces in various poses Gross, Matthews <i>et al.</i> (2010)
Faces in the Wild http://vis-www.cs.umass.edu/lfw	5,749 internet celebrities Huang, Ramesh <i>et al.</i> (2007)
YouTube Faces (YTF) https://www.cs.tau.ac.il/~wolf/ytfaces	1,595 people in 3,425 YouTube videos Wolf, Hassner, and Maoz (2011)
MegaFace https://megaface.cs.washington.edu	1M internet faces Nech and Kemelmacher-Shlizerman (2017)
IARPA Janus Benchmark (IJB) https://www.nist.gov/programs-projects/face-challenges	31,334 faces of 3,531 people in videos Maze, Adams <i>et al.</i> (2018)
WIDER FACE http://shuoyang1213.me/WIDERFACE	32,203 images for face <i>detection</i> Yang, Luo <i>et al.</i> (2016)

Table 6.1 *Face recognition and detection datasets, adapted from Maze, Adams et al. (2018).*

pose, illumination, and expression (PIE) conditions (Phillips, Moon *et al.* 2000; Sim, Baker, and Bsat 2003; Gross, Shi, and Cohn 2005; Huang, Ramesh *et al.* 2007; Phillips, Scruggs *et al.* 2010). More recent widely used datasets include labeled Faces in the Wild (LFW) (Huang, Ramesh *et al.* 2007; Learned-Miller, Huang *et al.* 2016), YouTube Faces (YTF) (Wolf, Hassner, and Maoz 2011), MegaFace (Kemelmacher-Shlizerman, Seitz *et al.* 2016; Nech and Kemelmacher-Shlizerman 2017), and the IARPA Janus Benchmark (IJB) (Klare, Klein *et al.* 2015; Maze, Adams *et al.* 2018), as tabulated in Table 6.1. (See Masi, Wu *et al.* (2018) for additional datasets used for training.)

Some of the earliest approaches to face recognition involved finding the locations of distinctive image features, such as the eyes, nose, and mouth, and measuring the distances between these feature locations (Fischler and Elschlager 1973; Kanade 1977; Yuille 1991). Other approaches relied on comparing gray-level images projected onto lower dimensional subspaces called *eigenfaces* (Section 5.2.3) and jointly modeling shape and appearance variations (while discounting pose variations) using *active appearance models* (Section 6.2.4). Descriptions of “classic” (pre-DNN) face recognition systems can be found in a number of surveys and books on this topic (Chellappa, Wilson, and Sirohey 1995; Zhao, Chellappa *et al.* 2003; Li and Jain 2005) as well as the Face Recognition website.¹² The survey on face recognition by humans by Sinha, Balas *et al.* (2006) is also well worth reading; it includes a number of surprising results, such as humans’ ability to recognize low-resolution images of familiar

¹²<https://www.face-rec.org>

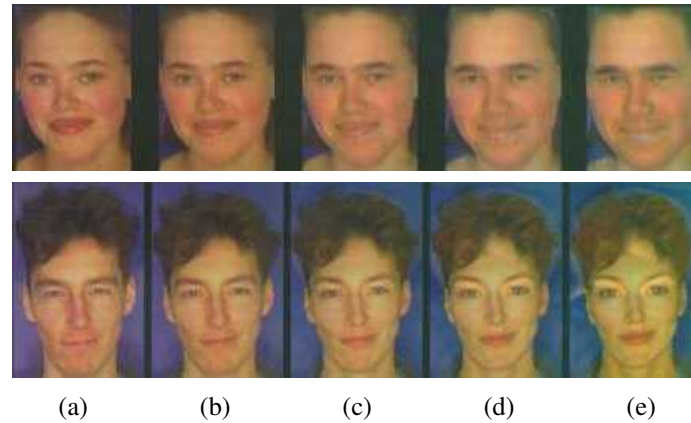


Figure 6.14 *Manipulating facial appearance through shape and color (Rowland and Perrett 1995) © 1995 IEEE. By adding or subtracting gender-specific shape and color characteristics to an input image (b), different amounts of gender variation can be induced. The amounts added (from the mean) are: (a) +50% (gender enhancement), (b) 0% (original image), (c) –50% (near “androgyny”), (d) –100% (gender switched), and (e) –150% (opposite gender attributes enhanced).*

faces (Figure 6.13) and the importance of eyebrows in recognition. Researchers have also studied the automatic recognition of facial expressions. See Chang, Hu *et al.* (2006), Shan, Gong, and McOwan (2009), and Li and Deng (2020) for some representative papers.

Active appearance and 3D shape models

The need to use modular or view-based eigenspaces for face recognition, which we discussed in Section 5.2.3, is symptomatic of a more general observation, i.e., that facial appearance and identifiability depend as much on *shape* as they do on color or texture (which is what eigenfaces capture). Furthermore, when dealing with 3D head rotations, the *pose* of a person’s head should be discounted when performing recognition.

In fact, the earliest face recognition systems, such as those by Fischler and Elschlager (1973), Kanade (1977), and Yuille (1991), found distinctive feature points on facial images and performed recognition on the basis of their relative positions or distances. Later techniques such as *local feature analysis* (Penev and Atick 1996) and *elastic bunch graph matching* (Wiskott, Fellous *et al.* 1997) combined local filter responses (jets) at distinctive feature locations together with shape models to perform recognition.

A visually compelling example of why both shape and texture are important is the work

of Rowland and Perrett (1995), who manually traced the contours of facial features and then used these contours to normalize (warp) each image to a canonical shape. After analyzing both the shape and color images for deviations from the mean, they were able to associate certain shape and color deformations with personal characteristics such as age and gender (Figure 6.14). Their work demonstrates that both shape and color have an important influence on the perception of such characteristics.

Around the same time, researchers in computer vision were beginning to use simultaneous shape deformations and texture interpolation to model the variability in facial appearance caused by identity or expression (Beymer 1996; Vetter and Poggio 1997), developing techniques such as Active Shape Models (Lanitis, Taylor, and Cootes 1997), 3D Morphable Models (Banz and Vetter 1999; Egger, Smith *et al.* 2020), and Elastic Bunch Graph Matching (Wiskott, Fellous *et al.* 1997).¹³

The *active appearance models* (AAMs) of Cootes, Edwards, and Taylor (2001) model both the variation in the shape of an image $\mathbf{s}$, which is normally encoded by the location of key feature points on the image, as well as the variation in texture $\mathbf{t}$, which is normalized to a canonical shape before being analyzed. Both shape and texture are represented as deviations from a mean shape $\bar{\mathbf{s}}$ and texture $\bar{\mathbf{t}}$,

$$\mathbf{s} = \bar{\mathbf{s}} + \mathbf{U}_s \mathbf{a} \quad (6.2)$$

$$\mathbf{t} = \bar{\mathbf{t}} + \mathbf{U}_t \mathbf{a}, \quad (6.3)$$

where the eigenvectors in $\mathbf{U}_s$ and $\mathbf{U}_t$ have been pre-scaled (whitened) so that unit vectors in $\mathbf{a}$ represent one standard deviation of variation observed in the training data. In addition to these principal deformations, the shape parameters are transformed by a global similarity to match the location, size, and orientation of a given face. Similarly, the texture image contains a scale and offset to best match novel illumination conditions.

As you can see, the same appearance parameters $\mathbf{a}$ in (6.2–6.3) simultaneously control both the shape and texture deformations from the mean, which makes sense if we believe them to be correlated. Figure 6.15 shows how moving three standard deviations along each of the first four principal directions ends up changing several correlated factors in a person’s appearance, including expression, gender, age, and identity.

Although active appearance models are primarily designed to accurately capture the variability in appearance and deformation that are characteristic of faces, they can be adapted to face recognition by computing an identity subspace that separates variation in identity from other sources of variability such as lighting, pose, and expression (Costen, Cootes *et al.*

¹³We will look at the application of PCA to 3D head and face modeling and animation in Section 13.6.3.

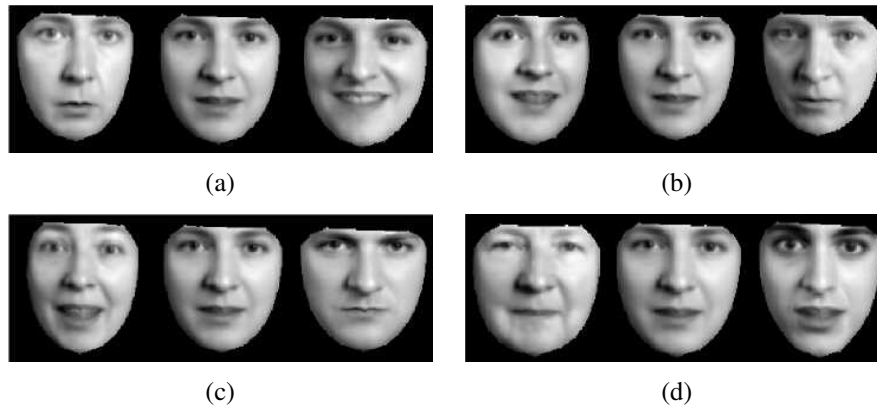


Figure 6.15 *Principal modes of variation in active appearance models (Cootes, Edwards, and Taylor 2001) © 2001 IEEE. The four images show the effects of simultaneously changing the first four modes of variation in both shape and texture by $\pm\sigma$ from the mean. You can clearly see how the shape of the face and the shading are simultaneously affected.*

1999). The basic idea, which is modeled after similar work in eigenfaces (Bellhumeur, Hespanha, and Kriegman 1997; Moghaddam, Jebara, and Pentland 2000), is to compute separate statistics for intrapersonal and extrapersonal variation and then find discriminating directions in these subspaces. While AAMs have sometimes been used directly for recognition (Blanz and Vetter 2003), their main use in the context of recognition is to align faces into a canonical pose (Liang, Xiao *et al.* 2008; Ren, Cao *et al.* 2014) so that more traditional methods of face recognition (Penev and Atick 1996; Wiskott, Fellous *et al.* 1997; Ahonen, Hadid, and Pietikäinen 2006; Zhao and Pietikäinen 2007; Cao, Yin *et al.* 2010) can be used.

Active appearance models have been extended to deal with illumination and viewpoint variation (Gross, Baker *et al.* 2005) as well as occlusions (Gross, Matthews, and Baker 2006). One of the most significant extensions is to construct 3D models of shape (Matthews, Xiao, and Baker 2007), which are much better at capturing and explaining the full variability of facial appearance across wide changes in pose. Such models can be constructed either from monocular video sequences (Matthews, Xiao, and Baker 2007), as shown in Figure 6.16a, or from multi-view video sequences (Ramnath, Koterba *et al.* 2008), which provide even greater reliability and accuracy in reconstruction and tracking (Murphy-Chutorian and Trivedi 2009).

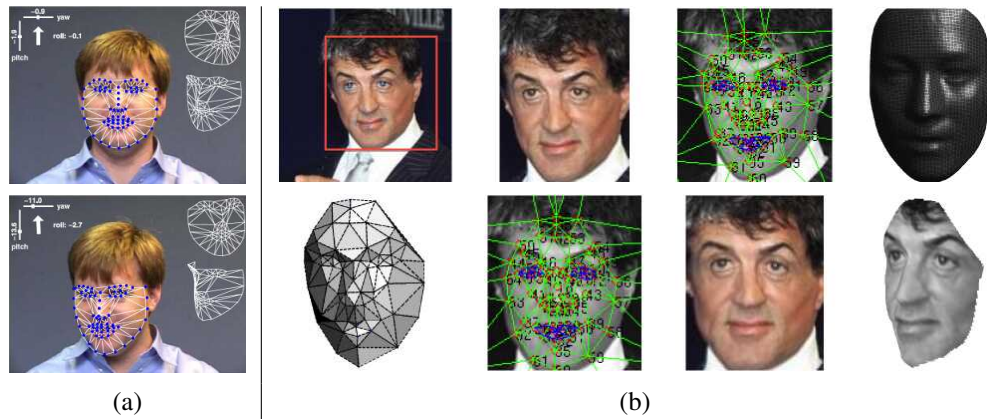


Figure 6.16 Head tracking and frontalization: (a) using 3D active appearance models (AAMs) (Matthews, Xiao, and Baker 2007) © 2007 Springer, showing video frames along with the estimated yaw, pitch, and roll parameters and the fitted 3D deformable mesh; (b) using six and then 67 fiducial points in the DeepFace system (Taigman, Yang et al. 2014) © 2014 IEEE, used to frontalize the face image (bottom row).

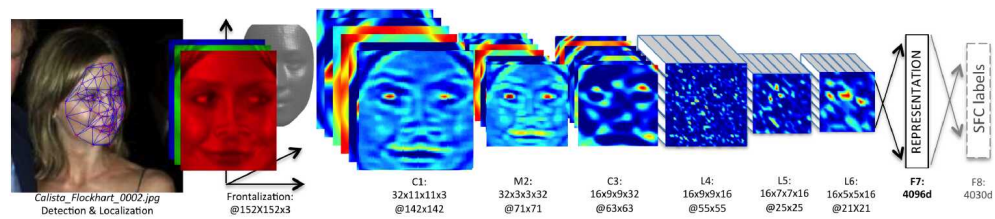


Figure 6.17 The DeepFace architecture (Taigman, Yang et al. 2014) © 2014 IEEE, starts with a frontalization stage, followed by several locally connected (non-convolutional) layers, and then two fully connected layers with a K -class softmax.

Facial recognition using deep learning

Prompted by the dramatic success of deep networks in whole-image categorization, face recognition researchers started using deep neural network backbones as part of their systems. Figures 6.16b–6.17 shows two stages in the DeepFace system of Taigman, Yang *et al.* (2014), which was one of the first systems to realize large gains using deep networks. In their system, a landmark-based pre-processing *frontalization* step is used to convert the original color image into a well-cropped front-looking face. Then, a deep locally connected network (where the convolution kernels can vary spatially) is fed into two final fully connected layers before classification.

Some of the more recent deep face recognizers omit the frontalization stage and instead use data augmentation (Section 5.3.3) to create synthetic inputs with a larger variety of poses (Schroff, Kalenichenko, and Philbin 2015; Parkhi, Vedaldi, and Zisserman 2015). Masi, Wu *et al.* (2018) provide an excellent tutorial and survey on deep face recognition, including a list of widely used training and testing datasets, a discussion of frontalization and dataset augmentation, and a section on training losses (Figure 6.18). This last topic is central to the ability to scale to larger and larger numbers of people. Schroff, Kalenichenko, and Philbin (2015) and Parkhi, Vedaldi, and Zisserman (2015) use triplet losses to construct a low-dimensional embedding space that is independent of the number of subjects. More recent systems use contrastive losses inspired by the softmax function, which we discussed in Section 5.3.4. For example, the ArcFace paper by Deng, Guo *et al.* (2019) measures angular distances on the unit hypersphere in the embedding space and adds an extra margin to get identities to clump together. This idea has been further extended for visual similarity search (Bell, Liu *et al.* 2020) and face recognition (Huang, Shen *et al.* 2020; Deng, Guo *et al.* 2020a).

Personal photo collections

In addition to digital cameras automatically finding faces to aid in auto-focusing and video cameras finding faces in video conferencing to center on the speaker (either mechanically or digitally), face detection has found its way into most consumer-level photo organization packages and photo sharing sites. Finding faces and allowing users to tag them makes it easier to find photos of selected people at a later date or to automatically share them with friends. In fact, the ability to tag friends in photos is one of the more popular features on Facebook.

Sometimes, however, faces can be hard to find and recognize, especially if they are small, turned away from the camera, or otherwise occluded. In such cases, combining face recognition with person detection and clothes recognition can be very effective, as illustrated in Figure 6.19 (Sivic, Zitnick, and Szeliski 2006). Combining person recognition with other

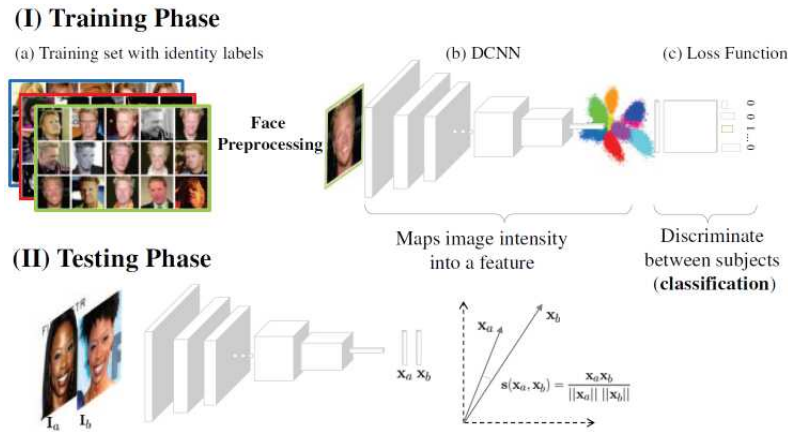


Figure 6.18 A typical modern deep face recognition architecture, from the survey by *Masi, Wu et al. (2018)* © 2018 IEEE. At training time, a huge labeled face set (a) is used to constrain the weights of a DCNN (b), optimizing a loss function (c) for a classification task. At test time, the classification layer is often discarded, and the DCNN is used as a feature extractor for comparing face descriptors.

kinds of context, such as location recognition (Section 11.2.3) or activity or event recognition, can also help boost performance (Lin, Kapoor *et al.* 2010).

6.3 Object detection

If we are given an image to analyze, such as the group portrait in Figure 6.20, we could try to apply a recognition algorithm to every possible sub-window in this image. Such algorithms are likely to be both slow and error-prone. Instead, it is more effective to construct special-purpose *detectors*, whose job it is to rapidly find likely regions where particular objects might occur.

We begin this section with face detectors, which were some of the earliest successful examples of recognition. Such algorithms are built into most of today’s digital cameras to enhance auto-focus and into video conferencing systems to control panning and zooming. We then look at pedestrian detectors, as an example of more general methods for object detection. Finally, we turn to the problem of multi-class object detection, which today is solved using deep neural networks.

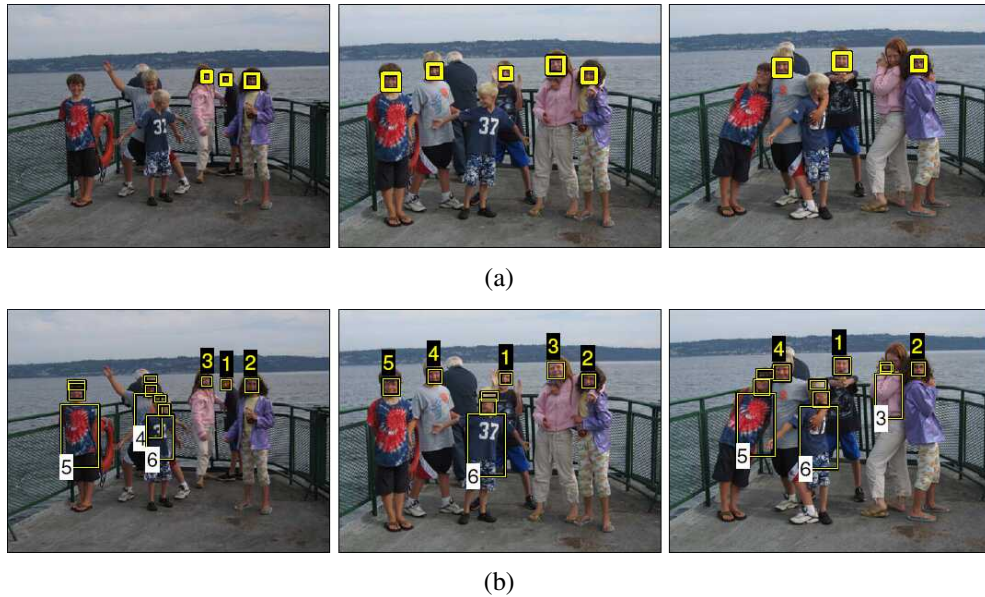


Figure 6.19 Person detection and re-recognition using a combined face, hair, and torso model (Sivic, Zitnick, and Szeliski 2006) © 2006 Springer. (a) Using face detection alone, several of the heads are missed. (b) The combined face and clothing model successfully re-finds all the people.

6.3.1 Face detection

Before face recognition can be applied to a general image, the locations and sizes of any faces must first be found (Figures 6.1c and 6.20). In principle, we could apply a face recognition algorithm at every pixel and scale (Moghaddam and Pentland 1997) but such a process would be too slow in practice.

Over the last four decades, a wide variety of fast face detection algorithms have been developed. Yang, Kriegman, and Ahuja (2002) and Zhao, Chellappa *et al.* (2003) provide comprehensive surveys of earlier work in this field. According to their taxonomy, face detection techniques can be classified as feature-based, template-based, or appearance-based. Feature-based techniques attempt to find the locations of distinctive image features such as the eyes, nose, and mouth, and then verify whether these features are in a plausible geometrical arrangement. These techniques include some of the early approaches to face recognition (Fischler and Elschlager 1973; Kanade 1977; Yuille 1991), as well as later approaches based on modular eigenspaces (Moghaddam and Pentland 1997), local filter jets (Leung, Burl, and



Figure 6.20 *Face detection results produced by Rowley, Baluja, and Kanade (1998) © 1998 IEEE. Can you find the one false positive (a box around a non-face) among the 57 true positive results?*

Perona 1995; Penev and Atick 1996; Wiskott, Fellous *et al.* 1997), support vector machines (Heisele, Ho *et al.* 2003; Heisele, Serre, and Poggio 2007), and boosting (Schneiderman and Kanade 2004).

Template-based approaches, such as active appearance models (AAMs) (Section 6.2.4), can deal with a wide range of pose and expression variability. Typically, they require good initialization near a real face and are therefore not suitable as fast face detectors.

Appearance-based approaches scan over small overlapping rectangular patches of the image searching for likely face candidates, which can then be refined using a *cascade* of more expensive but selective detection algorithms (Sung and Poggio 1998; Rowley, Baluja, and Kanade 1998; Romdhani, Torr *et al.* 2001; Fleuret and Geman 2001; Viola and Jones 2004). To deal with scale variation, the image is usually converted into a sub-octave pyramid and a separate scan is performed on each level. Most appearance-based approaches rely heavily on training classifiers using sets of labeled face and non-face patches.

Sung and Poggio (1998) and Rowley, Baluja, and Kanade (1998) present two of the earliest appearance-based face detectors and introduce a number of innovations that are widely used in later work by others. To start with, both systems collect a set of labeled face patches (Figure 6.20) as well as a set of patches taken from images that are known not to contain faces, such as aerial images or vegetation. The collected face images are augmented by artificially mirroring, rotating, scaling, and translating the images by small amounts to make the face detectors less sensitive to such effects.

The next few paragraphs provide quick reviews of a number of early appearance-based

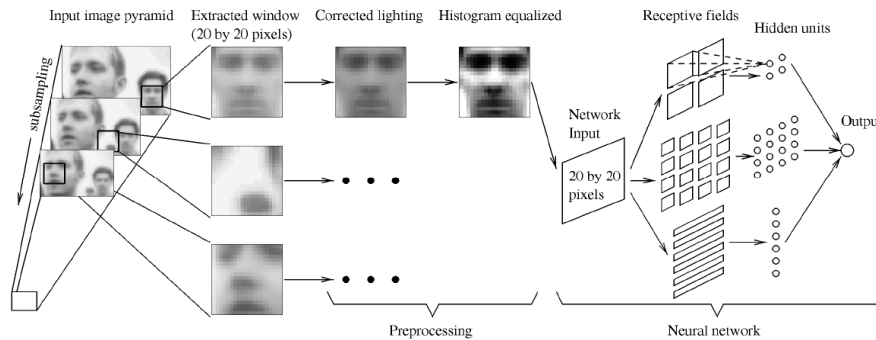


Figure 6.21 A neural network for face detection (Rowley, Baluja, and Kanade 1998) © 1998 IEEE. Overlapping patches are extracted from different levels of a pyramid and then pre-processed. A three-layer neural network is then used to detect likely face locations.

face detectors, keyed by the machine algorithms they are based on. These systems provide an interesting glimpse into the gradual adoption and evolution of machine learning in computer vision. More detailed descriptions can be found in the original papers, as well as the first edition of this book (Szeliski 2010).

Clustering and PCA. Once the face and non-face patterns have been pre-processed, Sung and Poggio (1998) cluster each of these datasets into six separate clusters using k-means and then fit PCA subspaces to each of the resulting 12 clusters. At detection time, the DIFS and DFFS metrics first developed by Moghaddam and Pentland (1997) are used to produce 24 Mahalanobis distance measurements (two per cluster). The resulting 24 measurements are input to a multi-layer perceptron (MLP), i.e., a fully connected neural network.

Neural networks. Instead of first clustering the data and computing Mahalanobis distances to the cluster centers, Rowley, Baluja, and Kanade (1998) apply a neural network (MLP) directly to the 20×20 pixel patches of gray-level intensities, using a variety of differently sized hand-crafted “receptive fields” to capture both large-scale and smaller scale structure (Figure 6.21). The resulting neural network directly outputs the likelihood of a face at the center of every overlapping patch in a multi-resolution pyramid. Since several overlapping patches (in both space and resolution) may fire near a face, an additional merging network is used to merge overlapping detections. The authors also experiment with training several networks and merging their outputs. Figure 6.20 shows a sample result from their face detector.

Support vector machines. Instead of using a neural network to classify patches, Osuna, Freund, and Girosi (1997) use *support vector machines* (SVMs), which we discussed in Section 5.1.4, to classify the same preprocessed patches as Sung and Poggio (1998). An SVM searches for a series of *maximum margin* separating planes in feature space between different classes (in this case, face and non-face patches). In those cases where linear classification boundaries are insufficient, the feature space can be lifted into higher-dimensional features using *kernels* (5.29). SVMs have been used by other researchers for both face detection and face recognition (Heisele, Ho *et al.* 2003; Heisele, Serre, and Poggio 2007) as well as general object recognition (Lampert 2008).

Boosting. Of all the face detectors developed in the 2000s, the one introduced by Viola and Jones (2004) is probably the best known. Their technique was the first to introduce the concept of *boosting* to the computer vision community, which involves training a series of increasingly discriminating simple classifiers and then blending their outputs (Bishop 2006, Section 14.3; Hastie, Tibshirani, and Friedman 2009, Chapter 10; Murphy 2012, Section 16.4; Glassner 2018, Section 14.7).

In more detail, boosting involves constructing a *classifier* $h(\mathbf{x})$ as a sum of simple *weak learners*,

$$h(\mathbf{x}) = \text{sign} \left[\sum_{j=0}^{m-1} \alpha_j h_j(\mathbf{x}) \right], \quad (6.4)$$

where each of the weak learners $h_j(\mathbf{x})$ is an extremely simple function of the input, and hence is not expected to contribute much (in isolation) to the classification performance.

In most variants of boosting, the weak learners are threshold functions,

$$h_j(\mathbf{x}) = a_j[f_j < \theta_j] + b_j[f_j \geq \theta_j] = \begin{cases} a_j & \text{if } f_j < \theta_j \\ b_j & \text{otherwise,} \end{cases} \quad (6.5)$$

which are also known as *decision stumps* (basically, the simplest possible version of *decision trees*). In most cases, it is also traditional (and simpler) to set a_j and b_j to ± 1 , i.e., $a_j = -s_j$, $b_j = +s_j$, so that only the feature f_j , the threshold value θ_j , and the polarity of the threshold $s_j \in \pm 1$ need to be selected.¹⁴

In many applications of boosting, the features are simply coordinate axes x_k , i.e., the boosting algorithm selects one of the input vector components as the best one to threshold. In Viola and Jones' face detector, the features are differences of rectangular regions in the input patch, as shown in Figure 6.22. The advantage of using these features is that, while they are

¹⁴Some variants, such as that of Viola and Jones (2004), use $(a_j, b_j) \in [0, 1]$ and adjust the learning algorithm accordingly.

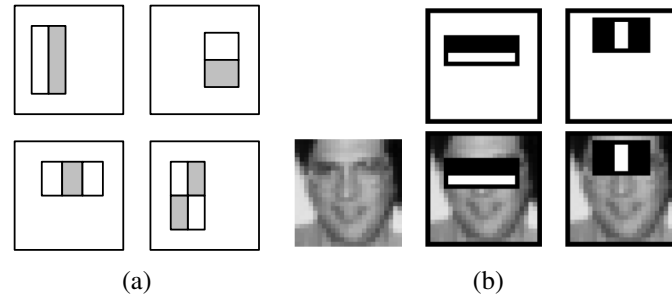


Figure 6.22 Simple features used in boosting-based face detector (Viola and Jones 2004)
 © 2004 Springer: (a) difference of rectangle feature composed of 2–4 different rectangles (pixels inside the white rectangles are subtracted from the gray ones); (b) the first and second features selected by AdaBoost. The first feature measures the differences in intensity between the eyes and the cheeks, the second one between the eyes and the bridge of the nose.

more discriminating than single pixels, they are extremely fast to compute once a summed area table has been precomputed, as described in Section 3.2.3 (3.31–3.32). Essentially, for the cost of an $O(N)$ precomputation phase (where N is the number of pixels in the image), subsequent differences of rectangles can be computed in $4r$ additions or subtractions, where $r \in \{2, 3, 4\}$ is the number of rectangles in the feature.

The key to the success of boosting is the method for incrementally selecting the weak learners and for re-weighting the training examples after each stage. The AdaBoost (Adaptive Boosting) algorithm (Bishop 2006; Hastie, Tibshirani, and Friedman 2009; Murphy 2012) does this by re-weighting each sample as a function of whether it is correctly classified at each stage, and using the stage-wise average classification error to determine the final weightings α_j among the weak classifiers.

To further increase the speed of the detector, it is possible to create a *cascade* of classifiers, where each classifier uses a small number of tests (say, a two-term AdaBoost classifier) to reject a large fraction of non-faces while trying to pass through all potential face candidates (Fleuret and Geman 2001; Viola and Jones 2004; Brubaker, Wu *et al.* 2008).

Deep networks. Since the initial burst of face detection research in the early 2000s, face detection algorithms have continued to evolve and improve (Zafeiriou, Zhang, and Zhang 2015). Researchers have proposed using cascades of features (Li and Zhang 2013), deformable parts models (Mathias, Benenson *et al.* 2014), aggregated channel features (Yang, Yan *et al.* 2014), and neural networks (Li, Lin *et al.* 2015; Yang, Luo *et al.* 2015). The WIDER FACE bench-

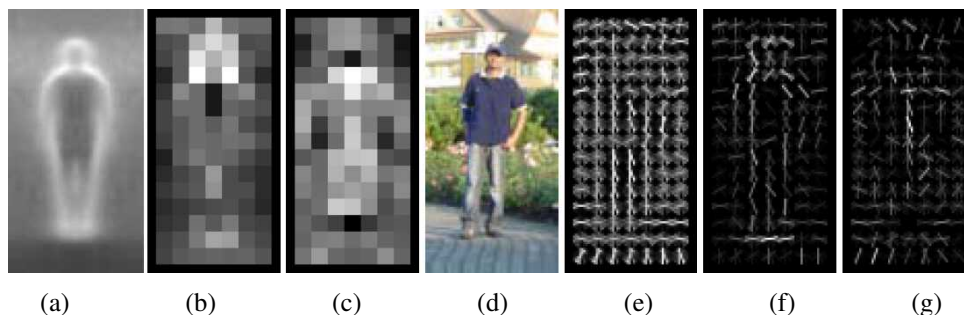


Figure 6.23 *Pedestrian detection using histograms of oriented gradients (Dalal and Triggs 2005) © 2005 IEEE: (a) the average gradient image over the training examples; (b) each “pixel” shows the maximum positive SVM weight in the block centered on the pixel; (c) likewise, for the negative SVM weights; (d) a test image; (e) the computed R-HOG (rectangular histogram of gradients) descriptor; (f) the R-HOG descriptor weighted by the positive SVM weights; (g) the R-HOG descriptor weighted by the negative SVM weights.*

mark^{15,16} (Yang, Luo *et al.* 2016) contains results from, and pointers to, more recent papers, including RetinaFace (Deng, Guo *et al.* 2020b), which combines ideas from other recent neural networks and object detectors such as Feature Pyramid Networks (Lin, Dollár *et al.* 2017) and RetinaNet (Lin, Goyal *et al.* 2017), and also has a nice review of other recent face detectors.

6.3.2 Pedestrian detection

While a lot of the early research on object detection focused on faces, the detection of other objects, such as pedestrians and cars, has also received widespread attention (Gavrila and Philomin 1999; Gavrila 1999; Papageorgiou and Poggio 2000; Mohan, Papageorgiou, and Poggio 2001; Schneiderman and Kanade 2004). Some of these techniques maintained the same focus as face detection on speed and efficiency. Others, however, focused on accuracy, viewing detection as a more challenging variant of generic class recognition (Section 6.3.3) in which the locations and extents of objects are to be determined as accurately as possible (Everingham, Van Gool *et al.* 2010; Everingham, Eslami *et al.* 2015; Lin, Maire *et al.* 2014).

An example of a well-known pedestrian detector is the algorithm developed by Dalal and Triggs (2005), who use a set of overlapping *histogram of oriented gradients* (HOG)

¹⁵<http://shuoyang1213.me/WIDERFACE>

¹⁶The WIDER FACE benchmark has expanded to a larger set of detection challenges and workshops: <https://wider-challenge.org/2019.html>.

descriptors fed into a support vector machine (Figure 6.23). Each HOG has cells to accumulate magnitude-weighted votes for gradients at particular orientations, just as in the scale invariant feature transform (SIFT) developed by Lowe (2004), which we will describe in Section 7.1.2 and Figure 7.16. Unlike SIFT, however, which is only evaluated at interest point locations, HOGs are evaluated on a regular overlapping grid and their descriptor magnitudes are normalized using an even coarser grid; they are only computed at a single scale and a fixed orientation. To capture the subtle variations in orientation around a person's outline, a large number of orientation bins are used and no smoothing is performed in the central difference gradient computation—see Dalal and Triggs (2005) for more implementation details. Figure 6.23d shows a sample input image, while Figure 6.23e shows the associated HOG descriptors.

Once the descriptors have been computed, a support vector machine (SVM) is trained on the resulting high-dimensional continuous descriptor vectors. Figures 6.23b–c show a diagram of the (most) positive and negative SVM weights in each block, while Figures 6.23f–g show the corresponding weighted HOG responses for the central input image. As you can see, there are a fair number of positive responses around the head, torso, and feet of the person, and relatively few negative responses (mainly around the middle and the neck of the sweater).

Much like face detection, the fields of pedestrian and general object detection continued to advance rapidly in the 2000s (Belongie, Malik, and Puzicha 2002; Mikolajczyk, Schmid, and Zisserman 2004; Dalal and Triggs 2005; Leibe, Seemann, and Schiele 2005; Opelt, Pinz, and Zisserman 2006; Torralba 2007; Andriluka, Roth, and Schiele 2009; Maji and Berg 2009; Andriluka, Roth, and Schiele 2010; Dollár, Belongie, and Perona 2010).

A significant advance in the field of person detection was the work of Felzenszwalb, McAllester, and Ramanan (2008), who extend the histogram of oriented gradients person detector to incorporate flexible parts models (Section 6.2.1). Each part is trained and detected on HOGs evaluated at two pyramid levels below the overall object model and the locations of the parts relative to the parent node (the overall bounding box) are also learned and used during recognition (Figure 6.24b). To compensate for inaccuracies or inconsistencies in the training example bounding boxes (dashed white lines in Figure 6.24c), the “true” location of the parent (blue) bounding box is considered a latent (hidden) variable and is inferred during both training and recognition. Since the locations of the parts are also latent, the system can be trained in a semi-supervised fashion, without needing part labels in the training data. An extension to this system (Felzenszwalb, Girshick *et al.* 2010), which includes among its improvements a simple contextual model, was among the two best object detection systems in the 2008 Visual Object Classes detection challenge (Everingham, Van Gool *et al.* 2010).

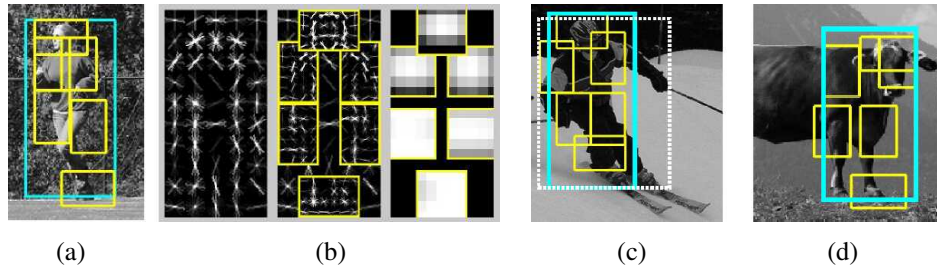


Figure 6.24 Part-based object detection (Felzenszwalb, McAllester, and Ramanan 2008) © 2008 IEEE: (a) An input photograph and its associated person (blue) and part (yellow) detection results. (b) The detection model is defined by a coarse template, several higher resolution part templates, and a spatial model for the location of each part. (c) True positive detection of a skier and (d) false positive detection of a cow (labeled as a person).

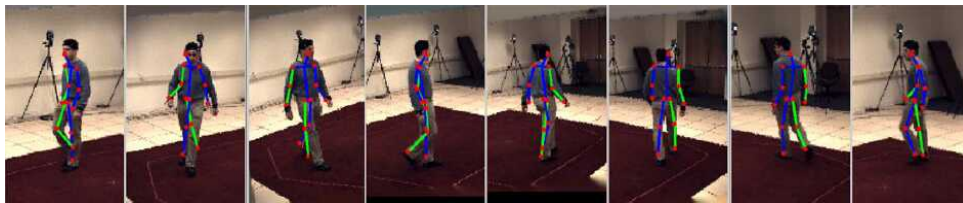


Figure 6.25 Pose detection using random forests (Rogez, Rihan et al. 2008) © 2008 IEEE. The estimated pose (state of the kinematic model) is drawn over each input frame.

Improvements to part-based person detection and pose estimation include work by Andriluka, Roth, and Schiele (2009) and Kumar, Zisserman, and Torr (2009).

An even more accurate estimate of a person's pose and location is presented by Rogez, Rihan *et al.* (2008), who compute both the phase of a person in a walk cycle and the locations of individual joints, using random forests built on top of HOGs (Figure 6.25). Since their system produces full 3D pose information, it is closer in its application domain to 3D person trackers (Sidenbladh, Black, and Fleet 2000; Andriluka, Roth, and Schiele 2010), which we will discuss in Section 13.6.4. When video sequences are available, the additional information present in the optical flow and motion discontinuities can greatly aid in the detection task, as discussed by Efros, Berg *et al.* (2003), Viola, Jones, and Snow (2003), and Dalal, Triggs, and Schmid (2006).

Since the 2000s, pedestrian and general person detection have continued to be actively developed, often in the context of more general multi-class object detection (Everingham, Van Gool *et al.* 2010; Everingham, Eslami *et al.* 2015; Lin, Maire *et al.* 2014). The Cal-

tech pedestrian detection benchmark¹⁷ and survey by Dollár, Belongie, and Perona (2010) introduces a new dataset and provides a nice review of algorithms through 2012, including Integral Channel Features (Dollár, Tu *et al.* 2009), the Fastest Pedestrian Detector in the West (Dollár, Belongie, and Perona 2010), and 3D pose estimation algorithms such as Poselets (Bourdev and Malik 2009). Since its original construction, this benchmark continues to tabulate and evaluate more recent detectors, including Dollár, Appel, and Kienzle (2012), Dollár, Appel *et al.* (2014), and more recent algorithms based on deep neural networks (Sermanet, Kavukcuoglu *et al.* 2013; Ouyang and Wang 2013; Tian, Luo *et al.* 2015; Zhang, Lin *et al.* 2016). The CityPersons dataset (Zhang, Benenson, and Schiele 2017) and WIDER Face and Person Challenge¹⁸ also report results on recent algorithms.

6.3.3 General object detection

While face and pedestrian detection algorithms were the earliest to be extensively studied, computer vision has always been interested in solving the general object detection and labeling problem, in addition to whole-image classification. The PASCAL Visual Object Classes (VOC) Challenge (Everingham, Van Gool *et al.* 2010), which contained 20 classes, had both classification and detection challenges. Early entries that did well on the detection challenge include a feature-based detector and spatial pyramid matching SVM classifier by Chum and Zisserman (2007), a star-topology deformable part model by Felzenszwalb, McAllester, and Ramanan (2008), and a sliding window SVM classifier by Lampert, Blaschko, and Hofmann (2008). The competition was re-run annually, with the two top entries in the 2012 detection challenge (Everingham, Eslami *et al.* 2015) using a sliding window spatial pyramid matching (SPM) SVM (de Sande, Uijlings *et al.* 2011) and a University of Oxford re-implementation of a deformable parts model (Felzenszwalb, Girshick *et al.* 2010).

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC), released in 2010, scaled up the dataset from around 20 thousand images in PASCAL VOC 2010 to over 1.4 million in ILSVRC 2010, and from 20 object classes to 1,000 object classes (Russakovsky, Deng *et al.* 2015). Like PASCAL, it also had an object detection task, but it contained a much wider range of challenging images (Figure 6.4). The Microsoft COCO (Common Objects in Context) dataset (Lin, Maire *et al.* 2014) contained even more objects per image, as well as pixel-accurate segmentations of multiple objects, enabling the study of not only *semantic segmentation* (Section 6.4), but also individual object *instance segmentation* (Section 6.4.2). Table 6.2 list some of the datasets used for training and testing general object detection algorithms.

¹⁷http://www.vision.caltech.edu/Image_Datasets/CaltechPedestrians

¹⁸<https://wider-challenge.org/2019.html>

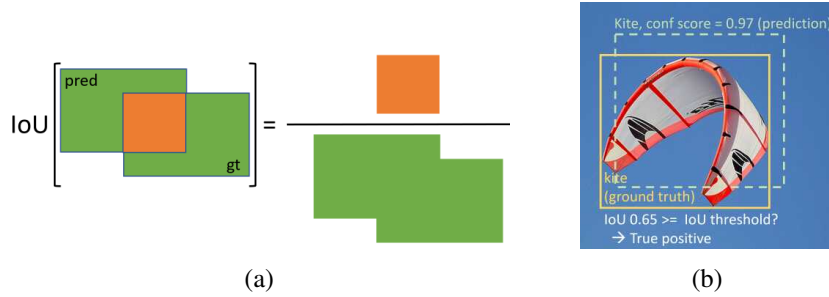


Figure 6.26 *Intersection over union (IoU): (a) schematic formula, (b) real-world example*
 © 2020 Ross Girshick.

The release of COCO coincided with a wholesale shift to deep networks for image classification, object detection, and segmentation (Jiao, Zhang *et al.* 2019; Zhao, Zheng *et al.* 2019). Figure 6.29 shows the rapid improvements in average precision (AP) on the COCO object detection task, which correlates strongly with advances in deep neural network architectures (Figure 5.40).

Precision vs. recall

Before we describe the elements of modern object detectors, we should first discuss what metrics they are trying to optimize. The main task in object detection, as illustrated in Figures 6.5a and 6.26b, is to put accurate bounding boxes around all the objects of interest and to correctly label such objects. To measure the accuracy of each bounding box (not too small and not too big), the common metric is *intersection over union* (IoU), which is also known as the *Jaccard index* or *Jaccard similarity coefficient* (Rezatoflghi, Tsoi *et al.* 2019). The IoU is computed by taking the predicted and ground truth bounding boxes B_{pr} and B_{gt} for an object and computing the ratio of their area of intersection and their area of union,

$$IoU = \frac{B_{pr} \cap B_{gt}}{B_{pr} \cup B_{gt}}, \quad (6.6)$$

as shown in Figure 6.26a.

As we will shortly see, object detectors operate by first proposing a number of plausible rectangular regions (detections) and then classifying each detection while also producing a confidence score (Figure 6.26b). These regions are then run through some kind of *non-maximal suppression* (NMS) stage, which removes weaker detections that have too much overlap with stronger detections, using a greedy most-confident-first algorithm.

To evaluate the performance of an object detector, we run through all of the detections,

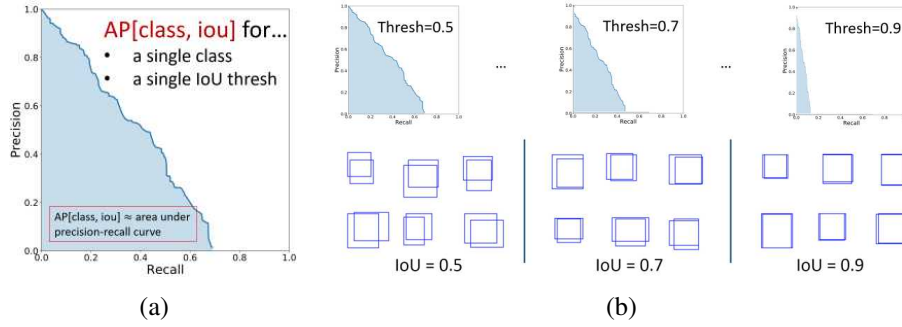


Figure 6.27 Object detector average precision © 2020 Ross Girshick: (a) a precision-recall curve for a single class and IoU threshold, with the AP being the area under the P-R curve; (b) average precision averaged over several IoU thresholds (from looser to tighter).

from most confident to least, and classify them as *true positive* TP (correct label and sufficiently high IoU) or *false positive* FP (incorrect label or ground truth object already matched). For each new decreasing confidence threshold, we can compute the *precision* and *recall* as

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (6.7)$$

$$\text{recall} = \frac{\text{TP}}{P}, \quad (6.8)$$

where P is the number of positive examples, i.e., the number of labeled ground truth detections in the test image.¹⁹ (See Section 7.1.3 on feature matching for additional terms that are often used in measuring and describing error rates.)

Computing the precision and recall at every confidence threshold allows us to populate a precision-recall curve, such as the one in Figure 6.27a. The area under this curve is called *average precision* (AP). A separate AP score can be computed for each class being detected, and the results averaged to produce a *mean average precision* (mAP). Another widely used measure is the While earlier benchmarks such as PASCAL VOC determined the mAP using a single IoU threshold of 0.5 (Everingham, Eslami *et al.* 2015), the COCO benchmark (Lin, Maire *et al.* 2014) averages the mAP over a set of IoU thresholds, $\text{IoU} \in \{0.50, 0.55, \dots, 0.95\}$, as shown in Figure 6.27a. While this AP score continues to be widely used, an alternative *probability-based detection quality* (PDQ) score has recently been proposed (Hall, Dayoub *et al.* 2020). A smoother version of average precision called Smooth-AP has also been proposed and shown to have benefits on large-scale image retrieval tasks (Brown, Xie *et al.* 2020).

¹⁹ Another widely reported measure is the *F-score*, which is the harmonic mean of the precision and recall.

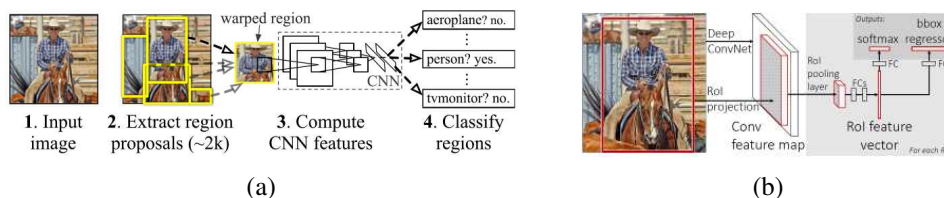


Figure 6.28 The R-CNN and Fast R-CNN object detectors. (a) R-CNN rescales pixels inside each proposal region and performs a CNN + SVM classification (Girshick, Donahue et al. 2015) © 2015 IEEE. (b) Fast R-CNN resamples convolutional features and uses fully connected layers to perform classification and bounding box regression (Girshick 2015) © 2015 IEEE.

Modern object detectors

The first stage in detecting objects in an image is to propose a set of plausible rectangular regions in which to run a classifier. The development of such *region proposal* algorithms was an active research area in the early 2000s (Alexe, Deselaers, and Ferrari 2012; Uijlings, Van De Sande et al. 2013; Cheng, Zhang et al. 2014; Zitnick and Dollár 2014).

One of the earliest object detectors based on neural networks is R-CNN, the Region-based Convolutional Network developed by Girshick, Donahue et al. (2014). As illustrated in Figure 6.28a, this detector starts by extracting about 2,000 region proposals using the selective search algorithm of Uijlings, Van De Sande et al. (2013). Each proposed regions is then rescaled (warped) to a 224 square image and passed through an AlexNet or VGG neural network with a support vector machine (SVM) final classifier.

The follow-on Fast R-CNN paper by Girshick (2015) interchanges the convolutional neural network and region extraction stages and replaces the SVM with some fully connected (FC) layers, which compute both an object class and a bounding box refinement (Figure 6.28b). This reuses the CNN computations and leads to much faster training and test times, as well as dramatically better accuracy compared to previous networks (Figure 6.29). As you can see from Figure 6.28b, Fast R-CNN is an example of a deep network with a shared *backbone* and two separate *heads*, and hence two different loss functions, although these terms were not introduced until the Mask R-CNN paper by He, Gkioxari et al. (2017).

The Faster R-CNN system, introduced a few month later by Ren, He et al. (2015), replaces the relatively slow selective search stage with a convolutional region proposal network (RPN), resulting in much faster inference. After computing convolutional features, the RPN suggests at each coarse location a number of potential *anchor boxes*, which vary in shape and size

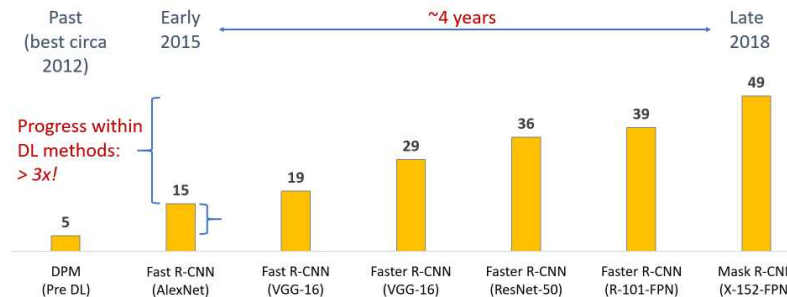


Figure 6.29 Best average precision (AP) results by year on the COCO object detection task (Lin, Maire et al. 2014) © 2020 Ross Girshick.

to accommodate different potential objects. Each proposal is then classified and refined by an instance of the Fast R-CNN heads and the final detections are ranked and merged using non-maximal suppression.

R-CNN, Fast R-CNN, and Faster R-CNN all operate on a single resolution convolutional feature map (Figure 6.30b). To obtain better scale invariance, it would be preferable to operate on a range of resolutions, e.g. by computing a feature map at each image pyramid level, as shown in Figure 6.30a, but this is computationally expensive. We could, instead, simply start with the various levels inside the convolutional network (Figure 6.30c), but these levels have different degrees of semantic abstraction, i.e., higher/smaller levels are attuned to more abstract constructs. The best solution is to construct a *Feature Pyramid Network* (FPN), as shown in Figure 6.30d, where top-down connections are used to endow higher-resolution (lower) pyramid levels with the semantics inferred at higher levels (Lin, Dollár et al. 2017).²⁰ This additional information significantly enhances the performance of object detectors (and other downstream tasks) and makes their behavior much less sensitive to object size.

DETR (Carion, Massa et al. 2020) uses a simpler architecture that eliminates the use of non-maximum suppression and anchor generation. Their model consists of a ResNet backbone that feeds into a transformer encoder-decoder. At a high level, it makes N bounding box predictions, some of which may include the “no object class”. The ground truth bounding boxes are also padded with “no object class” bounding boxes to obtain N total bounding boxes. During training, *bipartite matching* is then used to build a one-to-one mapping from every predicted bounding box to a ground truth bounding box, with the chosen mapping lead-

²⁰It’s interesting to note that the human visual system is full of such re-entrant or feedback pathways (Gilbert and Li 2013), although the extent to which cognition influences perception is still being debated (Firestone and Scholl 2016).

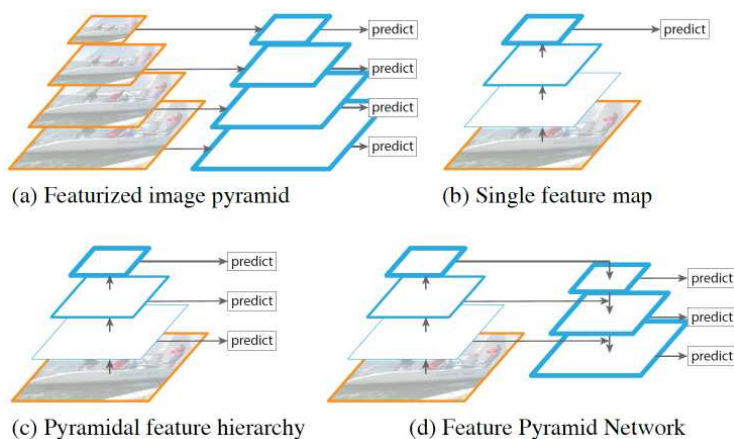


Figure 6.30 A Feature Pyramid Network and its precursors (Lin, Dollár et al. 2017) © 2017 IEEE: (a) deep features extracted at each level in an image pyramid; (b) a single low-resolution feature map; (c) a deep feature pyramid, with higher levels having greater abstraction; (d) a Feature Pyramid Network, with top-down context for all levels.

ing to the lowest possible cost. The overall training loss is then the sum of the losses between the matched bounding boxes. They find that their approach is competitive with state-of-the-art object detection performance on COCO.

Single-stage networks

In the architectures we’ve looked at so far, a region proposal algorithm or network selects the locations and shapes of the detections to be considered, and a second network is then used to classify and regress the pixels or features inside each region. An alternative is to use a *single-stage network*, which uses a single neural network to output detections at a variety of locations. Two examples of such detectors are SSD (Single Shot MultiBox Detector) from Liu, Anguelov et al. (2016) and the family of YOLO (You Only Look Once) detectors described in Redmon, Divvala et al. (2016), Redmon and Farhadi (2017), and Redmon and Farhadi (2018). RetinaNet (Lin, Goyal et al. 2017) is also a single-stage detector built on top of a feature pyramid network. It uses a *focal loss* to focus the training on hard examples by downweighting the loss on well-classified samples, thus preventing the larger number of easy negatives from overwhelming the training. These and more recent convolutional object detectors are described in the recent survey by Jiao, Zhang et al. (2019). Figure 6.31 shows the speed and accuracy of detectors published up through early 2017.

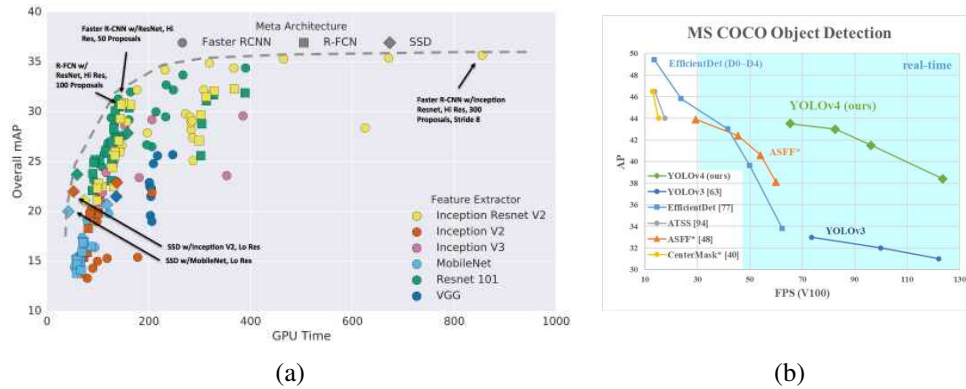


Figure 6.31 Speed/accuracy trade-offs for convolutional object detectors: (a) (Huang, Rathod et al. 2017) © 2017 IEEE; (b) YOLOv4 © Bochkovskiy, Wang, and Liao (2020).

The latest in the family of YOLO detectors is YOLOv4 by Bochkovskiy, Wang, and Liao (2020). In addition to outperforming other recent fast detectors such as EfficientDet (Tan, Pang, and Le 2020), as shown in Figure 6.31b, the paper breaks the processing pipeline into several stages, including a *neck*, which performs the top-down feature enhancement found in the feature pyramid network. The paper also evaluates many different components, which they categorize into a “bag of freebies” that can be used during training and a “bag of specials” that can be used at detection time with minimal additional cost.

While most bounding box object detectors continue to evaluate their results on the COCO dataset (Lin, Maire et al. 2014),²¹ newer datasets such as Open Images (Kuznetsova, Rom et al. 2020), and LVIS: Large Vocabulary Instance Segmentation (Gupta, Dollár, and Girshick 2019) are now also being used (see Table 6.2). Two recent workshops that highlight the latest results using these datasets are Zendel et al. (2020) and Kirillov, Lin et al. (2020) and also have challenges related to instance segmentation, panoptic segmentation, keypoint estimation, and dense pose estimation, which are topics we discuss later in this chapter. Open-source frameworks for training and fine-tuning object detectors include the TensorFlow Object Detection API²² and PyTorch’s Detectron2.²³

Name/URL	Extents	Contents/Reference
<i>Object recognition</i>		
Oxford buildings dataset https://www.robots.ox.ac.uk/~vgg/data/oxbuildings	Pictures of buildings	5,062 images Philbin, Chum <i>et al.</i> (2007)
INRIA Holidays https://lear.inrialpes.fr/people/jegou/data.php	Holiday scenes	1,491 images Jégou, Douze, and Schmid (2008)
PASCAL http://host.robots.ox.ac.uk/pascal/VOC	Segmentations, boxes	11k images (2.9k with segmentations) Everingham, Eslami <i>et al.</i> (2015)
ImageNet https://www.image-net.org	Complete images	21k (WordNet) classes, 14M images Deng, Dong <i>et al.</i> (2009)
Fashion MNIST https://github.com/zalandoresearch/fashion-mnist	Complete images	70k fashion products Xiao, Rasul, and Vollgraf (2017)
<i>Object detection and segmentation</i>		
Caltech Pedestrian Dataset http://www.vision.caltech.edu/Image_Datasets/CaltechPedestrians	Bounding boxes	Pedestrians Dollár, Wojek <i>et al.</i> (2009)
MSR Cambridge https://www.microsoft.com/en-us/research/project/image-understanding	Per-pixel segmentations	23 classes Shotton, Winn <i>et al.</i> (2009)
LabelMe dataset http://labelme.csail.mit.edu	Polygonal boundaries	>500 categories Russell, Torralba <i>et al.</i> (2008)
Microsoft COCO https://cocodataset.org	Segmentations, boxes	330k images Lin, Maire <i>et al.</i> (2014)
Cityscapes https://www.cityscapes-dataset.com	Polygonal boundaries	30 classes, 25,000 images Cordts, Omran <i>et al.</i> (2016)
Broden http://netdissect.csail.mit.edu	Segmentation masks	A variety of visual concepts Bau, Zhou <i>et al.</i> (2017)
Broden+ https://github.com/CSAILVision/unifiedparsing	Segmentation masks	A variety of visual concepts Xiao, Liu <i>et al.</i> (2018)
LVIS https://www.lvisdataset.org	Instance segmentations	1,000 categories, 2.2M images Gupta, Dollár, and Girshick (2019)
Open Images https://g.co/dataset/openimages	Segs., relationships	478k images, 3M relationships Kuznetsova, Rom <i>et al.</i> (2020)

Table 6.2 *Image databases for classification, detection, and localization.*



Figure 6.32 Examples of image segmentation (Kirillov, He et al. 2019) © 2019 IEEE: (a) original image; (b) semantic segmentation (per-pixel classification); (c) instance segmentation (delineate each object); (d) panoptic segmentation (label all things and stuff).

6.4 Semantic segmentation

A challenging version of general object recognition and scene understanding is to simultaneously perform recognition and accurate boundary segmentation (Fergus 2007). In this section, we examine a number of related problems, namely *semantic segmentation* (per-pixel class labeling), *instance segmentation* (accurately delineating each separate object), *panoptic segmentation* (labeling both objects and stuff), and dense pose estimation (labeling pixels belonging to people and their body parts). Figures 6.32 and 6.43 show some of these kinds of segmentations.

The basic approach to simultaneous recognition and segmentation is to formulate the problem as one of labeling every pixel in an image with its class membership. Older approaches often did this using energy minimization or Bayesian inference techniques, i.e., conditional random fields (Section 4.3.1). The TextonBoost system of Shotton, Winn *et al.* (2009) uses unary (pixel-wise) potentials based on image-specific color distributions (Section 4.3.2), location information (e.g., foreground objects are more likely to be in the middle of the image, sky is likely to be higher, and road is likely to be lower), and novel texture-layout classifiers trained using shared boosting. It also uses traditional pairwise potentials that look at image color gradients. The texton-layout features first filter the image with a series of 17 oriented filter banks and then cluster the responses to classify each pixel into 30 different texton classes (Malik, Belongie *et al.* 2001). The responses are then filtered using offset rectangular regions trained with joint boosting (Viola and Jones 2004) to produce the texton-layout features used as unary potentials. Figure 6.33 shows some examples of images successfully labeled and segmented using TextonBoost

The TextonBoost conditional random field framework has been extended to LayoutCRFs

²¹See <https://codalab.org> for the latest competitions and leaderboards.

²²https://github.com/tensorflow/models/tree/master/research/object_detection

²³<https://github.com/facebookresearch/detectron2>

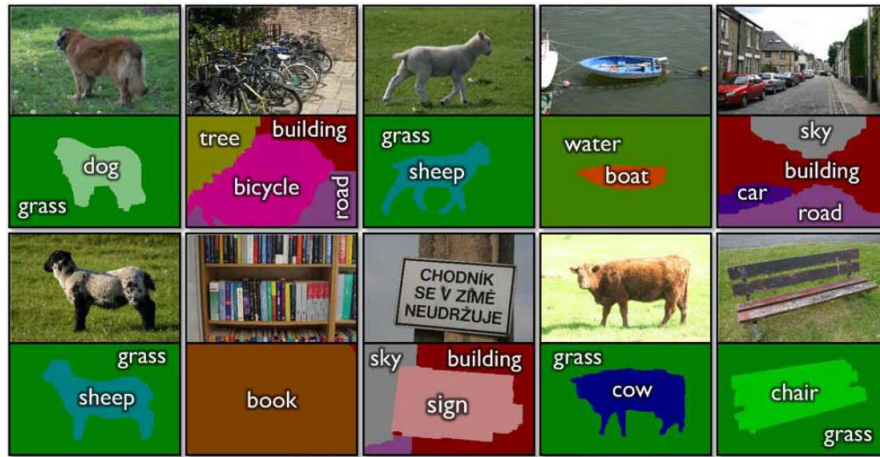


Figure 6.33 Simultaneous recognition and segmentation using TextonBoost (Shotton, Winn et al. 2009) © 2009 Springer.

by Winn and Shotton (2006), who incorporate additional constraints to recognize multiple object instances and deal with occlusions, and by Hoiem, Rother, and Winn (2007) to incorporate full 3D models. Conditional random fields continued to be widely used and extended for simultaneous recognition and segmentation applications, as described in the first edition of this book (Szeliski 2010, Section 14.4.3), along with approaches that first performed low-level or hierarchical segmentations (Section 7.5).

The development of fully convolutional networks (Long, Shelhamer, and Darrell 2015), which we described in Section 5.4.1, enabled per-pixel semantic labeling using a single neural network. While the first networks suffered from poor resolution (very loose boundaries), the addition of conditional random fields at a final stage (Chen, Papandreou et al. 2018; Zheng, Jayasumana et al. 2015), deconvolutional upsampling (Noh, Hong, and Han 2015), and fine-level connections in U-nets (Ronneberger, Fischer, and Brox 2015), all helped improve accuracy and resolution.

Modern semantic segmentation systems are often built on architectures such as the feature pyramid network (Lin, Dollár et al. 2017), which have top-down connections to help percolate semantic information down to higher-resolution maps. For example, the Pyramid Scene Parsing Network (PSPNet) of Zhao, Shi et al. (2017) uses spatial pyramid pooling (He, Zhang et al. 2015) to aggregate features at various resolution levels. The Unified Perceptual Parsing network (UPerNet) of Xiao, Liu et al. (2018) uses both a feature pyramid network and a pyramid pooling module to label image pixels not only with object categories but also

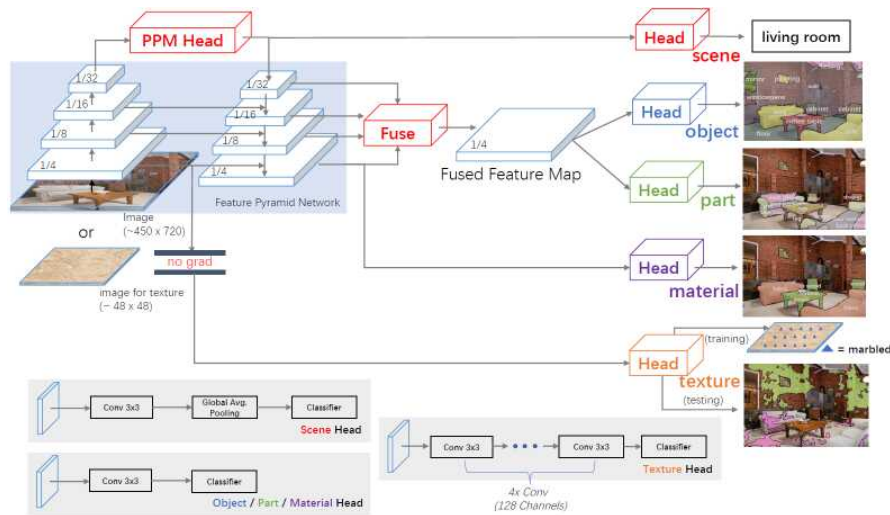


Figure 6.34 The UPerNet framework for Unified Perceptual Parsing (Xiao, Liu *et al.* 2018) © 2018 Springer. A Feature Pyramid Network (FPN) backbone is appended with a Pyramid Pooling Module (PPM) before feeding it into the top-down branch of the FPN. Various layers of the FPN and/or PPM are fed into different heads, including a scene head for image classification, object and part heads from the fused FPN features, a material head operating on the finest level of the FPN, and a texture head that does not participate in the FPN fine tuning. The bottom gray squares give more details into some of the heads.

materials, parts, and textures, as shown in Figure 6.34. HRNet (Wang, Sun *et al.* 2020) keeps high-resolution versions of feature maps throughout the pipeline with occasional interchange of information between channels at different resolution layers. Such networks can also be used to estimate surface normals and depths in an image (Huang, Zhou *et al.* 2019; Wang, Geraghty *et al.* 2020).

Semantic segmentation algorithms were initially trained and tested on datasets such as MSRC (Shotton, Winn *et al.* 2009) and PASCAL VOC (Everingham, Eslami *et al.* 2015). More recent datasets include the Cityscapes dataset for urban scene understanding (Cordts, Omran *et al.* 2016) and ADE20K (Zhou, Zhao *et al.* 2019), which labels pixels in a wider variety of indoor and outdoor scenes with 150 different category and part labels. The Broadly and Densely Labeled Dataset (Broden) created by Bau, Zhou *et al.* (2017) federates a number of such densely labeled datasets, including ADE20K, Pascal-Context, Pascal-Part, OpenSurfaces, and Describable Textures to obtain a wide range of labels such as materials and textures

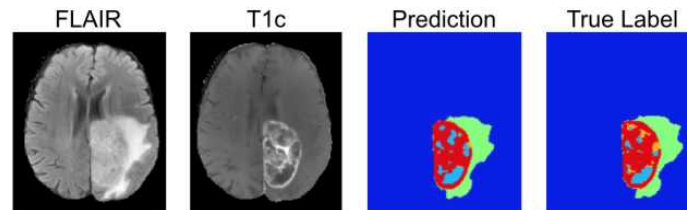


Figure 6.35 3D volumetric medical image segmentation using a deep network (Kamnitsas, Ferrante *et al.* 2016) © 2016 Springer.

in addition to basic object semantics. While this dataset was originally developed to aid in the interpretability of deep networks, it has also proven useful (with extensions) for training unified multi-task labeling systems such as UPerNet (Xiao, Liu *et al.* 2018). Table 6.2 list some of the datasets used for training and testing semantic segmentation algorithms.

One final note. While semantic image segmentation and labeling have widespread applications in image understanding, the converse problem of going from a semantic sketch or painting of a scene to a photorealistic image has also received widespread attention (Johnson, Gupta, and Fei-Fei 2018; Park, Liu *et al.* 2019; Bau, Strobel *et al.* 2019; Ntavelis, Romero *et al.* 2020b). We look at this topic in more detail in Section 10.5.3 on semantic image synthesis.

6.4.1 Application: Medical image segmentation

One of the most promising applications of image segmentation is in the medical imaging domain, where it can be used to segment anatomical tissues for later quantitative analysis. Figure 4.21 shows a binary graph cut with directed edges being used to segment the liver tissue (light gray) from its surrounding bone (white) and muscle (dark gray) tissue. Figure 6.35 shows the segmentation of a brain scan for the detection of brain tumors. Before the development of the mature optimization and deep learning techniques used in modern image segmentation algorithms, such processing required much more laborious manual tracing of individual X-ray slices.

Initially, optimization techniques such as Markov random fields (Section 4.3.2) and discriminative classifiers such as random forests (Section 5.1.5) were used for medical image segmentation (Criminisi, Robertson *et al.* 2013). More recently, the field has shifted to deep learning approaches (Kamnitsas, Ferrante *et al.* 2016; Kamnitsas, Ledig *et al.* 2017; Havaei, Davy *et al.* 2017).

The fields of medical image segmentation (McInerney and Terzopoulos 1996) and medical image registration (Kybic and Unser 2003) (Section 9.2.3) are rich research fields with



Figure 6.36 Instance segmentation using Mask R-CNN (He, Gkioxari et al. 2017) © 2017 IEEE: (a) system architecture, with an additional segmentation branch; (b) sample results.

their own specialized conferences, such as *Medical Imaging Computing and Computer Assisted Intervention (MICCAI)*, and journals, such as *Medical Image Analysis* and *IEEE Transactions on Medical Imaging*. These can be great sources of references and ideas for research in this area.

6.4.2 Instance segmentation

Instance segmentation is the task of finding all of the relevant objects in an image and producing pixel-accurate masks for their visible regions (Figure 6.36b). One potential approach to this task is to perform known object instance recognition (Section 6.1) and to then back-project the object model into the scene (Lowe 2004), as shown in Figure 6.1d, or matching portions of the new scene to pre-learned (segmented) object models (Ferrari, Tuytelaars, and Van Gool 2006b; Kannala, Rahtu et al. 2008). However, this approach only works for known rigid 3D models.

For more complex (flexible) object models, such as those for humans, a different approach is to pre-segment the image into larger or smaller pieces (Section 7.5) and to then match such pieces to portions of the model (Mori, Ren et al. 2004; Mori 2005; He, Zemel, and Ray 2006; Gu, Lim et al. 2009). For general highly variable classes, a related approach is to vote for potential object locations and scales based on feature correspondences and to then infer the object extents (Leibe, Leonardis, and Schiele 2008).

With the advent of deep learning, researchers started combining region proposals or image pre-segmentations with convolutional second stages to infer the final instance segmentations (Hariharan, Arbeláez et al. 2014; Hariharan, Arbeláez et al. 2015; Dai, He, and Sun 2015; Pinheiro, Lin et al. 2016; Dai, He, and Sun 2016; Li, Qi et al. 2017).

A breakthrough in instance segmentation came with the introduction of Mask R-CNN



Figure 6.37 Person keypoint detection and segmentation using Mask R-CNN (He, Gkioxari et al. 2017) © 2017 IEEE

(He, Gkioxari *et al.* 2017). As shown in Figure 6.36a, Mask R-CNN uses the same region proposal network as Faster R-CNN (Ren, He *et al.* 2015), but then adds an additional *branch* for predicting the object *mask*, in addition to the existing branch for bounding box refinement and classification.²⁴ As with other networks that have multiple branches (or heads) and outputs, the training losses corresponding to each supervised output need to be carefully balanced. It is also possible to add additional branches, e.g., branches trained to detect human keypoint locations (implemented as per-keypoint mask images), as shown in Figure 6.37.

Since its introduction, the performance of Mask R-CNN and its extensions has continued to improve with advances in backbone architectures (Liu, Qi *et al.* 2018; Chen, Pang *et al.* 2019). Two recent workshops that highlight the latest results in this area are the COCO + LVIS Joint Recognition Challenge (Kirillov, Lin *et al.* 2020) and the Robust Vision Challenge (Zendel *et al.* 2020).²⁵ It is also possible to replace the pixel masks produced by most instance segmentation techniques with time-evolving closed contours, i.e., “snakes” (Section 7.3.1), as in Peng, Jiang *et al.* (2020). In order to encourage higher-quality segmentation boundaries, Cheng, Girshick *et al.* (2021) propose a new Boundary Intersection-over-Union (Boundary IoU) metric to replace the commonly used Mask IoU metric.

6.4.3 Panoptic segmentation

As we have seen, semantic segmentation classifies each pixel in an image into its semantic category, i.e., what *stuff* does each pixel correspond to. Instance segmentation associates pixels with individual objects, i.e., how many *objects* are there and what are their extents

²⁴Mask R-CNN was the first paper to introduce the terms *backbone* and *head* to describe the common deep convolutional feature extraction front end and the specialized back end branches.

²⁵You can find the leaderboards for instance segmentation and other COCO recognition tasks at <https://cocodataset.org>.



Figure 6.38 *Panoptic segmentation results produced using a Panoptic Feature Pyramid Network (Kirillov, Girshick et al. 2019) © 2019 IEEE.*

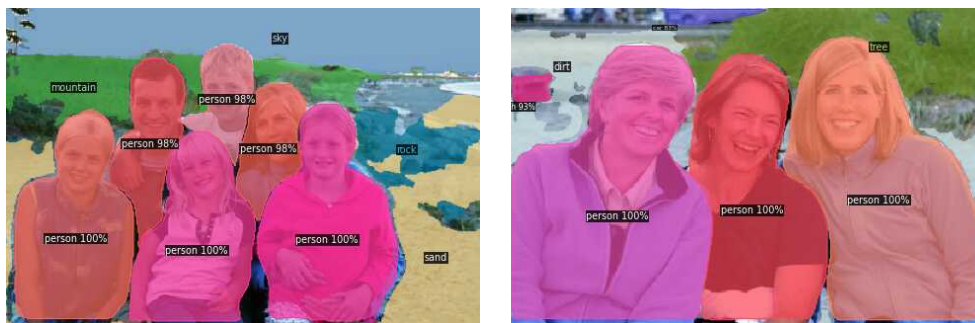


Figure 6.39 *Detectron2 panoptic segmentation results on some of my personal photos. (Click on the “Colab Notebook” link at <https://github.com/facebookresearch/detectron2> and then edit the input image URL to try your own.)*

(Figure 6.32). Putting both of these systems together has long been a goal of semantic scene understanding (Yao, Fidler, and Urtasun 2012; Tighe and Lazechnik 2013; Tu, Chen *et al.* 2005). Doing this on a per-pixel level results in a *panoptic segmentation* of the scene, where all of the objects are correctly segmented and the remaining stuff is correctly labeled (Kirillov, He *et al.* 2019). Producing a sensible *panoptic quality* (PQ) metric that simultaneously balances the accuracy on both tasks takes some careful design. In their paper, Kirillov, He *et al.* (2019) describe their proposed metric and analyze the performance of both humans (in terms of consistency) and recent algorithms on three different datasets.

The COCO dataset has now been extended to include a panoptic segmentation task, on which some recent results can be found in the ECCV 2020 workshop on this topic (Kirillov, Lin *et al.* 2020). Figure 6.38 show some segmentations produced by the panoptic feature pyramid network described by Kirillov, Girshick *et al.* (2019), which adds two branches for instance segmentation and semantic segmentation to a feature pyramid network.

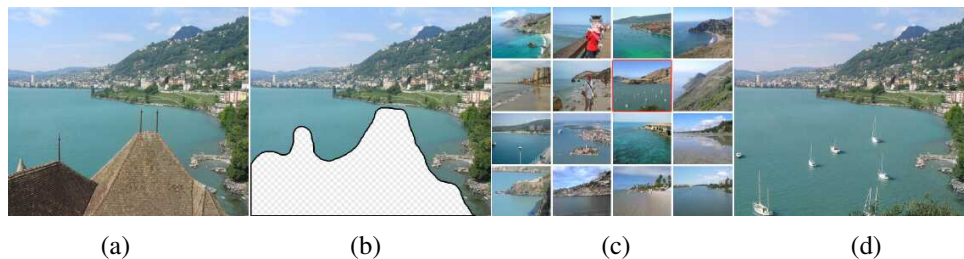


Figure 6.40 Scene completion using millions of photographs (Hays and Efros 2007) © 2007 ACM: (a) original image; (b) after unwanted foreground removal; (c) plausible scene matches, with the one the user selected highlighted in red; (d) output image after replacement and blending.

6.4.4 Application: Intelligent photo editing

Advances in object recognition and scene understanding have greatly increased the power of intelligent (semi-automated) photo editing applications. One example is the Photo Clip Art system of Lalonde, Hoiem *et al.* (2007), which recognizes and segments objects of interest, such as pedestrians, in internet photo collections and then allows users to paste them into their own photos. Another is the scene completion system of Hays and Efros (2007), which tackles the same *inpainting* problem we will study in Section 10.5. Given an image in which we wish to erase and fill in a large section (Figure 6.40a–b), where do you get the pixels to fill in the gaps in the edited image? Traditional approaches either use smooth continuation (Bertalmio, Sapiro *et al.* 2000) or borrow pixels from other parts of the image (Efros and Leung 1999; Criminisi, Pérez, and Toyama 2004; Efros and Freeman 2001). With the availability of huge numbers of images on the web, it often makes more sense to find a *different* image to serve as the source of the missing pixels.

In their system, Hays and Efros (2007) compute the *gist* of each image (Oliva and Torralba 2001; Torralba, Murphy *et al.* 2003) to find images with similar colors and composition. They then run a graph cut algorithm that minimizes image gradient differences and composite the new replacement piece into the original image using Poisson image blending (Section 8.4.4) (Pérez, Gangnet, and Blake 2003). Figure 6.40d shows the resulting image with the erased foreground rooftops region replaced with sailboats. Additional examples of photo editing and computational photography applications enabled by what has been dubbed “internet computer vision” can be found in the special journal issue edited by Avidan, Baker, and Shan (2010).

A different application of image recognition and segmentation is to infer 3D structure from a single photo by recognizing certain scene structures. For example, Criminisi, Reid,

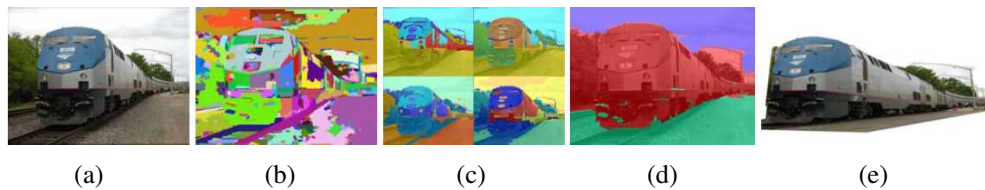


Figure 6.41 *Automatic photo pop-up (Hoiem, Efros, and Hebert 2005a) © 2005 ACM: (a) input image; (b) superpixels are grouped into (c) multiple regions; (d) labels indicating ground (green), vertical (red), and sky (blue); (e) novel view of resulting piecewise-planar 3D model.*

and Zisserman (2000) detect vanishing points and have the user draw basic structures, such as walls, to infer the 3D geometry (Section 11.1.2). Hoiem, Efros, and Hebert (2005a), on the other hand, work with more “organic” scenes such as the one shown in Figure 6.41. Their system uses a variety of classifiers and statistics learned from labeled images to classify each pixel as either ground, vertical, or sky (Figure 6.41d). To do this, they begin by computing superpixels (Figure 6.41b) and then group them into plausible regions that are likely to share similar geometric labels (Figure 6.41c). After all the pixels have been labeled, the boundaries between the vertical and ground pixels can be used to infer 3D lines along which the image can be folded into a “pop-up” (after removing the sky pixels), as shown in Figure 6.41e. In related work, Saxena, Sun, and Ng (2009) develop a system that directly infers the depth and orientation of each pixel instead of using just three geometric class labels. We will examine techniques to infer depth from single images in more detail in Section 12.8.

6.4.5 Pose estimation

The inference of human pose (head, body, and limb locations and attitude) from a single images can be viewed as yet another kind of segmentation task. We have already discussed some pose estimation techniques in Section 6.3.2 on pedestrian detection section, as shown in Figure 6.25. Starting with the seminal work by Felzenszwalb and Huttenlocher (2005), 2D and 3D pose detection and estimation rapidly developed as an active research area, with important advances and datasets (Sigal and Black 2006a; Rogez, Rihan *et al.* 2008; Andriluka, Roth, and Schiele 2009; Bourdev and Malik 2009; Johnson and Everingham 2011; Yang and Ramanan 2011; Pishchulin, Andriluka *et al.* 2013; Sapp and Taskar 2013; Andriluka, Pishchulin *et al.* 2014).

More recently, deep networks have become the preferred technique to identify human body keypoints in order to convert these into pose estimates (Tompson, Jain *et al.* 2014;

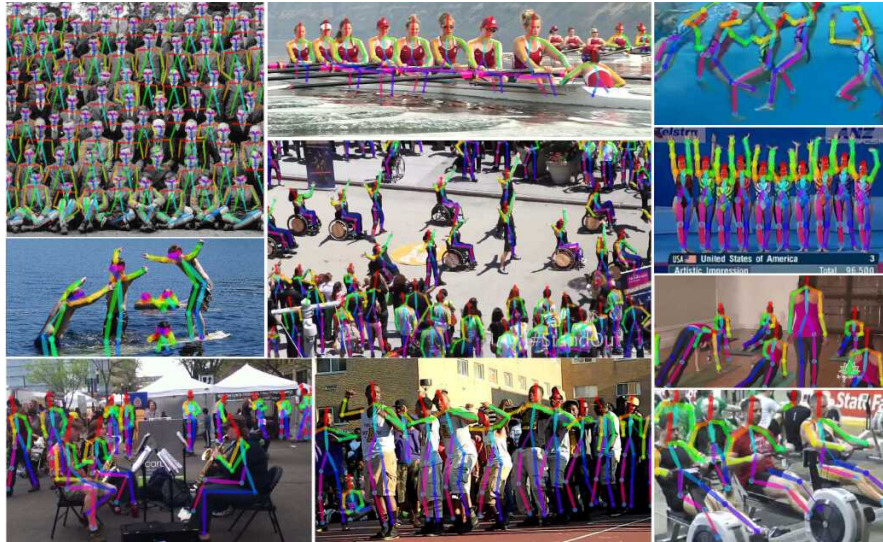


Figure 6.42 *OpenPose real-time multi-person 2D pose estimation (Cao, Simon et al. 2017) © 2017 IEEE.*

Toshev and Szegedy 2014; Pishchulin, Insafutdinov *et al.* 2016; Wei, Ramakrishna *et al.* 2016; Cao, Simon *et al.* 2017; He, Gkioxari *et al.* 2017; Hidalgo, Raaj *et al.* 2019; Huang, Zhu *et al.* 2020).²⁶ Figure 6.42 shows some of the impressive real-time multi-person 2D pose estimation results produced by the OpenPose system (Cao, Hidalgo *et al.* 2019).

The latest, most challenging, task in human pose estimation is the DensePose task introduced by Güler, Neverova, and Kokkinos (2018), where the task is to associate each pixel in RGB images of people with 3D points on a surface-based model, as shown in Figure 6.43. The authors provide dense annotations for 50,000 people appearing in COCO images and evaluate a number of correspondence networks, including their own DensePose-RCNN with several extensions. A more in-depth discussion on 3D human body modeling and tracking can be found in Section 13.6.4.

6.5 Video understanding

As we've seen in the previous sections of this chapter, image understanding mostly concerns itself with naming and delineating the objects and stuff in an image, although the relationships between objects and people are also sometimes inferred (Yao and Fei-Fei 2012; Gupta

²⁶You can find the leaderboards for human keypoint detection at <https://cocodataset.org>.

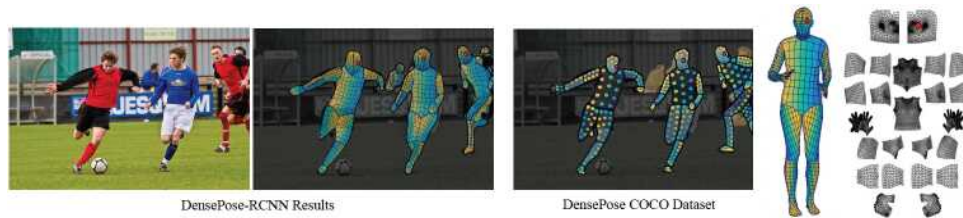


Figure 6.43 Dense pose estimation aims at mapping all human pixels of an RGB image to the 3D surface of the human body (Güler, Neverova, and Kokkinos 2018) © 2018 IEEE. The paper describes DensePose-COCO, a large-scale ground-truth dataset containing manually annotated image-to-surface correspondences for 50K persons and a DensePose-RCNN trained to densely regress UV coordinates at multiple frames per second.

and Malik 2015; Yatskar, Zettlemoyer, and Farhadi 2016; Gkioxari, Girshick *et al.* 2018). (We will look at the topic of describing complete images in the next section on vision and language.)

What, then, is video understanding? For many researchers, it starts with the detection and description of human actions, which are taken as the basic atomic units of videos. Of course, just as with images, these basic primitives can be chained into more complete descriptions of longer video sequences.

Human activity recognition began being studied in the 1990s, along with related topics such as human motion tracking, which we discuss in Sections 9.4.4 and 13.6.4. Aggarwal and Cai (1999) provide a comprehensive review of these two areas, which they call *human motion analysis*. Some of the techniques they survey use point and mesh tracking, as well as spatio-temporal signatures.

In the 2000s, attention shifted to spatio-temporal features, such as the clever use of optical flow in small patches to recognize sports activities (Efros, Berg *et al.* 2003) or spatio-temporal feature detectors for classifying actions in movies (Laptev, Marszalek *et al.* 2008), later combined with image context (Marszalek, Laptev, and Schmid 2009) and tracked feature trajectories (Wang and Schmid 2013). Poppe (2010), Aggarwal and Ryoo (2011), and Weinland, Ronfard, and Boyer (2011) provide surveys of algorithms from this decade. Some of the datasets used in this research include the KTH human motion dataset (Schüldt, Laptev, and Caputo 2004), the UCF sports action dataset (Rodriguez, Ahmed, and Shah 2008), the Hollywood human action dataset (Marszalek, Laptev, and Schmid 2009), UCF-101 (Soomro, Zamir, and Shah 2012), and the HMDB human motion database (Kuehne, Jhuang *et al.* 2011).

In the last decade, video understanding techniques have shifted to using deep networks

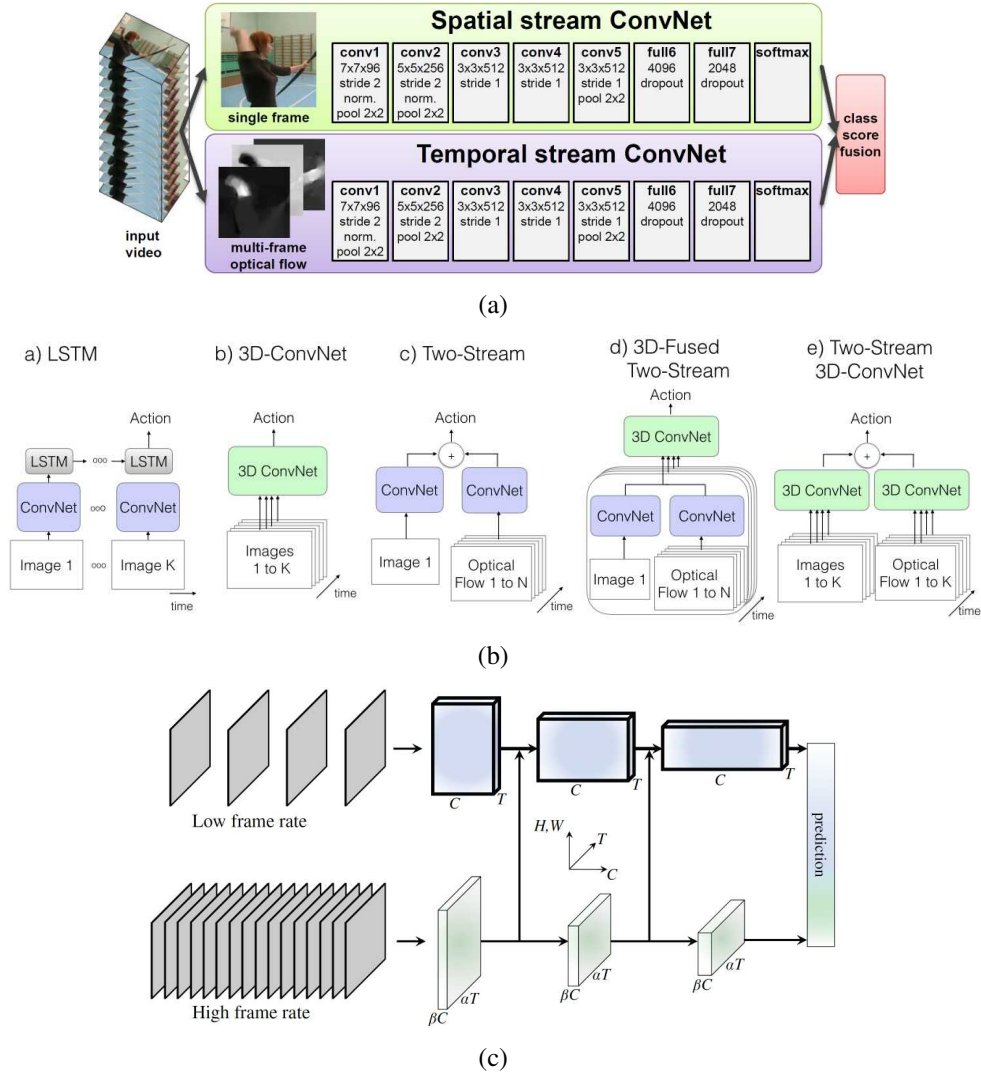


Figure 6.44 Video understanding using neural networks: (a) two-stream architecture for video classification © Simonyan and Zisserman (2014a); (b) some alternative video processing architectures (Carreira and Zisserman 2017) © 2017 IEEE; (c) a SlowFast network with a low frame rate, low temporal resolution Slow pathway and a high frame rate, higher temporal resolution Fast pathway (Feichtenhofer, Fan et al. 2019) © 2019 IEEE.

(Ji, Xu *et al.* 2013; Karpathy, Toderici *et al.* 2014; Simonyan and Zisserman 2014a; Tran, Bourdev *et al.* 2015; Feichtenhofer, Pinz, and Zisserman 2016; Carreira and Zisserman 2017; Varol, Laptev, and Schmid 2017; Wang, Xiong *et al.* 2019; Zhu, Li *et al.* 2020), sometimes combined with temporal models such as LSTMs (Baccouche, Mamalet *et al.* 2011; Donahue, Hendricks *et al.* 2015; Ng, Hausknecht *et al.* 2015; Srivastava, Mansimov, and Salakhudinov 2015).

While it is possible to apply these networks directly to the pixels in the video stream, e.g., using 3D convolutions (Section 5.5.1), researchers have also investigated using optical flow (Chapter 9.3) as an additional input. The resulting *two-stream architecture* was proposed by Simonyan and Zisserman (2014a) and is shown in Figure 6.44a. A later paper by Carreira and Zisserman (2017) compares this architecture to alternatives such as 3D convolutions on the pixel stream as well as hybrids of two streams and 3D convolutions (Figure 6.44b).

The latest architectures for video understanding have gone back to using 3D convolutions on the raw pixel stream (Tran, Wang *et al.* 2018, 2019; Kumawat, Verma *et al.* 2021). Wu, Feichtenhofer *et al.* (2019) store 3D CNN features into what they call a *long-term feature bank* to give a broader temporal context for action recognition. Feichtenhofer, Fan *et al.* (2019) propose a two-stream SlowFast architecture, where a slow pathway operates at a lower frame rate and is combined with features from a fast pathway with higher temporal sampling but fewer channels (Figure 6.44c). Some widely used datasets used for evaluating these algorithms are summarized in Table 6.3. They include Charades (Sigurdsson, Varol *et al.* 2016), YouTube8M (Abu-El-Haija, Kothari *et al.* 2016), Kinetics (Carreira and Zisserman 2017), “Something-something” (Goyal, Kahou *et al.* 2017), AVA (Gu, Sun *et al.* 2018), EPIC-KITCHENS (Damen, Doughty *et al.* 2018), and AVA-Kinetics (Li, Thotakuri *et al.* 2020). A nice exposition of these and other video understanding algorithms can be found in Johnson (2020, Lecture 18).

As with image recognition, researchers have also started using self-supervised algorithms to train video understanding systems. Unlike images, video clips are usually *multi-modal*, i.e., they contain audio tracks in addition to the pixels, which can be an excellent source of unlabeled supervisory signals (Alwassel, Mahajan *et al.* 2020; Patrick, Asano *et al.* 2020). When available at inference time, audio signals can improve the accuracy of such systems (Xiao, Lee *et al.* 2020).

Finally, while action recognition is the main focus of most recent video understanding work, it is also possible to classify videos into different scene categories such as “beach”, “fireworks”, or “snowing.” This problem is called *dynamic scene recognition* and can be addressed using spatio-temporal CNNs (Feichtenhofer, Pinz, and Wildes 2017).

Name/URL	Metadata	Contents/Reference
Charades https://prior.allenai.org/projects/charades	Actions, objects, descriptions	9.8k videos Sigurdsson, Varol <i>et al.</i> (2016)
YouTube8M https://research.google.com/youtube8m	Entities	4.8k visual entities, 8M videos Abu-El-Haija, Kothari <i>et al.</i> (2016)
Kinetics https://deepmind.com/research/open-source/kinetics	Action classes	700 action classes, 650k videos Carreira and Zisserman (2017)
“Something-something” https://20bn.com/datasets/something-something	Actions with objects	174 actions, 220k videos Goyal, Kahou <i>et al.</i> (2017)
AVA https://research.google.com/ava	Actions	80 actions in 430 15-minute videos Gu, Sun <i>et al.</i> (2018)
EPIC-KITCHENS https://epic-kitchens.github.io	Actions and objects	100 hours of egocentric videos Damen, Doughty <i>et al.</i> (2018)

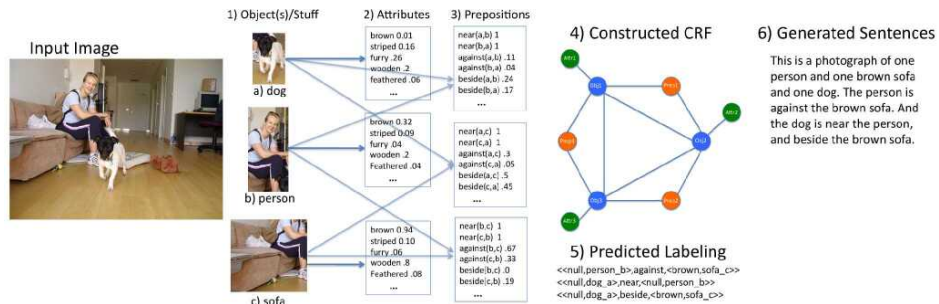
Table 6.3 Datasets for video understanding and action recognition.

6.6 Vision and language

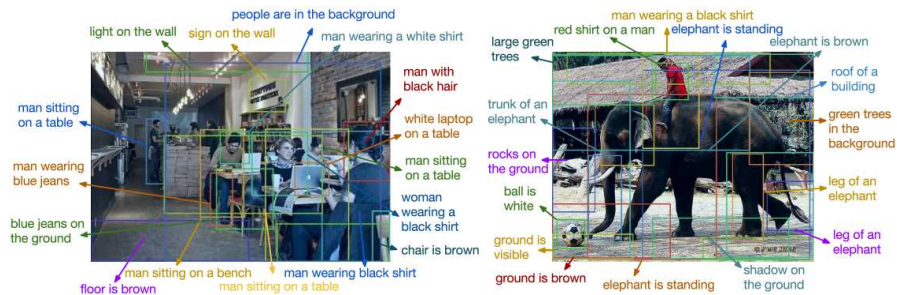
The ultimate goal of much of computer vision research is not just to solve simpler tasks such as building 3D models of the world or finding relevant images, but to become an essential component of *artificial general intelligence* (AGI). This requires vision to integrate with other components of artificial intelligence such as speech and language understanding and synthesis, logical inference, and commonsense and specialized knowledge representation and reasoning.

Advances in speech and language processing have enabled the widespread deployment of speech-based intelligent virtual assistants such as Siri, Google Assistant, and Alexa. Earlier in this chapter, we’ve seen how computer vision systems can name individual objects in images and find similar images by appearance or keywords. The next natural step of integration with other AI components is to merge vision and language, i.e., *natural language processing* (NLP).

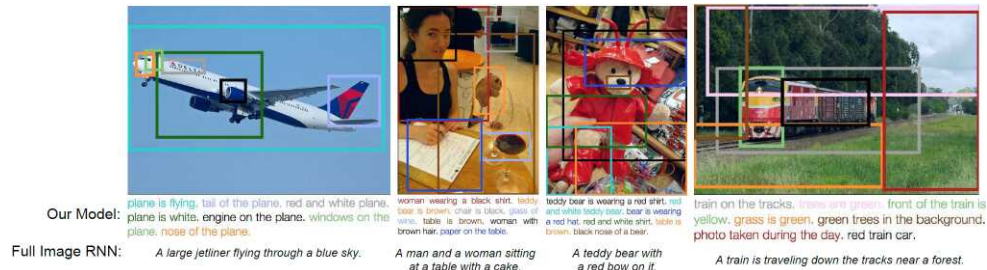
While this area has been studied for a long time (Duygulu, Barnard *et al.* 2002; Farhadi, Hejrati *et al.* 2010), the last decade has seen a rapid increase in performance and capabilities (Mogadala, Kalimuthu, and Klakow 2021; Gan, Yu *et al.* 2020). An example of this is the BabyTalk system developed by Kulkarni, Premraj *et al.* (2013), which first detects objects, their attributes, and their positional relationships, then infers a likely compatible labeling of these objects, and finally generates an image caption, as shown in Figure 6.45a.



(a)



(b)



(c)

Figure 6.45 Image captioning systems: (a) BabyTalk detects objects, attributes, and positional relationships and composes these into image captions (Kulkarni, Premraj et al. 2013) © 2013 IEEE; (b–c) DenseCap associates word phrases with regions and then uses an RNN to construct plausible sentences (Johnson, Karpathy, and Fei-Fei 2016) © 2016 IEEE.

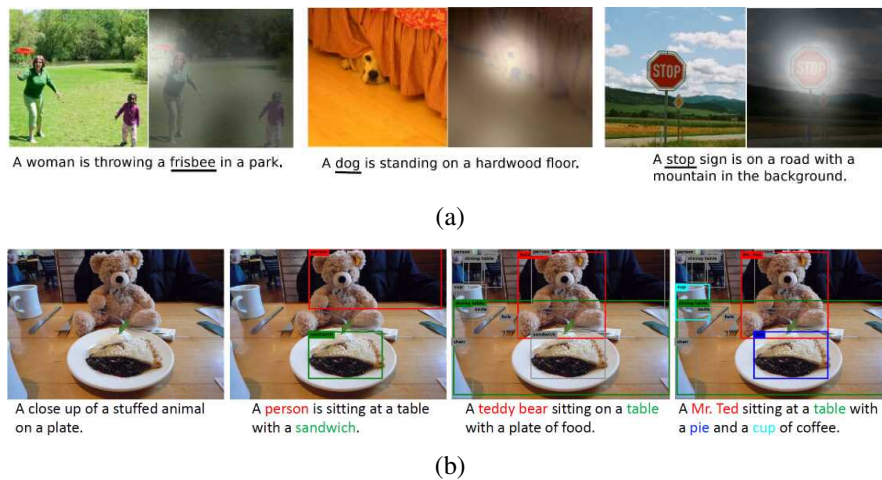


Figure 6.46 Image captioning with attention: (a) The “Show, Attend, and Tell” system, which uses hard attention to align generated words with image regions © Xu, Ba et al. (2015); (b) Neural Baby Talk captions generated using different detectors, showing the association between words and grounding regions (Lu, Yang et al. 2018) © 2018 IEEE.

Visual captioning

The next few years brought a veritable explosion of papers on the topic of image captioning and description, including (Chen and Lawrence Zitnick 2015; Donahue, Hendricks *et al.* 2015; Fang, Gupta *et al.* 2015; Karpathy and Fei-Fei 2015; Vinyals, Toshev *et al.* 2015; Xu, Ba *et al.* 2015; Johnson, Karpathy, and Fei-Fei 2016; Yang, He *et al.* 2016; You, Jin *et al.* 2016). Many of these systems combine CNN-based image understanding components (mostly object and human action detectors) with RNNs or LSTMs to generate the description, often in conjunction with other techniques such as multiple instance learning, maximum entropy language models, and visual attention. One somewhat surprising early result was that nearest-neighbor techniques, i.e., finding sets of similar looking images with captions and then creating a consensus caption, work surprisingly well (Devlin, Gupta *et al.* 2015).

Over the last few years, attention-based systems have continued to be essential components of image captioning systems (Lu, Xiong *et al.* 2017; Anderson, He *et al.* 2018; Lu, Yang *et al.* 2018). Figure 6.46 shows examples from two such papers, where each word in the generated caption is *grounded* with a corresponding image region. The CVPR 2020 tutorial by (Zhou 2020) summarizes over two dozen related papers from the last five years, including papers that use transformers (Section 5.5.3) to do the captioning. It also covers video descrip-

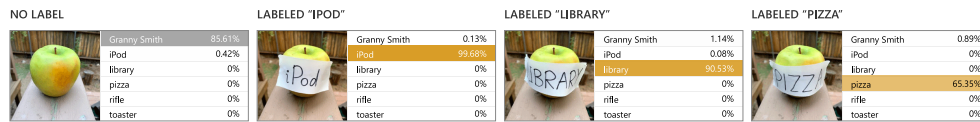


Figure 6.47 An adversarial typographic attack used against CLIP (Radford, Kim *et al.* 2021) discovered by ©Goh, Cammarata *et al.* (2021). Instead of predicting the object that exists in the scene, CLIP predicts the output based on the adversarial handwritten label.

tion and dense video captioning (Aafaq, Mian *et al.* 2019; Zhou, Kalantidis *et al.* 2019) and vision-language pre-training (Sun, Myers *et al.* 2019; Zhou, Palangi *et al.* 2020; Li, Yin *et al.* 2020). The tutorial also has lectures on visual question answering and reasoning (Gan 2020), text-to-image synthesis (Cheng 2020), and vision-language pre-training (Yu, Chen, and Li 2020).

For the task of image classification (Section 6.2), one of the major restrictions is that a model can only predict a label from the discrete pre-defined set of labels it trained on. CLIP (Radford, Kim *et al.* 2021) proposes an alternative approach that relies on image captions to enable zero-shot transfer to any possible set of labels. Given an image with a set of labels (e.g., {dog, cat, . . . , house}), CLIP predicts the label that maximizes the probability that the image is captioned with a prompt similar to “A photo of a {label}”. Section 5.4.7 discusses the training aspect of CLIP, which collects 400 million text-image pairs and uses contrastive learning to determine how likely it is for an image to be paired with a caption.

Remarkably, without having seen or fine-tuned to many popular image classification benchmarks (e.g., ImageNet, Caltech 101), CLIP can outperform independently fine-tuned ResNet-50 models supervised on each specific dataset. Moreover, compared to state-of-the-art classification models, CLIP’s zero-shot generalization is significantly more robust to dataset distribution shifts, performing well on each of ImageNet Sketch (Wang, Ge *et al.* 2019), ImageNetV2 (Recht, Roelofs *et al.* 2019), and ImageNet-R (Hendrycks, Basart *et al.* 2020), without being specifically trained on any of them. In fact, Goh, Cammarata *et al.* (2021) found that CLIP units responded similarly with concepts presented in different modalities (e.g., an image of Spiderman, text of the word spider, and a drawing of Spiderman). Figure 6.47 shows the adversarial typographic attack they discovered that could fool CLIP. By simply placing a handwritten class label (e.g., iPod) on a real-world object (e.g., Apple), CLIP often predicted the class written on the label.

As with other areas of visual recognition and learning-based systems, datasets have played an important role in the development of vision and language systems. Some widely used datasets of images with captions include Conceptual Captions (Sharma, Ding *et al.* 2018),



Figure 6.48 Images and data from the Visual Genome dataset (Krishna, Zhu et al. 2017) © 2017 Springer. (a) An example image with its region descriptors. (b) Each region has a graph representation of objects, attributes, and pairwise relationships, which are combined into a scene graph where all the objects are grounded to the image, and also associated questions and answers. (c) Some sample question and answer pairs, which cover a spectrum of visual tasks from recognition to high-level reasoning.

Name/URL	Metadata	Contents/Reference
Flickr30k (Entities)	Image captions (grounded)	30k images (+ bounding boxes)
https://shannon.cs.illinois.edu/DenotationGraph		Young, Lai <i>et al.</i> (2014)
http://bryanplummer.com/Flickr30kEntities		Plummer, Wang <i>et al.</i> (2017)
COCO Captions	Whole image captions	1.5M captions, 330k images
https://cocodataset.org/#captions-2015		Chen, Fang <i>et al.</i> (2015)
Conceptual Captions	Whole image captions	3.3M image caption pairs
https://ai.google.com/research/ConceptualCaptions		Sharma, Ding <i>et al.</i> (2018)
YFCC100M	Flickr metadata	100M images with metadata
http://projects.dfki.uni-kl.de/yfcc100m		Thomee, Shamma <i>et al.</i> (2016)
Visual Genome	Dense annotations	108k images with region graphs
https://visualgenome.org		Krishna, Zhu <i>et al.</i> (2017)
VQA v2.0	Question/answer pairs	265k images
https://visualqa.org		Goyal, Khot <i>et al.</i> (2017)
VCR	Multiple choice questions	110k movie clips, 290k QAs
https://visualcommonsense.com		Zellers, Bisk <i>et al.</i> (2019)
GQA	Compositional QA	22M questions on Visual Genome
https://visualreasoning.net		Hudson and Manning (2019)
VisDial	Dialogs for chatbot	120k COCO images + dialogs
https://visualdialog.org		Das, Kottur <i>et al.</i> (2017)

Table 6.4 *Image datasets for vision and language research.*

the UIUC Pascal Sentence Dataset (Farhadi, Hejrati *et al.* 2010), the SBU Captioned Photo Dataset (Ordonez, Kulkarni, and Berg 2011), Flickr30k (Young, Lai *et al.* 2014), COCO Captions (Chen, Fang *et al.* 2015), and their extensions to 50 sentences per image (Vedantam, Lawrence Zitnick, and Parikh 2015) (see Table 6.4). More densely annotated datasets such as Visual Genome (Krishna, Zhu *et al.* 2017) describe different sub-regions of an image with their own phrases, i.e., provide *dense captioning*, as shown in Figure 6.48. YFCC100M (Thomee, Shamma *et al.* 2016) contains around 100M images from Flickr, but it only includes the raw user uploaded metadata for each image, such as the title, time of upload, description, tags, and (optionally) the location of the image.

Metrics for measuring sentence similarity also play an important role in the development of image captioning and other vision and language systems. Some widely used metrics include BLEU: BiLingual Evaluation Understudy (Papineni, Roukos *et al.* 2002), ROUGE: Recall Oriented Understudy of Gisting Evaluation (Lin 2004), METEOR: Metric for Evaluation of Translation with Explicit ORdering (Banerjee and Lavie 2005), CIDEr: Consensus-based

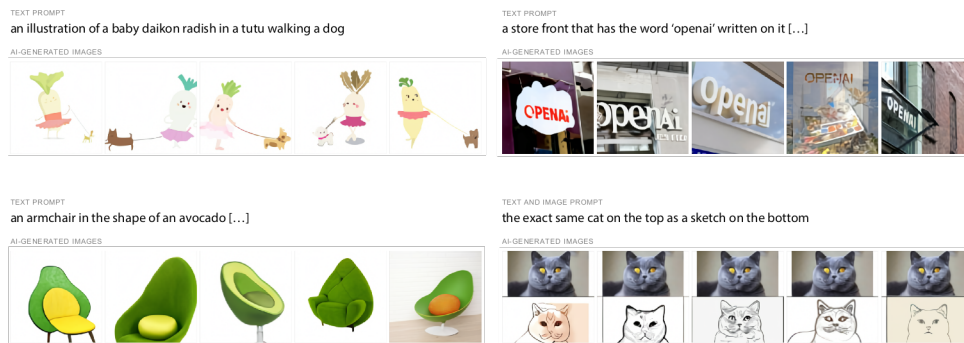


Figure 6.49 *Qualitative text-to-image generation results from DALL-E, showing a wide range of generalization abilities ©Ramesh, Pavlov et al. (2021). The bottom right example provides a partially complete image prompt of a cat, along with text, and has the model fill in the rest of the image. The other three examples only start with the text prompt as input, with the model generating the entire image.*

Image Description Evaluation (Vedantam, Lawrence Zitnick, and Parikh 2015), and SPICE: Semantic Propositional Image Caption Evaluation (Anderson, Fernando et al. 2016).²⁷

Text-to-image generation

The task of text-to-image generation is the inverse of visual captioning, i.e., given a text prompt, generate the image. Since images are represented in such high dimensionality, generating them to look coherent has historically been difficult. Generating images from a text prompt can be thought of as a generalization of generating images from a small set of class labels (Section 5.5.4). Since there is a near-infinite number of possible text prompts, successful models must be able to generalize from the relatively small fraction seen during training.

Early work on this task from Mansimov, Parisotto et al. (2016) used an RNN to iteratively draw an image from scratch. Their results showed some resemblance to the text prompts, although the generated images were quite blurred. The following year, Reed, Akata et al. (2016) applied a GAN to the problem, where unseen text prompts began to show promising results. Their generated images were relatively small (64×64), which was improved in later papers, which often first generated a small-scale image and then conditioned on that image and the text input to generate a higher-resolution image (Zhang, Xu et al. 2017, 2018; Xu, Zhang et al. 2018; Li, Qi et al. 2019).

DALL-E (Ramesh, Pavlov et al. 2021) uses orders of magnitude of more data (250 million

²⁷See https://www.cs.toronto.edu/~fidler/slides/2017/CSC2539/Kaustav_slides.pdf.

text-image pairs on the internet) and compute to achieve astonishing qualitative results (Figure 6.49).²⁸ Their approach produces promising results for generalizing beyond training data, even compositionally piecing together objects that are not often related (e.g., an armchair and an avocado), producing many styles (e.g., painting, cartoon, charcoal drawings), and working reasonably well with difficult objects (e.g., mirrors or text).

The model for DALL·E consists of two components: a VQ-VAE-2 (Section 5.5.4) and a decoder transformer (Section 5.5.3). The text is tokenized into 256 tokens, each of which is one of 16,384 possible vectors using a BPE-encoding (Sennrich, Haddow, and Birch 2015). The VQ-VAE-2 uses a codebook of size 8,192 (significantly larger than the codebook of size 512 used in the original VQ-VAE-2 paper) to compress images as a 32×32 grid of vector tokens. At inference time, DALL·E uses a transformer decoder, which starts with the 256 text tokens to autoregressively predict the 32×32 grid of image tokens. Given such a grid, the VQ-VAE-2 is able to use its decoder to generate the final RGB image of size 256×256 . To achieve better empirical results, DALL·E generates 512 image candidates and reranks them using CLIP (Radford, Kim *et al.* 2021), which determines how likely a given caption is associated with a given image.

An intriguing extension of DALL·E is to use the VQ-VAE-2 encoder to predict a subset of the compressed image tokens. For instance, suppose we are given a text input and an image. The text input can be tokenized into its 256 tokens, and one can obtain the 32×32 image tokens using the VQ-VAE-2 encoder. If we then discard the bottom half of the image tokens, the transformer decoder can be used to autoregressively predict which tokens might be there. These tokens, along with the non-discarded ones from the original image, can be passed into the VQ-VAE-2 decoder to produce a completed image. Figure 6.49 (bottom right) shows how such a text and partial image prompt can be used for applications such as image-to-image translation (Section 5.5.4).

Visual Question Answering and Reasoning

Image and video captioning are useful tasks that bring us closer to building artificially intelligent systems, as they demonstrate the ability to put together visual cues such as object identities, attributes, and actions. However, it remains unclear if the system has understood the scene at a deeper level and if it can reason about the constituent pieces and how they fit together.

To address these concerns, researchers have been building *visual question answering* (VQA) systems, which require the vision algorithm to answer open-ended questions about

²⁸Play with the results at <https://openai.com/blog/dall-e>.

the image, such as the ones shown in Figure 6.48c. A lot of this work started with the creation of the Visual Question Answering (VQA) dataset (Antol, Agrawal *et al.* 2015), which spurred a large amount of subsequent research. The following year, VQA v2.0 improved this dataset by creating a *balanced* set of image pairs, where each question had different answers in the two images (Goyal, Khot *et al.* 2017).²⁹ This dataset was further extended to reduce the influence of prior assumptions and data distributions and to encourage answers to be grounded in the images (Agrawal, Batra *et al.* 2018).

Since then, many additional VQA datasets have been created. These include the VCR dataset for visual commonsense reasoning (Zellers, Bisk *et al.* 2019) and the GQA dataset and metrics for evaluating visual reasoning and compositional question answering (Hudson and Manning 2019), which is built on top of the information about objects, attributes, and relations provided through the Visual Genome scene graphs (Krishna, Zhu *et al.* 2017). A discussion of these and other datasets for VQA can be found in the CVPR 2020 tutorial by Gan (2020), including datasets that test visual grounding and referring expression comprehension, visual entailment, using external knowledge, reading text, answering sub-questions, and using logic. Some of these datasets are summarized in Table 6.4.

As with image and video captioning, VQA systems use various flavors of attention to associate pixel regions with semantic concepts (Yang, He *et al.* 2016). However, instead of using sequence models such as RNNs, LSTMs, or transformers to generate text, the natural language question is first parsed to produce an encoding that is then fused with the image embedding to generate the desired answer.

The image semantic features can either be computed on a coarse grid, or a “bottom-up” object detector can be combined with a “top-down” attention mechanism to provide feature weightings (Anderson, He *et al.* 2018). In recent years, the pendulum has swung back and forth between techniques that use bottom-up regions and gridded feature descriptors, with two of the recent best-performing algorithms going back to the simpler (and much faster) gridded approach (Jiang, Misra *et al.* 2020; Huang, Zeng *et al.* 2020). The CVPR 2020 tutorial by Gan (2020) discusses these and dozens of other VQA systems as well as their subcomponents, such as multimodal fusion variants (bilinear pooling, alignment, relational reasoning), neural module networks, robust VQA, and multimodal pre-training. The survey by Mogadala, Kalimuthu, and Klakow (2021) and the annual VQA Challenge workshop (Shrivastava, Hudson *et al.* 2020) are also excellent sources of additional information. And if you would like to test out the current state of VQA systems, you can upload your own image to <https://vqa.cloudcv.org> and ask the system your own questions.

²⁹<https://visualqa.org>

Visual Dialog. An even more challenging version of VQA is *visual dialog*, where a chatbot is given an image and asked to answer open-ended questions about the image while also referring to previous elements of the conversation. The VisDial dataset was the earliest to be widely used for this task (Das, Kottur *et al.* 2017).³⁰ You can find pointers to systems that have been developed for this task at the Visual Dialog workshop and challenge (Shrivastava, Hudson *et al.* 2020). There’s also a chatbot at <https://visualchatbot.cloudcv.org> where you can upload your own image and start a conversation, which can sometimes lead to humorous (or weird) outcomes (Shane 2019).

Vision-language pre-training. As with many other recognition tasks, pre-training has had some dramatic success in the last few years, with systems such as ViLBERT (Lu, Batra *et al.* 2019), Oscar (Li, Yin *et al.* 2020), and many other systems described in the CVPR 2020 tutorial on self-supervised learning for vision-and-language (Yu, Chen, and Li 2020).

6.7 Additional reading

Unlike machine learning or deep learning, there are no recent textbooks or surveys devoted specifically to the general topics of image recognition and scene understanding. Some earlier surveys (Pinz 2005; Andreopoulos and Tsotsos 2013) and collections of papers (Ponce, Hebert *et al.* 2006; Dickinson, Leonardis *et al.* 2007) review the “classic” (pre-deep learning) approaches, but given the tremendous changes in the last decade, many of these techniques are no longer used. Currently, some of the best sources for the latest material, in addition to this chapter and university computer vision courses, are tutorials at the major vision conferences such as ICCV (Xie, Girshick *et al.* 2019), CVPR (Girshick, Kirillov *et al.* 2020), and ECCV (Xie, Girshick *et al.* 2020). Image recognition datasets such as those listed in Tables 6.1–6.4 that maintain active leaderboards can also be a good source for recent papers.

Algorithms for instance recognition, i.e., the detection of static manufactured objects that only vary slightly in appearance but may vary in 3D pose, are still often based on detecting 2D points of interest and describing them using viewpoint-invariant descriptors, as discussed in Chapter 7 and (Lowe 2004), Rothganger, Lazebnik *et al.* (2006), and Gordon and Lowe (2006). In more recent years, attention has shifted to the more challenging problem of *instance retrieval* (also known as *content-based image retrieval*), in which the number of images being searched can be very large (Sivic and Zisserman 2009). Section 7.1.4 in the next chapter reviews such techniques, as does the survey in (Zheng, Yang, and Tian 2018). This topic is also related to visual similarity search (Bell and Bala 2015; Arandjelovic, Gronat *et*

³⁰<https://visualdialog.org>

al. 2016; Song, Xiang *et al.* 2016; Gordo, Almazán *et al.* 2017; Rawat and Wang 2017; Bell, Liu *et al.* 2020), which was covered in Section 6.2.3.

A number of surveys, collections of papers, and course notes have been written on the topic of feature-based whole image (single-object) category recognition (Pinz 2005; Ponce, Hebert *et al.* 2006; Dickinson, Leonardis *et al.* 2007; Fei-Fei, Fergus, and Torralba 2009). Some of these papers use a bag of words or keypoints (Csurka, Dance *et al.* 2004; Lazebnik, Schmid, and Ponce 2006; Csurka, Dance *et al.* 2006; Grauman and Darrell 2007b; Zhang, Marszalek *et al.* 2007; Boiman, Shechtman, and Irani 2008; Ferencz, Learned-Miller, and Malik 2008). Other papers recognize objects based on their contours, e.g., using shape contexts (Belongie, Malik, and Puzicha 2002) or other techniques (Shotton, Blake, and Cipolla 2005; Opelt, Pinz, and Zisserman 2006; Ferrari, Tuytelaars, and Van Gool 2006a).

Many object recognition algorithms use part-based decompositions to provide greater invariance to articulation and pose. Early algorithms focused on the relative positions of the parts (Fischler and Elschlager 1973; Kanade 1977; Yuille 1991) while later algorithms used more sophisticated models of appearance (Felzenszwalb and Huttenlocher 2005; Fergus, Perona, and Zisserman 2007; Felzenszwalb, McAllester, and Ramanan 2008). Good overviews on part-based models for recognition can be found in the course notes by Fergus (2009). Carneiro and Lowe (2006) discuss a number of graphical models used for part-based recognition, which include trees and stars, k -fans, and constellations.

Classical recognition algorithms often used scene context as part of their recognition strategy. Representative papers in this area include Torralba (2003), Torralba, Murphy *et al.* (2003), Rabinovich, Vedaldi *et al.* (2007), Russell, Torralba *et al.* (2007), Sudderth, Torralba *et al.* (2008), and Divvala, Hoiem *et al.* (2009). Machine learning also became a key component of classical object detection and recognition algorithms (Felzenszwalb, McAllester, and Ramanan 2008; Sivic, Russell *et al.* 2008), as did exploiting large human-labeled databases (Russell, Torralba *et al.* 2007; Torralba, Freeman, and Fergus 2008).

The breakthrough success of the “AlexNet” SuperVision system of Krizhevsky, Sutskever, and Hinton (2012) shifted the focus in category recognition research from feature-based approaches to deep neural networks. The rapid improvement in recognition accuracy, captured in Figure 5.40 and described in more detail in Section 5.4.3 has been driven to a large degree by deeper networks and better training algorithms, and also in part by larger (unlabeled) training datasets (Section 5.4.7).

More specialized recognition systems such as those for recognizing faces underwent a similar evolution. While some of the earliest approaches to face recognition involved finding the distinctive image features and measuring the distances between them (Fischler and Elschlager 1973; Kanade 1977; Yuille 1991), later approaches relied on comparing gray-

level images, often projected onto lower dimensional subspaces (Turk and Pentland 1991; Belhumeur, Hespanha, and Kriegman 1997; Heisele, Ho *et al.* 2003) or local binary patterns (Ahonen, Hadid, and Pietikäinen 2006). A variety of shape and pose deformation models were also developed (Beymer 1996; Vetter and Poggio 1997), including Active Shape Models (Cootes, Cooper *et al.* 1995), 3D Morphable Models (Blanz and Vetter 1999; Egger, Smith *et al.* 2020), and Active Appearance Models (Cootes, Edwards, and Taylor 2001; Matthews and Baker 2004; Ramnath, Koterba *et al.* 2008). Additional information about classic face recognition algorithms can be found in a number of surveys and books on this topic (Chellappa, Wilson, and Sirohey 1995; Zhao, Chellappa *et al.* 2003; Li and Jain 2005).

The concept of shape models for *frontalization* continued to be used as the community shifted to deep neural network approaches (Taigman, Yang *et al.* 2014). Some more recent deep face recognizers, however, omit the frontalization stage and instead use data augmentation to create synthetic inputs with a larger variety of poses (Schroff, Kalenichenko, and Philbin 2015; Parkhi, Vedaldi, and Zisserman 2015). Masi, Wu *et al.* (2018) provide an excellent tutorial and survey on deep face recognition, including a list of widely used training and testing datasets, a discussion of frontalization and dataset augmentation, and a section on training losses.

As the problem of whole-image (single object) category recognition became more “solved”, attention shifted to multiple object delineation and labeling, i.e., object detection. Object detection was originally studied in the context of specific categories such as faces, pedestrians, cars, etc. Seminal papers in face detection include those by Osuna, Freund, and Girosi (1997); Sung and Poggio (1998); Rowley, Baluja, and Kanade (1998); Viola and Jones (2004); Heisele, Ho *et al.* (2003), with Yang, Kriegman, and Ahuja (2002) providing a comprehensive survey of early work in this field. Early work in pedestrian and car detection was carried out by Gavrila and Philomin (1999); Gavrila (1999); Papageorgiou and Poggio (2000); Schneiderman and Kanade (2004). Subsequent papers include (Mikolajczyk, Schmid, and Zisserman 2004; Dalal and Triggs 2005; Leibe, Seemann, and Schiele 2005; Andriluka, Roth, and Schiele 2009; Dollár, Belongie, and Perona 2010; Felzenszwalb, Girshick *et al.* 2010).

Modern generic object detectors are typically constructed using a region proposal algorithm (Uijlings, Van De Sande *et al.* 2013; Zitnick and Dollár 2014) that then feeds selected regions of the image (either as pixels or precomputed neural features) into a multi-way classifier, resulting in architectures such as R-CNN (Girshick, Donahue *et al.* 2014), Fast R-CNN (Girshick 2015), Faster R-CNN (Ren, He *et al.* 2015), and FPN (Lin, Dollár *et al.* 2017). An alternative to this two-stage approach is a *single-stage network*, which uses a single network to output detections at a variety of locations. Examples of such architectures include

SSD (Liu, Anguelov *et al.* 2016), RetinaNet (Lin, Goyal *et al.* 2017), and YOLO (Redmon, Divvala *et al.* 2016; Redmon and Farhadi 2017, 2018; Bochkovskiy, Wang, and Liao 2020). These and more recent convolutional object detectors are described in the recent survey by Jiao, Zhang *et al.* (2019).

While object detection can be sufficient in many computer vision applications such as counting cars or pedestrians or even describing images, a detailed pixel-accurate labeling can be potentially even more useful, e.g., for photo editing. This kind of labeling comes in several flavors, including semantic segmentation (what stuff is this?), instance segmentation (which countable object is this?), panoptic segmentation (what stuff or object is it?). One early approach to this problem was to pre-segment the image into pieces and then match these pieces to portions of the model (Mori, Ren *et al.* 2004; Russell, Efros *et al.* 2006; Borenstein and Ullman 2008; Gu, Lim *et al.* 2009). Another popular approach was to use conditional random fields (Kumar and Hebert 2006; He, Zemel, and Carreira-Perpiñán 2004; Winn and Shotton 2006; Rabinovich, Vedaldi *et al.* 2007; Shotton, Winn *et al.* 2009). which at that time produced some of the best results on the PASCAL VOC segmentation challenge. Modern semantic segmentation algorithms use pyramidal fully-convolutional architectures to map input pixels to class labels (Long, Shelhamer, and Darrell 2015; Zhao, Shi *et al.* 2017; Xiao, Liu *et al.* 2018; Wang, Sun *et al.* 2020).

The more challenging task of instance segmentation, where each distinct object gets its own unique label, is usually tackled using a combination of object detectors and per-object segmentation, as exemplified in the seminal Mask R-CNN paper by He, Gkioxari *et al.* (2017). Follow-on work uses more sophisticated backbone architectures (Liu, Qi *et al.* 2018; Chen, Pang *et al.* 2019). Two recent workshops that highlight the latest results in this area are the COCO + LVIS Joint Recognition Challenge (Kirillov, Lin *et al.* 2020) and the Robust Vision Challenge (Zendel *et al.* 2020).

Putting semantic and instance segmentation together has long been a goal of semantic scene understanding (Yao, Fidler, and Urtasun 2012; Tighe and Lazechnik 2013; Tu, Chen *et al.* 2005). Doing this on a per-pixel level results in a *panoptic segmentation* of the scene, where all of the objects are correctly segmented and the remaining stuff is correctly labeled (Kirillov, He *et al.* 2019; Kirillov, Girshick *et al.* 2019). The COCO dataset has now been extended to include a panoptic segmentation task, on which some recent results can be found in the ECCV 2020 workshop on this topic (Kirillov, Lin *et al.* 2020).

Research in video understanding, or more specifically human activity recognition, dates back to the 1990s; some good surveys include (Aggarwal and Cai 1999; Poppe 2010; Aggarwal and Ryoo 2011; Weinland, Ronfard, and Boyer 2011). In the last decade, video understanding techniques shifted to using deep networks (Ji, Xu *et al.* 2013; Karpathy, Toderici *et*

et al. 2014; Simonyan and Zisserman 2014a; Donahue, Hendricks *et al.* 2015; Tran, Bourdev *et al.* 2015; Feichtenhofer, Pinz, and Zisserman 2016; Carreira and Zisserman 2017; Tran, Wang *et al.* 2019; Wu, Feichtenhofer *et al.* 2019; Feichtenhofer, Fan *et al.* 2019). Some widely used datasets used for evaluating these algorithms are summarized in Table 6.3.

While associating words with images has been studied for a while (Duygulu, Barnard *et al.* 2002), sustained research into describing images with captions and complete sentences started in the early 2010s (Farhadi, Hejrati *et al.* 2010; Kulkarni, Premraj *et al.* 2013). The last decade has seen a rapid increase in performance and capabilities of such systems (Mogadala, Kalimuthu, and Klakow 2021; Gan, Yu *et al.* 2020). The first sub-problem to be widely studied was image captioning (Donahue, Hendricks *et al.* 2015; Fang, Gupta *et al.* 2015; Karpathy and Fei-Fei 2015; Vinyals, Toshev *et al.* 2015; Xu, Ba *et al.* 2015; Devlin, Gupta *et al.* 2015), with later systems using attention mechanisms (Anderson, He *et al.* 2018; Lu, Yang *et al.* 2018). More recently, researchers have developed systems for visual question answering (Antol, Agrawal *et al.* 2015) and visual commonsense reasoning (Zellers, Bisk *et al.* 2019).

The CVPR 2020 tutorial on recent advances in visual captioning (Zhou 2020) summarizes over two dozen related papers from the last five years, including papers that use Transformers to do the captioning. It also covers video description and dense video captioning (Aafaq, Mian *et al.* 2019; Zhou, Kalantidis *et al.* 2019) and vision-language pre-training (Sun, Myers *et al.* 2019; Zhou, Palangi *et al.* 2020; Li, Yin *et al.* 2020). The tutorial also has lectures on visual question answering and reasoning (Gan 2020), text-to-image synthesis (Cheng 2020), and vision-language pre-training (Yu, Chen, and Li 2020).

6.8 Exercises

Ex 6.1: Pre-trained recognition networks. Find a pre-trained network for image classification, segmentation, or some other task such as face recognition or pedestrian detection.

After running the network, can you characterize the most common kinds of errors the network is making? Create a “confusion matrix” indicating which categories get classified as other categories. Now try the network on your own data, either from a web search or from your personal photo collection. Are there surprising results?

My own favorite code to try is Detectron2,³¹ which I used to generate the panoptic segmentation results shown in Figure 6.39.

³¹Click on the “Colab Notebook” link at <https://github.com/facebookresearch/detectron2> and then edit the input image URL to try your own.

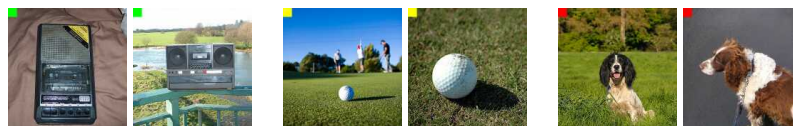
Ex 6.2: Re-training recognition networks. After analyzing the performance of your pre-trained network, try re-training it on the original dataset on which it was trained, but with modified parameters (numbers of layers, channels, training parameters) or with additional examples. Can you get the network to perform more to your liking?

Many of the online tutorials, such as the Detectron2 Collab notebook mentioned above, come with instructions on how to re-train the network from scratch on a different dataset. Can you create your own dataset, e.g., using a web search and figure out how to label the examples? A low effort (but not very accurate) way is to trust the results of the web search. Russakovsky, Deng *et al.* (2015), Kovashka, Russakovsky *et al.* (2016), and other papers on image datasets discuss the challenges in obtaining accurate labels.

Train your network, try to optimize its architecture, and report on the challenges you faced and discoveries you made.

Note: the following exercises were suggested by Matt Deitke.

Ex 6.3: Image perturbations. Download either ImageNet or Imagenette.³² Now, perturb each image by adding a small square to the top left of the image, where the color of the square is unique for each label, as shown in the following figure:



(a) cassette player

(b) golf ball

(c) English Springer

Using any image classification model,³³ e.g., ResNet, EfficientNet, or ViT, train the model from scratch on the perturbed images. Does the model overfit to the color of the square and ignore the rest of the image? When evaluating the model on the training and validation data, try adversarially swapping colors between different labels.

Ex 6.4: Image normalization. Using the same dataset downloaded for the previous exercise, take a ViT model and remove all the intermediate layer normalization operations. Are you able to train the network? Using techniques in Li, Xu *et al.* (2018), how do the plots of the loss landscape appear with and without the intermediate layer normalization operations?

Ex 6.5: Semantic segmentation. Explain the differences between instance segmentation, semantic segmentation, and panoptic segmentation. For each type of segmentation, can it be post-processed to obtain the other kinds of segmentation?

³²Imagenette, <https://github.com/fastai/imagenette>, is a smaller 10-class subset of ImageNet that is easier to use with limited computing resources. .

³³You may find the PyTorch Image Models at <https://github.com/rwightman/pytorch-image-models> useful.

Ex 6.6: Class encoding. Categorical inputs to a neural network, such as a word or object, can be encoded with one-hot encoded vector.³⁴ However, it is common to pass the one-hot encoded vector through an embedding matrix, where the output is then passed into the neural network loss function. What are the advantages of vector embedding over using one-hot encoding?

Ex 6.7: Object detection. For object detection, how do the number of parameters for DETR, Faster-RCNN, and YOLOv4 compare? Try training each of them on MS COCO. Which one tends to train the slowest? How long does it take each model to evaluate a single image at inference time?

Ex 6.8: Image classification vs. description. For image classification, list at least two significant differences between using categorical labels and natural language descriptions.

Ex 6.9: ImageNet Sketch. Try taking several pre-trained models on ImageNet and evaluating them, without any fine-tuning, on ImageNet Sketch (Wang, Ge *et al.* 2019). For each of these models, to what extent does the performance drop due to the shift in distribution?

Ex 6.10: Self-supervised learning. Provide examples of self-supervised learning pretext tasks for each of the following data types: static images, videos, and vision-and-language.

Ex 6.11: Video understanding. For many video understanding tasks, we may be interested in tracking an object through time. Why might this be preferred to making predictions independently for each frame? Assume that inference speed is not a problem.

Ex 6.12: Fine-tuning a new head. Take the backbone of a network trained for object classification and fine-tune it for object detection with a variant of YOLO. Why might it be desirable to freeze the early layers of the network?

Ex 6.13: Movie understanding. Currently, most video understanding networks, such as those discussed in this chapter, tend to only deal with short video clips as input. What modifications might be necessary in order to operate over longer sequences such as an entire movie?

³⁴With a categorical variable, one-hot encoding is used to represent which label is chosen, i.e., when a label is chosen, its entry in the vector is 1 with all other entries being 0.

Chapter 7

Feature detection and matching

7.1	Points and patches	419
7.1.1	Feature detectors	422
7.1.2	Feature descriptors	434
7.1.3	Feature matching	441
7.1.4	Large-scale matching and retrieval	448
7.1.5	Feature tracking	452
7.1.6	<i>Application: Performance-driven animation</i>	454
7.2	Edges and contours	455
7.2.1	Edge detection	456
7.2.2	Contour detection	461
7.2.3	<i>Application: Edge editing and enhancement</i>	465
7.3	Contour tracking	466
7.3.1	Snakes and scissors	467
7.3.2	Level Sets	474
7.3.3	<i>Application: Contour tracking and rotoscoping</i>	476
7.4	Lines and vanishing points	477
7.4.1	Successive approximation	477
7.4.2	Hough transforms	477
7.4.3	Vanishing points	481
7.5	Segmentation	483
7.5.1	Graph-based segmentation	486
7.5.2	Mean shift	487
7.5.3	Normalized cuts	489
7.6	Additional reading	491
7.7	Exercises	495

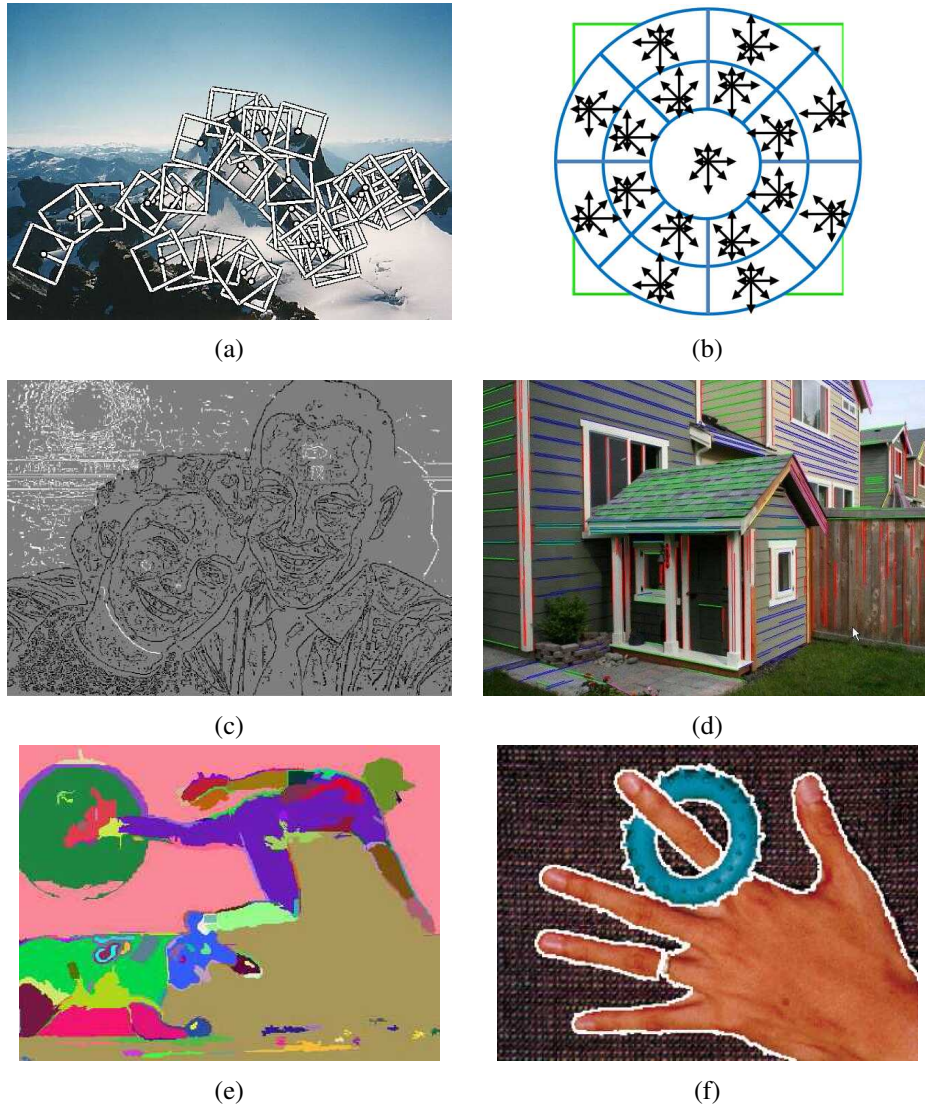


Figure 7.1 Feature detectors and descriptors can be used to analyze, describe and match images: (a) point-like interest operators (Brown, Szeliski, and Winder 2005) © 2005 IEEE; (b) GLOH descriptor (Mikolajczyk and Schmid 2005); (c) edges (Elder and Goldberg 2001) © 2001 IEEE; (d) straight lines (Sinha, Steedly et al. 2008) © 2008 ACM; (e) graph-based merging (Felzenszwalb and Huttenlocher 2004) © 2004 Springer; (f) mean shift (Comaniciu and Meer 2002) © 2002 IEEE.

Feature detection and matching are an essential component of many computer vision applications. Consider the two pairs of images shown in Figure 7.2. For the first pair, we may wish to *align* the two images so that they can be seamlessly stitched into a composite mosaic (Section 8.2). For the second pair, we may wish to establish a dense set of *correspondences* so that a 3D model can be constructed or an in-between view can be generated (Chapter 12). In either case, what kinds of *features* should you detect and then match to establish such an alignment or set of correspondences? Think about this for a few moments before reading on.

The first kind of feature that you may notice are specific locations in the images, such as mountain peaks, building corners, doorways, or interestingly shaped patches of snow. These kinds of localized features are often called *keypoint features* or *interest points* (or even *corners*) and are often described by the appearance of pixel patches surrounding the point location (Section 7.1). Another class of important features are *edges*, e.g., the profile of mountains against the sky (Section 7.2). These kinds of features can be matched based on their orientation and local appearance (edge profiles) and can also be good indicators of object boundaries and *occlusion* events in image sequences. Edges can be grouped into longer *curves* and *contours*, which can then be tracked (Section 7.3). They can also be grouped into *straight line segments*, which can be directly matched or analyzed to find *vanishing points* and hence internal and external camera parameters (Section 7.4).

In this chapter, we describe some practical approaches to detecting such features and also discuss how feature correspondences can be established across different images. Point features are now used in such a wide variety of applications that it is good practice to read and implement some of the algorithms from Section 7.1. Edges and lines provide information that is complementary to both keypoint and region-based descriptors and are well suited to describing the boundaries of manufactured objects. These alternative descriptors, while extremely useful, can be skipped in a short introductory course.

The last part of this chapter (Section 7.5) discusses bottom-up non-semantic segmentation techniques. While these were once widely used as essential components of both recognition and matching algorithms, they have mostly been supplanted by the *semantic segmentation* techniques we studied in Section 6.4. They are still used occasionally to group pixels together for faster or more reliable matching.

7.1 Points and patches

Point features can be used to find a sparse set of corresponding locations in different images, often as a precursor to computing camera pose (Chapter 11), which is a prerequisite for computing a denser set of correspondences using stereo matching (Chapter 12). Such cor-



Figure 7.2 *Two pairs of images to be matched. What kinds of features might one use to establish a set of correspondences between these images?*

respondences can also be used to align different images, e.g., when stitching image mosaics (Section 8.2) or high dynamic range images (Section 10.2), or performing video stabilization (Section 9.2.1). They are also used extensively to perform object instance recognition (Section 6.1). A key advantage of keypoints is that they permit matching even in the presence of clutter (occlusion) and large scale and orientation changes.

Feature-based correspondence techniques have been used since the early days of stereo matching (Hannah 1974; Moravec 1983; Hannah 1988) and subsequently gained popularity for image-stitching applications (Zoghلامي, Faugeras, and Deriche 1997; Brown and Lowe 2007) as well as fully automated 3D modeling (Beardsley, Torr, and Zisserman 1996; Schaffalitzky and Zisserman 2002; Brown and Lowe 2005; Snavely, Seitz, and Szeliski 2006).

There are two main approaches to finding feature points and their correspondences. The first is to find features in one image that can be accurately *tracked* using a local search technique, such as correlation or least squares (Section 7.1.5). The second is to independently detect features in all the images under consideration and then *match* features based on their local appearance (Section 7.1.3). The former approach is more suitable when images are taken from nearby viewpoints or in rapid succession (e.g., video sequences), while the latter is more suitable when a large amount of motion or appearance change is expected, e.g., in stitching together panoramas (Brown and Lowe 2007), establishing correspondences in *wide baseline stereo* (Schaffalitzky and Zisserman 2002), or performing object recognition

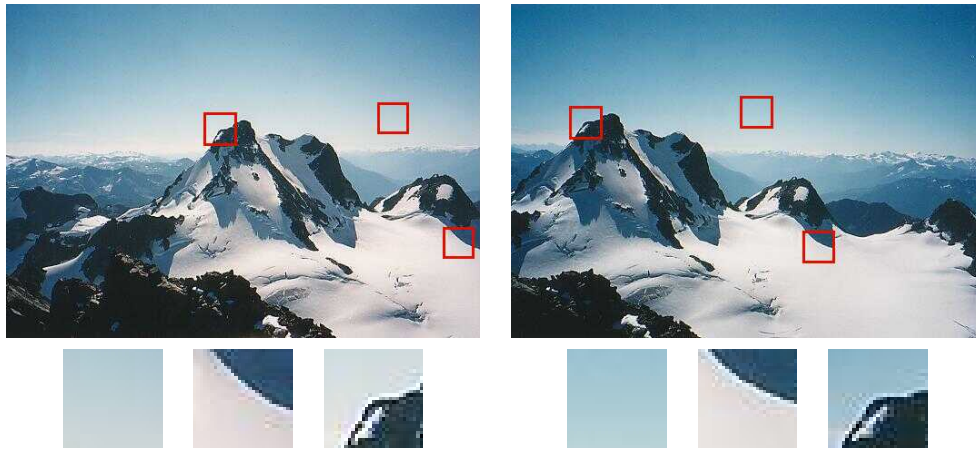


Figure 7.3 Image pairs with extracted patches below. Notice how some patches can be localized or matched with higher accuracy than others.

(Fergus, Perona, and Zisserman 2007).

In this section, we split the keypoint detection and matching pipeline into four separate stages. During the *feature detection* (extraction) stage (Section 7.1.1), each image is searched for locations that are likely to match well in other images. In the *feature description* stage (Section 7.1.2), each region around detected keypoint locations is converted into a more compact and stable (invariant) *descriptor* that can be matched against other descriptors. The *feature matching* stage (Sections 7.1.3 and 7.1.4) efficiently searches for likely matching candidates in other images. The *feature tracking* stage (Section 7.1.5) is an alternative to the third stage that only searches a small neighborhood around each detected feature and is therefore more suitable for video processing.

A wonderful example of all of these stages can be found in David Lowe’s (2004) paper, which describes the development and refinement of his *Scale Invariant Feature Transform* (SIFT). Comprehensive descriptions of alternative techniques can be found in a series of survey and evaluation papers covering both feature detection (Schmid, Mohr, and Bauckhage 2000; Mikolajczyk, Tuytelaars *et al.* 2005; Tuytelaars and Mikolajczyk 2008) and feature descriptors (Mikolajczyk and Schmid 2005; Balntas, Lenc *et al.* 2020). Shi and Tomasi (1994) and Triggs (2004) also provide nice reviews of classic (pre-neural network) feature detection techniques.

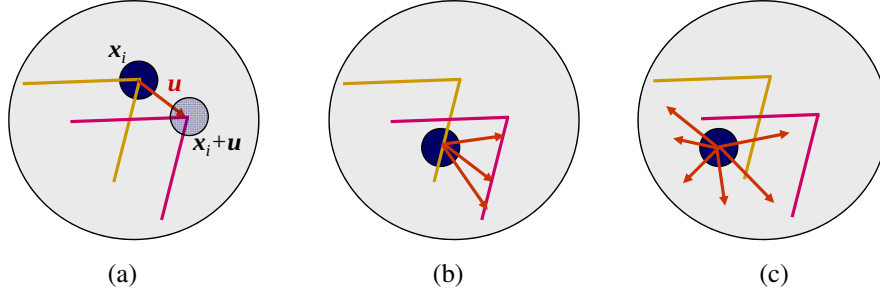


Figure 7.4 Aperture problems for different image patches: (a) stable (“corner-like”) flow; (b) classic aperture problem (barber-pole illusion); (c) textureless region. The two images I_0 (yellow) and I_1 (red) are overlaid. The red vector $\mathbf{u}$ indicates the displacement between the patch centers and the $w(\mathbf{x}_i)$ weighting function (patch window) is shown as a dark circle.

7.1.1 Feature detectors

How can we find image locations where we can reliably find correspondences with other images, i.e., what are good features to track (Shi and Tomasi 1994; Triggs 2004)? Look again at the image pair shown in Figure 7.3 and at the three sample *patches* to see how well they might be matched or tracked. As you may notice, textureless patches are nearly impossible to localize. Patches with large contrast changes (gradients) are easier to localize, although straight line segments at a single orientation suffer from the *aperture problem* (Horn and Schunck 1981; Lucas and Kanade 1981; Anandan 1989), i.e., it is only possible to align the patches along the direction *normal* to the edge direction (Figure 7.4b). Patches with gradients in at least two (significantly) different orientations are the easiest to localize, as shown schematically in Figure 7.4a.

These intuitions can be formalized by looking at the simplest possible matching criterion for comparing two image patches, i.e., their (weighted) summed square difference,

$$E_{\text{WSSD}}(\mathbf{u}) = \sum_i w(\mathbf{x}_i) [I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)]^2, \quad (7.1)$$

where I_0 and I_1 are the two images being compared, $\mathbf{u} = (u, v)$ is the *displacement* vector, $w(\mathbf{x})$ is a spatially varying weighting (or window) function, and the summation i is over all the pixels in the patch. Note that this is the same formulation we later use to estimate motion between complete images (Section 9.1).

When performing feature detection, we do not know which other image locations the feature will end up being matched against. Therefore, we can only compute how stable this metric is with respect to small variations in position $\Delta\mathbf{u}$ by comparing an image patch against

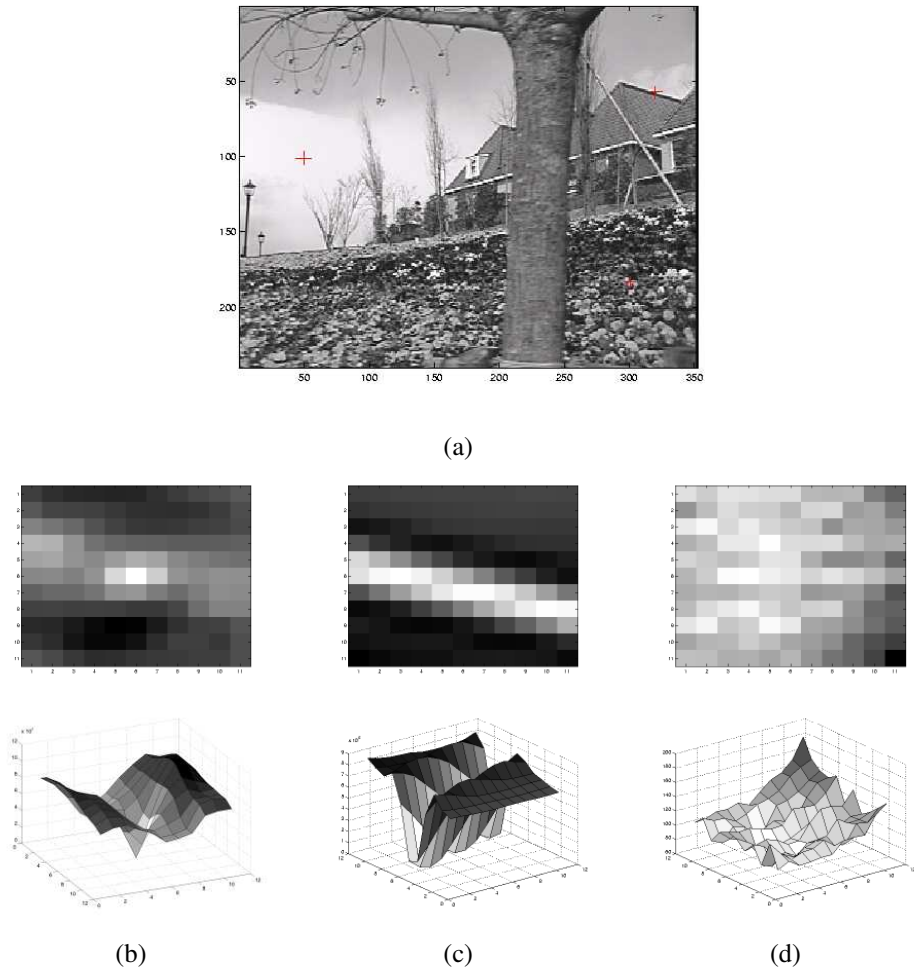


Figure 7.5 Three auto-correlation surfaces $E_{AC}(\Delta \mathbf{u})$ shown as both grayscale images and surface plots: (a) The original image is marked with three red crosses to denote where the auto-correlation surfaces were computed; (b) this patch is from the flower bed (good unique minimum); (c) this patch is from the roof edge (one-dimensional aperture problem); and (d) this patch is from the cloud (no good peak). Each grid point in figures b–d is one value of $\Delta \mathbf{u}$.

itself, which is known as an *auto-correlation function* or *surface*

$$E_{AC}(\Delta \mathbf{u}) = \sum_i w(\mathbf{x}_i) [I_0(\mathbf{x}_i + \Delta \mathbf{u}) - I_0(\mathbf{x}_i)]^2 \quad (7.2)$$

(Figure 7.5).¹ Note how the auto-correlation surface for the textured lower bed (Figure 7.5b and the red cross in the lower right quadrant of Figure 7.5a) exhibits a strong minimum, indicating that it can be well localized. The correlation surface corresponding to the roof edge (Figure 7.5c) has a strong ambiguity along one direction, while the correlation surface corresponding to the cloud region (Figure 7.5d) has no stable minimum.

Using a Taylor Series expansion of the image function $I_0(\mathbf{x}_i + \Delta \mathbf{u}) \approx I_0(\mathbf{x}_i) + \nabla I_0(\mathbf{x}_i) \cdot \Delta \mathbf{u}$ (Lucas and Kanade 1981; Shi and Tomasi 1994), we can approximate the auto-correlation surface as

$$E_{AC}(\Delta \mathbf{u}) = \sum_i w(\mathbf{x}_i) [I_0(\mathbf{x}_i + \Delta \mathbf{u}) - I_0(\mathbf{x}_i)]^2 \quad (7.3)$$

$$\approx \sum_i w(\mathbf{x}_i) [I_0(\mathbf{x}_i) + \nabla I_0(\mathbf{x}_i) \cdot \Delta \mathbf{u} - I_0(\mathbf{x}_i)]^2 \quad (7.4)$$

$$= \sum_i w(\mathbf{x}_i) [\nabla I_0(\mathbf{x}_i) \cdot \Delta \mathbf{u}]^2 \quad (7.5)$$

$$= \Delta \mathbf{u}^T \mathbf{A} \Delta \mathbf{u}, \quad (7.6)$$

where

$$\nabla I_0(\mathbf{x}_i) = \left(\frac{\partial I_0}{\partial x}, \frac{\partial I_0}{\partial y} \right)(\mathbf{x}_i) \quad (7.7)$$

is the *image gradient* at $\mathbf{x}_i$. This gradient can be computed using a variety of techniques (Schmid, Mohr, and Bauckhage 2000). The classic “Harris” detector (Harris and Stephens 1988) uses a $[-2 \ -1 \ 0 \ 1 \ 2]$ filter, but more modern variants (Schmid, Mohr, and Bauckhage 2000; Triggs 2004) convolve the image with horizontal and vertical derivatives of a Gaussian (typically with $\sigma = 1$).

The auto-correlation matrix $\mathbf{A}$ can be written as

$$\mathbf{A} = w * \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \quad (7.8)$$

where we have replaced the weighted summations with discrete convolutions with the weighting kernel w . This matrix can be interpreted as a tensor (multiband) image, where the outer products of the gradients ∇I are convolved with a weighting function w to provide a per-pixel estimate of the local (quadratic) shape of the auto-correlation function.

¹Strictly speaking, a correlation is the *product* of two patches (3.12); I’m using the term here in a more qualitative sense. The weighted sum of squared differences is often called an *SSD surface* (Section 9.1).

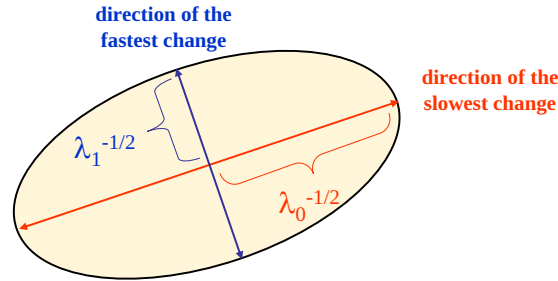


Figure 7.6 *Uncertainty ellipse corresponding to an eigenvalue analysis of the auto-correlation matrix $\mathbf{A}$.*

As first shown by Anandan (1984; 1989) and further discussed in Section 9.1.3 and Equation (9.37), the inverse of the matrix $\mathbf{A}$ provides a lower bound on the uncertainty in the location of a matching patch. It is therefore a useful indicator of which patches can be reliably matched. The easiest way to visualize and reason about this uncertainty is to perform an eigenvalue analysis of the auto-correlation matrix $\mathbf{A}$, which produces two eigenvalues (λ_0, λ_1) and two eigenvector directions (Figure 7.6). Since the larger uncertainty depends on the smaller eigenvalue, i.e., $\lambda_0^{-1/2}$, it makes sense to find maxima in the smaller eigenvalue to locate good features to track (Shi and Tomasi 1994).

Förstner–Harris. While Anandan (1984) and Lucas and Kanade (1981) were the first to analyze the uncertainty structure of the auto-correlation matrix, they did so in the context of associating certainties with optical flow measurements. Förstner (1986) and Harris and Stephens (1988) were the first to propose using local maxima in rotationally invariant scalar measures derived from the auto-correlation matrix to locate keypoints for the purpose of sparse feature matching.² Both of these techniques also proposed using a Gaussian weighting window instead of the previously used square patches, which makes the detector response insensitive to in-plane image rotations.

The minimum eigenvalue λ_0 (Shi and Tomasi 1994) is not the only quantity that can be used to find keypoints. A simpler quantity, proposed by Harris and Stephens (1988), is

$$\det(\mathbf{A}) - \alpha \text{trace}(\mathbf{A})^2 = \lambda_0 \lambda_1 - \alpha(\lambda_0 + \lambda_1)^2 \quad (7.9)$$

with $\alpha = 0.06$. Unlike eigenvalue analysis, this quantity does not require the use of square roots and yet is still rotationally invariant and also downweights edge-like features where

²Schmid, Mohr, and Bauckhage (2000) and Triggs (2004) give more detailed historical reviews of feature detection algorithms.

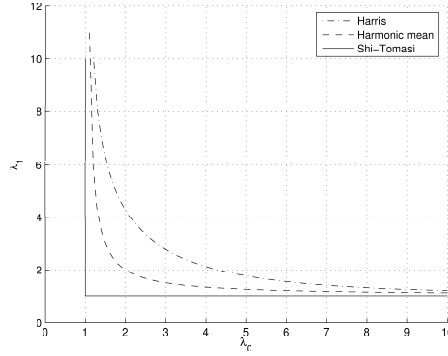


Figure 7.7 Isocontours of popular keypoint detection functions (Brown, Szeliski, and Winder 2004). Each detector looks for points where the eigenvalues λ_0, λ_1 of $\mathbf{A} = w * \nabla I \nabla I^T$ are both large.

$\lambda_1 \gg \lambda_0$. Triggs (2004) suggests using the quantity

$$\lambda_0 - \alpha \lambda_1 \quad (7.10)$$

(say, with $\alpha = 0.05$), which also reduces the response at 1D edges, where aliasing errors sometimes inflate the smaller eigenvalue. He also shows how the basic 2×2 Hessian can be extended to parametric motions to detect points that are also accurately localizable in scale and rotation. Brown, Szeliski, and Winder (2005), on the other hand, use the harmonic mean,

$$\frac{\det \mathbf{A}}{\text{tr } \mathbf{A}} = \frac{\lambda_0 \lambda_1}{\lambda_0 + \lambda_1}, \quad (7.11)$$

which is a smoother function in the region where $\lambda_0 \approx \lambda_1$. Figure 7.7 shows isocontours of the various interest point operators, from which we can see how the two eigenvalues are blended to determine the final interest value. Figure 7.8 shows the resulting interest operator responses for the classic Harris detector as well as the difference of Gaussian (DoG) detector discussed below.

Adaptive non-maximal suppression (ANMS). While most feature detectors simply look for local maxima in the interest function, this can lead to an uneven distribution of feature points across the image, e.g., points will be denser in regions of higher contrast. To mitigate this problem, Brown, Szeliski, and Winder (2005) only detect features that are both local maxima and whose response value is significantly (10%) greater than that of all of its neighbors within a radius r (Figure 7.9c–d). They devise an efficient way to associate suppression radii with all local maxima by first sorting them by their response strength and then creating

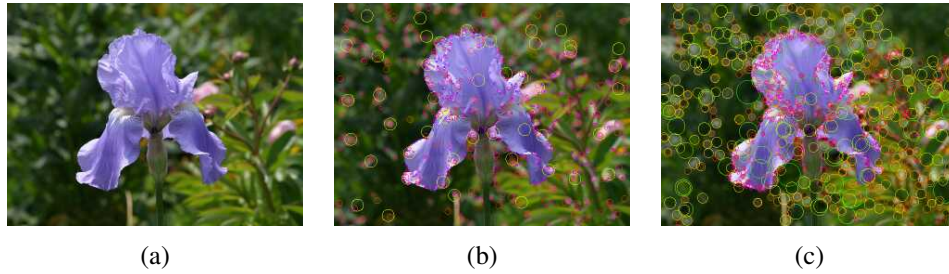


Figure 7.8 Interest operator responses: (a) Sample image, (b) Harris response, and (c) DoG response. The circle sizes and colors indicate the scale at which each interest point was detected. Notice how the two detectors tend to respond at complementary locations.

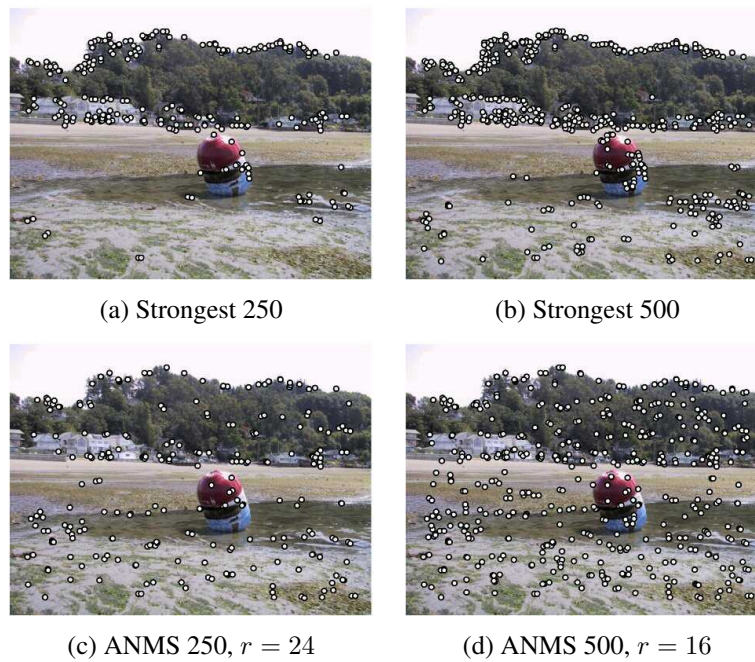


Figure 7.9 Adaptive non-maximal suppression (ANMS) (Brown, Szeliski, and Winder 2005) © 2005 IEEE: The upper two images show the strongest 250 and 500 interest points, while the lower two images show the interest points selected with adaptive non-maximal suppression, along with the corresponding suppression radius r . Note how the latter features have a much more uniform spatial distribution across the image.

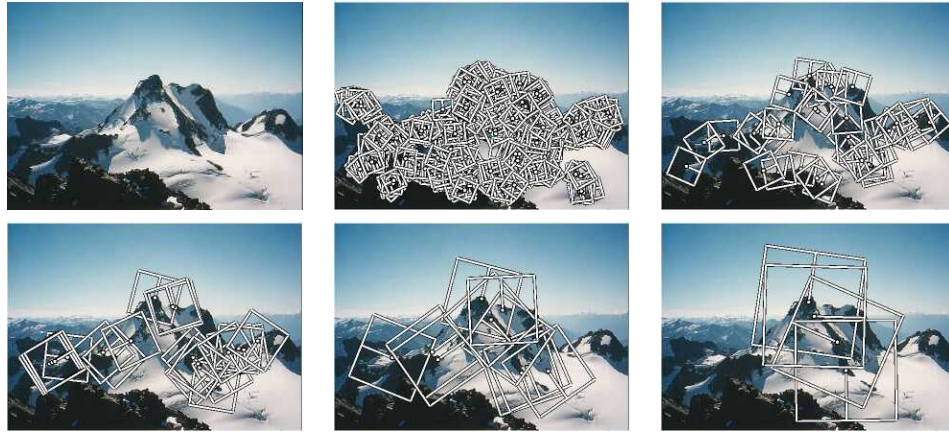


Figure 7.10 *Multi-scale oriented patches (MOPS) extracted at five pyramid levels (Brown, Szeliski, and Winder 2005) © 2005 IEEE. The boxes show the feature orientation and the region from which the descriptor vectors are sampled.*

a second list sorted by decreasing suppression radius (Brown, Szeliski, and Winder 2005). Figure 7.9 shows a qualitative comparison of selecting the top n features and using ANMS. Note that non-maximal suppression is now also an essential component of DNN-based object detectors, as discussed in Section 6.3.3.

Measuring repeatability. Given the large number of feature detectors that have been developed in computer vision, how can we decide which ones to use? Schmid, Mohr, and Bauckhage (2000) were the first to propose measuring the *repeatability* of feature detectors, which they define as the frequency with which keypoints detected in one image are found within ϵ (say, $\epsilon = 1.5$) pixels of the corresponding location in a transformed image. In their paper, they transform their planar images by applying rotations, scale changes, illumination changes, viewpoint changes, and adding noise. They also measure the *information content* available at each detected feature point, which they define as the entropy of a set of rotationally invariant local grayscale descriptors. Among the techniques they survey, they find that the improved (Gaussian derivative) version of the Harris operator with $\sigma_d = 1$ (scale of the derivative Gaussian) and $\sigma_i = 2$ (scale of the integration Gaussian) works best.

Scale invariance

In many situations, detecting features at the finest stable scale possible may not be appropriate. For example, when matching images with little high-frequency detail (e.g., clouds),

fine-scale features may not exist.

One solution to the problem is to extract features at a variety of scales, e.g., by performing the same operations at multiple resolutions in a pyramid and then matching features at the same level. This kind of approach is suitable when the images being matched do not undergo large scale changes, e.g., when matching successive aerial images taken from an airplane or stitching panoramas taken with a fixed-focal-length camera. Figure 7.10 shows the output of one such approach: the multi-scale oriented patch detector of Brown, Szeliski, and Winder (2005), for which responses at five different scales are shown.

However, for most object recognition applications, the scale of the object in the image is unknown. Instead of extracting features at many different scales and then matching all of them, it is more efficient to extract features that are stable in both location *and* scale (Lowe 2004; Mikolajczyk and Schmid 2004).

Early investigations into scale selection were performed by Lindeberg (1993; 1998b), who first proposed using extrema in the Laplacian of Gaussian (LoG) function as interest point locations. Based on this work, Lowe (2004) proposed computing a set of sub-octave Difference of Gaussian filters (Figure 7.11a), looking for 3D (space+scale) maxima in the resulting structure (Figure 7.11b), and then computing a sub-pixel space+scale location using a quadratic fit (Brown and Lowe 2002). The number of sub-octave levels was determined, after careful empirical investigation, to be three, which corresponds to a quarter-octave pyramid, which is the same as used by Triggs (2004).

As with the Harris operator, pixels where there is strong asymmetry in the local curvature of the indicator function (in this case, the DoG) are rejected. This is implemented by first computing the local Hessian of the difference image D ,

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}, \quad (7.12)$$

and then rejecting keypoints for which

$$\frac{\text{Tr}(\mathbf{H})^2}{\text{Det}(\mathbf{H})} > 10. \quad (7.13)$$

While Lowe's Scale Invariant Feature Transform (SIFT) performs well in practice, it is not based on the same theoretical foundation of maximum spatial stability as the auto-correlation-based detectors. (In fact, its detection locations are often complementary to those produced by such techniques and can therefore be used in conjunction with these other approaches.) In order to add a scale selection mechanism to the Harris corner detector, Mikolajczyk and Schmid (2004) evaluate the Laplacian of Gaussian function at each detected Harris point (in a multi-scale pyramid) and keep only those points for which the Laplacian is extremal (larger

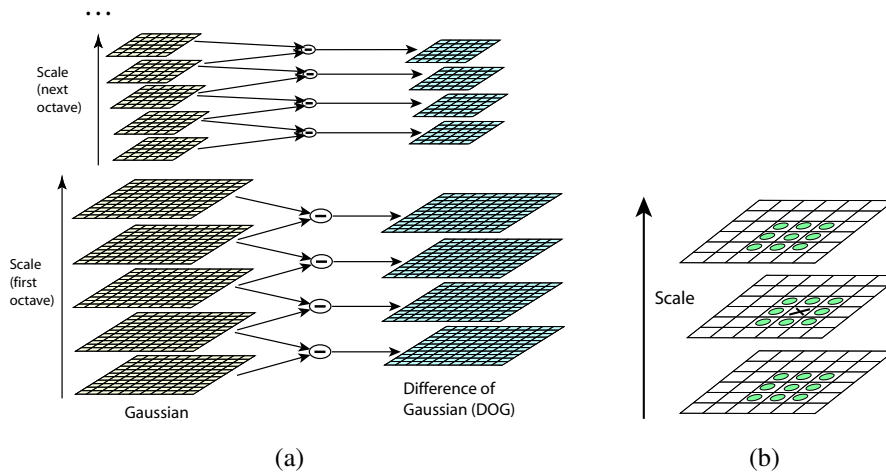


Figure 7.11 *Scale-space feature detection using a sub-octave Difference of Gaussian pyramid (Lowe 2004) © 2004 Springer: (a) Adjacent levels of a sub-octave Gaussian pyramid are subtracted to produce Difference of Gaussian images; (b) extrema (maxima and minima) in the resulting 3D volume are detected by comparing a pixel to its 26 neighbors.*

or smaller than both its coarser and finer-level values). An optional iterative refinement for both scale and position is also proposed and evaluated. Additional examples of scale-invariant region detectors are discussed by Mikolajczyk, Tuytelaars *et al.* (2005) and Tuytelaars and Mikolajczyk (2008).

Rotational invariance and orientation estimation

In addition to dealing with scale changes, most image matching and object recognition algorithms need to deal with (at least) in-plane image rotation. One way to deal with this problem is to design descriptors that are rotationally invariant (Schmid and Mohr 1997), but such descriptors have poor discriminability, i.e. they map different looking patches to the same descriptor.

A better method is to estimate a *dominant orientation* at each detected keypoint. Once the local orientation and scale of a keypoint have been estimated, a scaled and oriented patch around the detected point can be extracted and used to form a feature descriptor (Figures 7.10 and 7.15).

The simplest possible orientation estimate is the average gradient within a region around the keypoint. If a Gaussian weighting function is used (Brown, Szeliski, and Winder 2005),

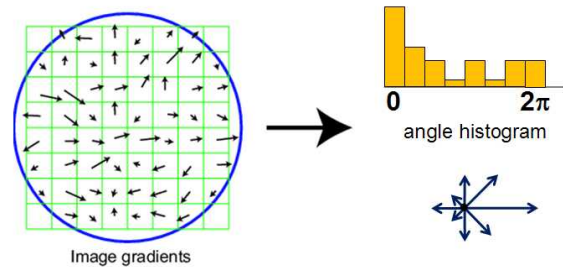


Figure 7.12 A dominant orientation estimate can be computed by creating a histogram of all the gradient orientations (weighted by their magnitudes or after thresholding out small gradients) and then finding the significant peaks in this distribution (Lowe 2004) © 2004 Springer.

this average gradient is equivalent to a first-order steerable filter (Section 3.2.3), i.e., it can be computed using an image convolution with the horizontal and vertical derivatives of Gaussian filter (Freeman and Adelson 1991). To make this estimate more reliable, it is usually preferable to use a larger aggregation window (Gaussian kernel size) than detection window (Brown, Szeliski, and Winder 2005). The orientations of the square boxes shown in Figure 7.10 were computed using this technique.

Sometimes, however, the averaged (signed) gradient in a region can be small and therefore an unreliable indicator of orientation. A more reliable technique is to look at the *histogram* of orientations computed around the keypoint. Lowe (2004) computes a 36-bin histogram of edge orientations weighted by both gradient magnitude and Gaussian distance to the center, finds all peaks within 80% of the global maximum, and then computes a more accurate orientation estimate using a three-bin parabolic fit (Figure 7.12).

Affine invariance

While scale and rotation invariance are highly desirable, for many applications such as *wide baseline stereo matching* (Pritchett and Zisserman 1998; Schaffalitzky and Zisserman 2002) or location recognition (Chum, Philbin *et al.* 2007), full affine invariance is preferred. Affine-invariant detectors not only respond at consistent locations after scale and orientation changes, they also respond consistently across affine deformations such as (local) perspective foreshortening (Figure 7.13). In fact, for a small enough patch, any continuous image warping can be well approximated by an affine deformation.

To introduce affine invariance, several authors have proposed fitting an ellipse to the auto-

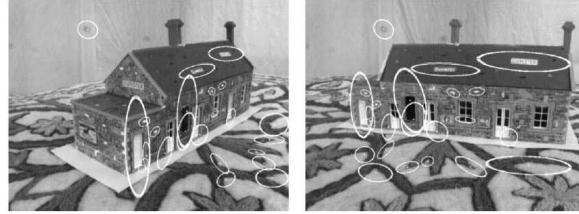


Figure 7.13 Affine region detectors used to match two images taken from dramatically different viewpoints (Mikolajczyk and Schmid 2004) © 2004 Springer.



Figure 7.14 Maximally stable extremal regions (MSERs) extracted and matched from a number of images (Matas, Chum et al. 2004) © 2004 Elsevier.

correlation or Hessian matrix (using eigenvalue analysis) and then using the principal axes and ratios of this fit as the affine coordinate frame (Lindeberg and Gärdening 1997; Baumberg 2000; Mikolajczyk and Schmid 2004; Mikolajczyk, Tuytelaars *et al.* 2005; Tuytelaars and Mikolajczyk 2008).

Another important affine invariant region detector is the maximally stable extremal region (MSER) detector developed by Matas, Chum *et al.* (2004). To detect MSERs, binary regions are computed by thresholding the image at all possible gray levels (the technique therefore only works for grayscale images). This operation can be performed efficiently by first sorting all pixels by gray value and then incrementally adding pixels to each connected component as the threshold is changed (Nistér and Stewénus 2008). As the threshold is changed, the area of each component (region) is monitored; regions whose rate of change of area with respect to the threshold is minimal are defined as *maximally stable*. Such regions are therefore invariant to both affine geometric and photometric (linear bias-gain or smooth monotonic) transformations (Figure 7.14). If desired, an affine coordinate frame can be fit to each detected region using its moment matrix.

The area of feature point detection continues to be very active, with papers appearing every year at major computer vision conferences. Mikolajczyk, Tuytelaars *et al.* (2005) and Tuytelaars and Mikolajczyk (2008) survey a number of popular (pre-DNN) affine region detectors and provide experimental comparisons of their invariance to common image transfor-

mations such as scaling, rotations, noise, and blur.

More recent papers published in the last decade include:

- SURF (Bay, Ess *et al.* 2008), which uses integral images for faster convolutions;
- FAST and FASTER (Rosten, Porter, and Drummond 2010), one of the first learned detectors;
- BRISK (Leutenegger, Chli, and Siegwart 2011), which uses a scale-space FAST detector together with a bit-string descriptor;
- ORB (Rubblee, Rabaud *et al.* 2011), which adds orientation to FAST; and
- KAZE (Alcantarilla, Bartoli, and Davison 2012) and Accelerated-KAZE (Alcantarilla, Nuevo, and Bartoli 2013), which use non-linear diffusion to select the scale for feature detection.

While FAST introduced the idea of machine learning for feature detectors, more recent papers use convolutional neural networks to perform the detection. These include:

- Learning covariant feature detectors (Lenc and Vedaldi 2016);
- Learning to assign orientations to feature points (Yi, Verdie *et al.* 2016);
- LIFT, learned invariant feature transforms (Yi, Trulls *et al.* 2016), SuperPoint, self-supervised interest point detection and description (DeTone, Malisiewicz, and Rabinovich 2018), and LF-Net, learning local features from images (Ono, Trulls *et al.* 2018), all three of which jointly optimize the detectors and descriptors in a single (multi-head) pipeline;
- AffNet (Mishkin, Radenovic, and Matas 2018), which detects matchable affine-covariant regions;
- Key.Net (Barroso-Laguna, Riba *et al.* 2019), which uses a combination of handcrafted and learned CNN features; and
- D2-Net (Dusmanu, Rocco *et al.* 2019), R2D2 (Revaud, Weinzaepfel *et al.* 2019), and D2D (Tian, Balntas *et al.* 2020), which all extract dense local feature descriptors and then keeps the ones that have high saliency or repeatability.

These last two papers also contains a nice review of other recent feature detectors, as does the paper by Balntas, Lenc *et al.* (2020).

Of course, keypoints are not the only features that can be used for registering images. Zoghلامي, Faugeras, and Deriche (1997) use line segments as well as point-like features to estimate homographies between pairs of images, whereas Bartoli, Coquerelle, and Sturm (2004) use line segments with local correspondences along the edges to extract 3D structure and motion. Tuytelaars and Van Gool (2004) use affine invariant regions to detect correspondences for wide baseline stereo matching, whereas Kadir, Zisserman, and Brady (2004) detect salient regions where patch entropy and its rate of change with scale are locally maximal. Corso and Hager (2005) use a related technique to fit 2D oriented Gaussian kernels to homogeneous regions. More details on techniques for finding and matching curves, lines, and regions can be found later in this chapter.

7.1.2 Feature descriptors

After detecting keypoint features, we must *match* them, i.e., we must determine which features come from corresponding locations in different images. In some situations, e.g., for video sequences (Shi and Tomasi 1994) or for stereo pairs that have been *rectified* (Zhang, Deriche *et al.* 1995; Loop and Zhang 1999; Scharstein and Szeliski 2002), the local motion around each feature point may be mostly translational. In this case, simple error metrics, such as the *sum of squared differences* or *normalized cross-correlation*, described in Section 9.1, can be used to directly compare the intensities in small patches around each feature point. (The comparative study by Mikolajczyk and Schmid (2005), discussed below, uses cross-correlation.) Because feature points may not be exactly located, a more accurate matching score can be computed by performing incremental motion refinement as described in Section 9.1.3, but this can be time-consuming and can sometimes even decrease performance (Brown, Szeliski, and Winder 2005).

In most cases, however, the local appearance of features will change in orientation and scale, and sometimes even undergo affine deformations. Extracting a local scale, orientation, or affine frame estimate and then using this to resample the patch before forming the feature descriptor is thus usually preferable (Figure 7.15).

Even after compensating for these changes, the local appearance of image patches will usually still vary from image to image. How can we make image descriptors more invariant to such changes, while still preserving discriminability between different (non-corresponding) patches? Mikolajczyk and Schmid (2005) review a number of view-invariant local image descriptors and experimentally compare their performance. More recently, Balntas, Lenc *et al.* (2020) and Jin, Mishkin *et al.* (2021) compare the large number of learned feature descriptors developed in the prior decade.³ Below, we describe a few of these descriptors in

³Many recent publications such as Tian, Yu *et al.* (2019) use their HPatches dataset to compare their performance

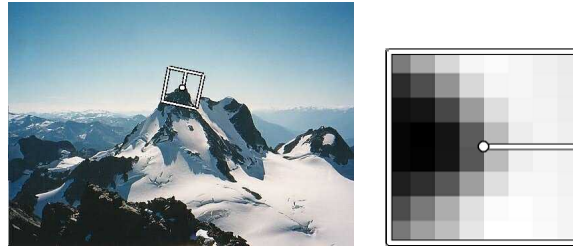


Figure 7.15 Once a local scale and orientation estimate has been determined, MOPS descriptors are formed using an 8×8 sampling of bias and gain normalized intensity values, with a sample spacing of five pixels relative to the detection scale (Brown, Szeliski, and Winder 2005) © 2005 IEEE. This low frequency sampling gives the features some robustness to interest point location error and is achieved by sampling at a higher pyramid level than the detection scale.

more detail.

Bias and gain normalization (MOPS). For tasks that do not exhibit large amounts of fore-shortening, such as image stitching, simple normalized intensity patches perform reasonably well and are simple to implement (Brown, Szeliski, and Winder 2005) (Figure 7.15). To compensate for slight inaccuracies in the feature point detector (location, orientation, and scale), multi-scale oriented patches (MOPS) are sampled at a spacing of five pixels relative to the detection scale, using a coarser level of the image pyramid to avoid aliasing. To compensate for affine photometric variations (linear exposure changes or bias and gain, (3.3)), patch intensities are re-scaled so that their mean is zero and their variance is one.

Scale invariant feature transform (SIFT). SIFT features (Lowe 2004) are formed by computing the gradient at each pixel in a 16×16 window around the detected keypoint, using the appropriate level of the Gaussian pyramid at which the keypoint was detected. The gradient magnitudes are downweighted by a Gaussian fall-off function (shown as a blue circle in Figure 7.16a) to reduce the influence of gradients far from the center, as these are more affected by small misregistrations.

In each 4×4 quadrant, a gradient orientation histogram is formed by (conceptually) adding the gradient values weighted by the Gaussian fall-off function to one of eight orientation histogram bins. To reduce the effects of location and dominant orientation misestimation, each of the original 256 weighted gradient magnitudes is softly added to $2 \times 2 \times 2$ adjacent

against previous approaches.

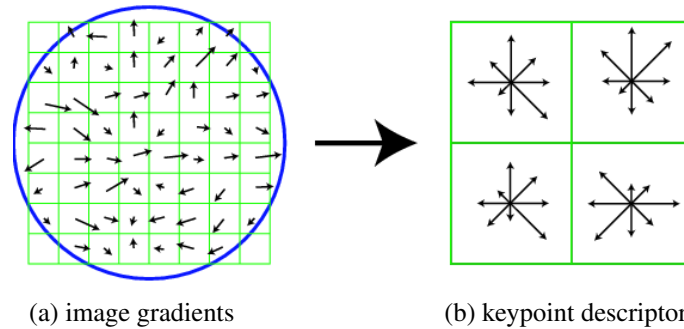


Figure 7.16 A schematic representation of Lowe's (2004) scale invariant feature transform (SIFT): (a) Gradient orientations and magnitudes are computed at each pixel and weighted by a Gaussian fall-off function (blue circle). (b) A weighted gradient orientation histogram is then computed in each subregion, using trilinear interpolation. While this figure shows an 8×8 pixel patch and a 2×2 descriptor array, Lowe's actual implementation uses 16×16 patches and a 4×4 array of eight-bin histograms.

histogram bins in the (x, y, θ) space using trilinear interpolation. Softly distributing values to adjacent histogram bins is generally a good idea in any application where histograms are being computed, e.g., for Hough transforms (Section 7.4.2) or local histogram equalization (Section 3.1.4).

The 4×4 array of eight-bin histogram yields 128 non-negative values form a raw version of the SIFT descriptor vector. To reduce the effects of contrast or gain (additive variations are already removed by the gradient), the 128-D vector is normalized to unit length. To further make the descriptor robust to other photometric variations, values are clipped to 0.2 and the resulting vector is once again renormalized to unit length.

PCA-SIFT. Ke and Sukthankar (2004) propose a simpler way to compute descriptors inspired by SIFT; it computes the x and y (gradient) derivatives over a 39×39 patch and then reduces the resulting 3042-dimensional vector to 36 using principal component analysis (PCA) (Section 5.2.3 and Appendix A.1.2). Another popular variant of SIFT is SURF (Bay, Ess *et al.* 2008), which uses box filters to approximate the derivatives and integrals used in SIFT.

RootSIFT. Arandjelović and Zisserman (2012) observe that by simply re-normalizing SIFT descriptors using an L_1 measure and then taking the square root of each component, a dramatic increase in performance (discriminability) can be obtained.

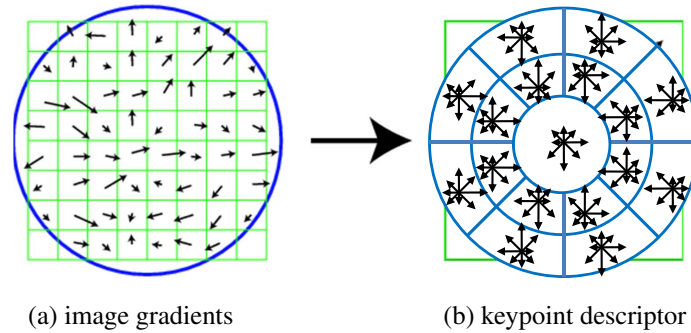


Figure 7.17 The gradient location-orientation histogram (GLOH) descriptor uses log-polar bins instead of square bins to compute orientation histograms (Mikolajczyk and Schmid 2005). GLOH uses 16 gradient orientations inside each bin, although this figure only shows 8 to appear less cluttered.

Gradient location-orientation histogram (GLOH). This descriptor, developed by Mikolajczyk and Schmid (2005), is a variant of SIFT that uses a log-polar binning structure instead of the four quadrants used by Lowe (2004) (Figure 7.17). The spatial bins extend over the radii $0 \dots 6$, $6 \dots 11$, and $11 \dots 15$, with eight angular bins (except for the single central region), for a total of 17 spatial bins and GLOH uses 16 orientation bins instead of the 8 used in SIFT. The 272-dimensional histogram is then projected onto a 128-dimensional descriptor using PCA trained on a large database. In their evaluation, Mikolajczyk and Schmid (2005) found that GLOH, which has the best performance overall, outperforms SIFT by a small margin.

Steerable filters. Steerable filters (Section 3.2.3) are combinations of derivative of Gaussian filters that permit the rapid computation of even and odd (symmetric and anti-symmetric) edge-like and corner-like features at all possible orientations (Freeman and Adelson 1991). Because they use reasonably broad Gaussians, they too are somewhat insensitive to localization and orientation errors.

Performance of local descriptors. Among the local descriptors that Mikolajczyk and Schmid (2005) compared, they found that GLOH performed best, followed closely by SIFT. They also present results for many other descriptors not covered in this book.

The field of feature descriptors continued to advance rapidly, with some techniques looking at local color information (van de Weijer and Schmid 2006; Abdel-Hakim and Farag 2006). Winder and Brown (2007) develop a multi-stage framework for feature descriptor computation that subsumes both SIFT and GLOH (Figure 7.18a) and also allows them to

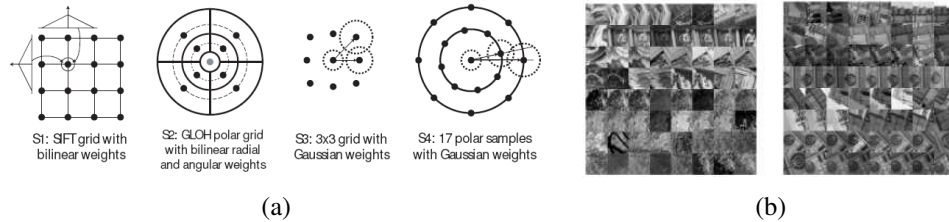


Figure 7.18 *Spatial summation blocks for SIFT, GLOH, and some related feature descriptors (Winder and Brown 2007) © 2007 IEEE: (a) The parameters for the features, e.g., their Gaussian weights, are learned from a training database of (b) matched real-world image patches obtained from robust structure from motion applied to internet photo collections (Hua, Brown, and Winder 2007).*

learn optimal parameters for newer descriptors that outperform previous hand-tuned descriptors. Hua, Brown, and Winder (2007) extend this work by learning lower-dimensional projections of higher-dimensional descriptors that have the best discriminative power, and Brown, Hua, and Winder (2011) further extend it by learning the optimal placement of the pooling regions. All of these papers use a database of real-world image patches (Figure 7.18b) obtained by sampling images at locations that were reliably matched using a robust structure-from-motion algorithm applied to internet photo collections (Snavely, Seitz, and Szeliski 2006; Goesele, Snavely *et al.* 2007). In concurrent work, Tola, Lepetit, and Fua (2010) developed a similar DAISY descriptor for dense stereo matching and optimized its parameters based on ground truth stereo data.

While these techniques construct feature detectors that optimize for repeatability across *all* object classes, it is also possible to develop class- or instance-specific feature detectors that maximize *discriminability* from other classes (Ferencz, Learned-Miller, and Malik 2008). If planar surface orientations can be determined in the images being matched, it is also possible to extract viewpoint-invariant patches (Wu, Clipp *et al.* 2008).

A more recent trend has been the development of binary bit-string feature descriptors, which can take advantage of fast Hamming distance operators in modern computer architectures. The BRIEF descriptor (Calonder, Lepetit *et al.* 2010) compares 128 different pairs of pixel values (denoted as line segments in Figure 7.19a) scattered around the keypoint location to obtain a 128-bit vector. ORB (Rublee, Rabaud *et al.* 2011) adds an orientation component to the FAST detector before computing oriented BRIEF descriptors. BRISK (Leutenegger, Chli, and Siewgwart 2011) adds scale-space analysis to the FAST detector and a radially symmetric sampling pattern (Figure 7.19b) to produce the binary descriptor. FREAK (Alahi,

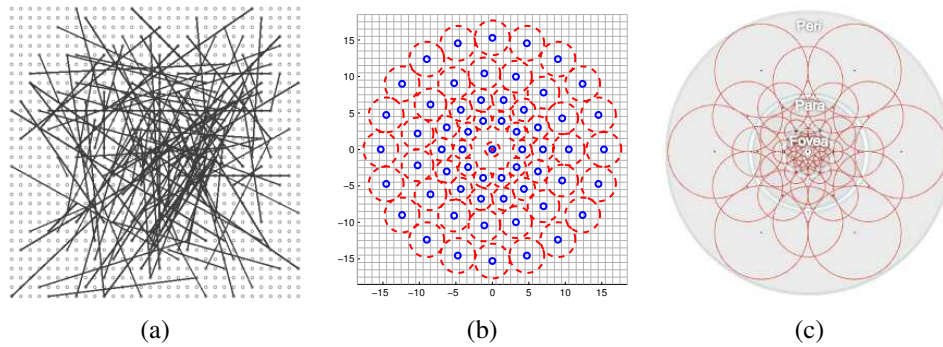


Figure 7.19 Binary bit-string feature descriptors: (a) the BRIEF descriptor compares 128 pairs of pixel values (denoted by line segments) and stores the comparison results in a 128-bit vector (Calonder, Lepetit *et al.* 2010) © 2010 Springer; (b) BRISK sampling pattern and Gaussian blur radii; (Leutenegger, Chli, and Siegwart 2011) © 2011 IEEE; (c) FREAK retinal sampling pattern (Alahi, Ortiz, and Vanderghenst 2012) © 2012 IEEE.

Ortiz, and Vanderghenst 2012) uses a more pronounced “retinal” (log-polar) sampling pattern paired with a cascade of bit comparisons for even greater speed and efficiency. The survey and evaluation by Mukherjee, Wu, and Wang (2015) compares all of these “classic” feature detectors and descriptors.

Since 2015 or so, most of the new feature descriptors are constructed using deep learning techniques, as surveyed in Balntas, Lenc *et al.* (2020) and Jin, Mishkin *et al.* (2021). Some of these descriptors, such as LIFT (Yi, Trulls *et al.* 2016), TFeat (Balntas, Riba *et al.* 2016), HPatches (Balntas, Lenc *et al.* 2020), L2-Net (Tian, Fan, and Wu 2017), HardNet (Mishchuk, Mishkin *et al.* 2017), Geodesc (Luo, Shen *et al.* 2018), LF-Net (Ono, Trulls *et al.* 2018), SOSNet (Tian, Yu *et al.* 2019), and Key.Net (Barroso-Laguna, Riba *et al.* 2019) operate on patches, much like the classical SIFT approach. They hence require an initial local feature detector to determine the center of the patch and use a predetermined patch size when constructing the input to the network.

In contrast, approaches such as DELF (Noh, Araujo *et al.* 2017), SuperPoint (DeTone, Malisiewicz, and Rabinovich 2018), D2-Net (Dusmanu, Rocco *et al.* 2019), ContextDesc (Luo, Shen *et al.* 2019), R2D2 (Revaud, Weinzaepfel *et al.* 2019), ASLFeat (Luo, Zhou *et al.* 2020), and CAPS (Wang, Zhou *et al.* 2020) use the entire image as the input to the descriptor computation. This has the added benefit that the receptive field used to compute the descriptor can be learned from the data and does not require specifying a patch size. Theoretically, these CNN models can learn receptive fields that use all of the pixels in the image, although in

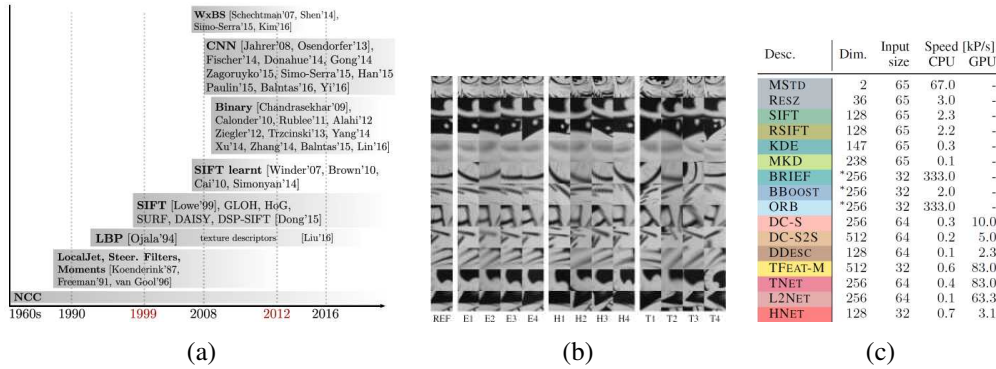


Figure 7.20 HPatches local descriptors benchmark (Balntas, Lenc et al. 2020) © 2019 IEEE: (a) chronology of feature descriptors; (b) typical patches in the dataset (grouped by Easy, Hard, and Tough); (c) size and speed of different descriptors.

practice they tend to use Gaussian-like receptive fields (Zhou, Khosla et al. 2015; Luo, Li et al. 2016; Selvaraju, Cogswell et al. 2017).

In the HPatches benchmark (Figure 7.20) for evaluating patch matching by Balntas, Lenc et al. (2020), HardNet and L2-net performed the best on average. Another paper (Wang, Zhou et al. 2020) shows CAPS and R2D2 as the best performers, while S2DNet (Germain, Bourmaud, and Lepetit 2020) and LISRD (Pautrat, Larsson et al. 2020) also claim state-of-the-art performance, while the WISW benchmark (Bellavia and Colombo 2020) shows that traditional descriptors such as SIFT enhanced with more recent ideas do the best. On the wide baseline image matching benchmark by Jin, Mishkin et al. (2021),⁴ HardNet, Key.Net, and D2-Net were top performers (e.g., D2-Net had the highest number of landmarks), although the results were quite task-dependent and the Difference of Gaussian detector was still the best. The performance of these descriptors on matching features across large illumination differences (day-night) has also been studied (Radenović, Schönberger et al. 2016; Zhou, Sattler, and Jacobs 2016; Mishkin 2021).

The most recent trend in wide-baseline matching has been to densely extract features without a detector stage and to then match and refine the set of correspondences (Jiang, Trulls et al. 2021; Sarlin, Unagar et al. 2021; Sun, Shen et al. 2021; Truong, Danelljan et al. 2021; Zhou, Sattler, and Leal-Taixé 2021). Some of these more recent techniques have been evaluated by Mishkin (2021).

⁴The benchmark is associated with the CVPR Workshop on Image Matching: Local Features & Beyond: <https://image-matching-workshop.github.io>.

7.1.3 Feature matching

Once we have extracted features and their descriptors from two or more images, the next step is to establish some preliminary feature matches between these images. The approach we take depends partially on the application, e.g., different strategies may be preferable for matching images that are known to overlap (e.g., in image stitching) vs. images that may have no correspondence whatsoever (e.g., when trying to recognize objects from a database).

In this section, we divide this problem into two separate components. The first is to select a *matching strategy*, which determines which correspondences are passed on to the next stage for further processing. The second is to devise efficient *data structures* and *algorithms* to perform this matching as quickly as possible, which we expand on in Section 7.1.4.

Matching strategy and error rates

Determining which feature matches are reasonable to process further depends on the context in which the matching is being performed. Say we are given two images that overlap to a fair amount (e.g., for image stitching or for tracking objects in a video). We know that most features in one image are likely to match the other image, although some may not match because they are occluded or their appearance has changed too much.

On the other hand, if we are trying to recognize how many known objects appear in a cluttered scene (Figure 6.2), most of the features may not match. Furthermore, a large number of potentially matching objects must be searched, which requires more efficient strategies, as described below.

To begin with, we assume that the feature descriptors have been designed so that Euclidean (vector magnitude) distances in feature space can be directly used for ranking potential matches. If it turns out that certain parameters (axes) in a descriptor are more reliable than others, it is usually preferable to re-scale these axes ahead of time, e.g., by determining how much they vary when compared against other known good matches (Hua, Brown, and Winder 2007). A more general process, which involves transforming feature vectors into a new scaled basis, is called *whitening* and is discussed in more detail in the context of eigenface-based face recognition (Section 5.2.3).

Given a Euclidean distance metric, the simplest matching strategy is to set a threshold (maximum distance) and to return all matches from other images within this threshold. Setting the threshold too high results in too many *false positives*, i.e., incorrect matches being returned. Setting the threshold too low results in too many *false negatives*, i.e., too many correct matches being missed (Figure 7.21).

We can quantify the performance of a matching algorithm at a particular threshold by

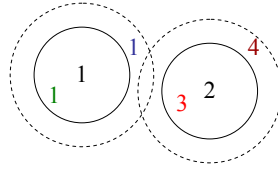


Figure 7.21 False positives and negatives: The black digits 1 and 2 are features being matched against a database of features in other images. At the current threshold setting (the solid circles), the green 1 is a true positive (good match), the blue 1 is a false negative (failure to match), and the red 3 is a false positive (incorrect match). If we set the threshold higher (the dashed circles), the blue 1 becomes a true positive but the brown 4 becomes an additional false positive.

	True matches	True non-matches	
Predicted matches	TP = 18	FP = 4	P' = 22
Predicted non-matches	FN = 2	TN = 76	N' = 78
	P = 20	N = 80	Total = 100

TPR = 0.90	FPR = 0.05	ACC = 0.94

PPV = 0.82

Table 7.1 The number of matches correctly and incorrectly estimated by a feature matching algorithm, showing the number of true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN). The columns sum up to the actual number of positives (P) and negatives (N), while the rows sum up to the predicted number of positives (P') and negatives (N'). The formulas for the true positive rate (TPR), the false positive rate (FPR), the positive predictive value (PPV), and the accuracy (ACC) are given in the text.

first counting the number of true and false matches and match failures, using the following definitions (Fawcett 2006), which we already discussed in Section 6.3.3:

- **TP**: true positives, i.e., number of correct matches;
- **FN**: false negatives, matches that were not correctly detected;
- **FP**: false positives, proposed matches that are incorrect;
- **TN**: true negatives, non-matches that were correctly rejected.

Table 7.1 shows a sample *confusion matrix* (contingency table) containing such numbers.

We can convert these numbers into *unit rates* by defining the following quantities (Fawcett 2006):

- true positive rate (TPR),

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{TP}}{P}; \quad (7.14)$$

- false positive rate (FPR),

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} = \frac{\text{FP}}{N}; \quad (7.15)$$

- positive predictive value (PPV),

$$\text{PPV} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\text{TP}}{P'}; \quad (7.16)$$

- accuracy (ACC),

$$\text{ACC} = \frac{\text{TP} + \text{TN}}{P + N}. \quad (7.17)$$

In the *information retrieval* (or document retrieval) literature (Baeza-Yates and Ribeiro-Neto 1999; Manning, Raghavan, and Schütze 2008), the term *precision* (how many returned documents are relevant) is used instead of PPV and *recall* (what fraction of relevant documents was found) is used instead of TPR (see also Section 6.3.3). The precision and recall can be combined into a single measure called the *F-score*, which is their harmonic mean. This single measure is often used to rank vision algorithms (Knapitsch, Park *et al.* 2017).

Any particular matching strategy (at a particular threshold or parameter setting) can be rated by the TPR and FPR numbers; ideally, the true positive rate will be close to 1 and the false positive rate close to 0. As we vary the matching threshold, we obtain a family of such points, which are collectively known as the *receiver operating characteristic (ROC) curve* (Fawcett 2006) (Figure 7.22a). The closer this curve lies to the upper left corner, i.e., the larger the area under the curve (AUC), the better its performance. Figure 7.22b shows how we can plot the number of matches and non-matches as a function of inter-feature distance d . These curves can then be used to plot an ROC curve (Exercise 7.3). The ROC curve can also be used to calculate the *mean average precision*, which is the average precision (PPV) as you vary the threshold to select the best results, then the two top results, etc. (see Section 6.3.3 and Figure 6.27).

The problem with using a fixed threshold is that it is difficult to set; the useful range of thresholds can vary a lot as we move to different parts of the feature space (Lowe 2004; Mikolajczyk and Schmid 2005). A better strategy in such cases is to simply match the *nearest neighbor* in feature space. Since some features may have no matches (e.g., they may be part of background clutter in object recognition or they may be occluded in the other image), a threshold is still used to reduce the number of false positives.

Ideally, this threshold itself will adapt to different regions of the feature space. If sufficient training data is available (Hua, Brown, and Winder 2007), it is sometimes possible to learn

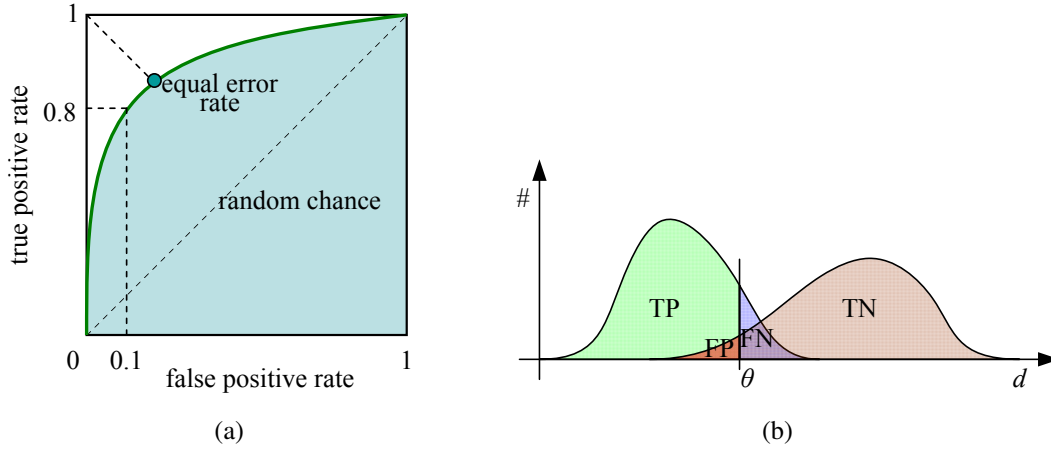


Figure 7.22 ROC curve and its related rates: (a) The ROC curve plots the true positive rate against the false positive rate for a particular combination of feature extraction and matching algorithms. Ideally, the true positive rate should be close to 1, while the false positive rate is close to 0. The area under the ROC curve (AUC) is often used as a single (scalar) measure of algorithm performance. Alternatively, the equal error rate is sometimes used. (b) The distribution of positives (matches) and negatives (non-matches) as a function of inter-feature distance d . As the threshold θ is increased, the number of true positives (TP) and false positives (FP) increases.

different thresholds for different features. Often, however, we are simply given a collection of images to match, e.g., when stitching images or constructing 3D models from unordered photo collections (Brown and Lowe 2007, 2005; Snavely, Seitz, and Szeliski 2006). In this case, a useful heuristic can be to compare the nearest neighbor distance to that of the second nearest neighbor, preferably taken from an image that is known not to match the target (e.g., a different object in the database) (Brown and Lowe 2002; Lowe 2004; Mishkin, Matas, and Perdoch 2015). We can define this *nearest neighbor distance ratio* (Mikolajczyk and Schmid 2005) as

$$\text{NNDR} = \frac{d_1}{d_2} = \frac{\|D_A - D_B\|}{\|D_A - D_C\|}, \quad (7.18)$$

where d_1 and d_2 are the nearest and second nearest neighbor distances, D_A is the target descriptor, and D_B and D_C are its closest two neighbors (Figure 7.23). Recent work has shown that mutual NNDR (or, at least NNDR with cross-consistency check) work noticeably better than one-way NNDR (Bellavia and Colombo 2020; Jin, Mishkin *et al.* 2021).

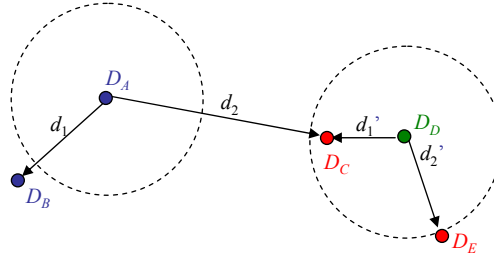


Figure 7.23 Fixed threshold, nearest neighbor, and nearest neighbor distance ratio matching. At a fixed distance threshold (dashed circles), descriptor D_A fails to match D_B and D_D incorrectly matches D_C and D_E . If we pick the nearest neighbor, D_A correctly matches D_B but D_D incorrectly matches D_C . Using nearest neighbor distance ratio (NNDR) matching, the small NNDR d_1/d_2 correctly matches D_A with D_B , and the large NNDR d'_1/d'_2 correctly rejects matches for D_D .

Efficient matching

Once we have decided on a matching strategy, we still need to efficiently search for potential candidates. The simplest way to find all corresponding feature points is to compare all features against all other features in each pair of potentially matching images. While traditionally this has been too computationally expensive, modern GPUs have enabled such comparisons.

A more efficient approach is to devise an *indexing structure*, such as a multi-dimensional search tree or a hash table, to rapidly search for features near a given feature. Such indexing structures can either be built for each image independently (which is useful if we want to only consider certain potential matches, e.g., searching for a particular object) or globally for all the images in a given database, which can potentially be faster, since it removes the need to iterate over each image. For extremely large databases (millions of images or more), even more efficient structures based on ideas from document retrieval, e.g., *vocabulary trees* (Nistér and Stewénius 2006), *product quantization* (Jégou, Douze, and Schmid 2010; Johnson, Douze, and Jégou 2021), or an *inverted multi-index* (Babenko and Lempitsky 2015b) can be used, as discussed in Section 7.1.4.

One of the simpler techniques to implement is multi-dimensional hashing, which maps descriptors into fixed size buckets based on some function applied to each descriptor vector. At matching time, each new feature is hashed into a bucket, and a search of nearby buckets is used to return potential candidates, which can then be sorted or graded to determine which are valid matches.

A simple example of hashing is the Haar wavelets used by Brown, Szeliski, and Winder

(2005) in their MOPS paper. During the matching structure construction, each 8×8 scaled, oriented, and normalized MOPS patch is converted into a three-element index by performing sums over different quadrants of the patch. The resulting three values are normalized by their expected standard deviations and then mapped to the two (of $b = 10$) nearest 1D bins. The three-dimensional indices formed by concatenating the three quantized values are used to index the $2^3 = 8$ bins where the feature is stored (added). At query time, only the primary (closest) indices are used, so only a single three-dimensional bin needs to be examined. The coefficients in the bin can then be used to select k approximate nearest neighbors for further processing (such as computing the NNDR).

A more complex, but more widely applicable, version of hashing is called *locality sensitive hashing*, which uses unions of independently computed hashing functions to index the features (Gionis, Indyk, and Motwani 1999; Shakhnarovich, Darrell, and Indyk 2006). Shakhnarovich, Viola, and Darrell (2003) extend this technique to be more sensitive to the distribution of points in parameter space, which they call *parameter-sensitive hashing*. More recent work converts high-dimensional descriptor vectors into binary codes that can be compared using Hamming distances (Torralba, Weiss, and Fergus 2008; Weiss, Torralba, and Fergus 2008) or that can accommodate arbitrary kernel functions (Kulis and Grauman 2009; Raginsky and Lazebnik 2009).

Another widely used class of indexing structures are multi-dimensional search trees. The best known of these are *k-d trees*, also often written as *kd-trees*, which divide the multi-dimensional feature space along alternating axis-aligned hyperplanes, choosing the threshold along each axis so as to maximize some criterion, such as the search tree balance (Samet 1989). Figure 7.24 shows an example of a two-dimensional k-d tree. Here, eight different data points A–H are shown as small diamonds arranged on a two-dimensional plane. The k-d tree recursively splits this plane along axis-aligned (horizontal or vertical) cutting planes. Each split can be denoted using the dimension number and split value (Figure 7.24b). The splits are arranged so as to try to balance the tree, i.e., to keep its maximum depth as small as possible. At query time, a classic k-d tree search first locates the query point (+) in its appropriate bin (D), and then searches nearby leaves in the tree (C, B, ...) until it can guarantee that the nearest neighbor has been found. The best bin first (BBF) search (Beis and Lowe 1999) searches bins in order of their spatial proximity to the query point and is therefore usually more efficient.

Many additional data structures have been developed for solving exact and approximate nearest neighbor problems (Arya, Mount *et al.* 1998; Liang, Liu *et al.* 2001; Hjaltason and Samet 2003). For example, Nene and Nayar (1997) developed a technique they call *slicing* that uses a series of 1D binary searches on the point list sorted along different dimensions

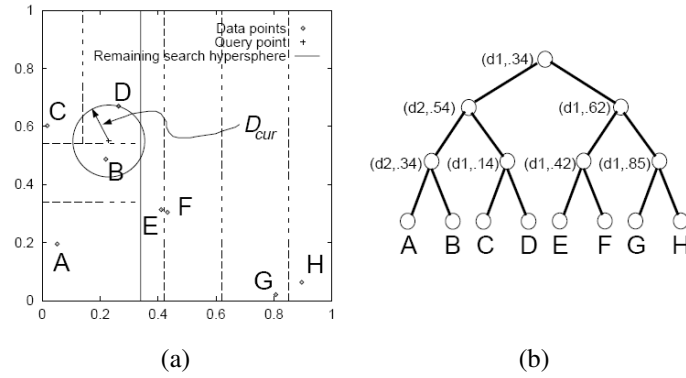


Figure 7.24 *K-d tree and best bin first (BBF) search (Beis and Lowe 1999) © 1999 IEEE:* (a) The spatial arrangement of the axis-aligned cutting planes is shown using dashed lines. Individual data points are shown as small diamonds. (b) The same subdivision can be represented as a tree, where each interior node represents an axis-aligned cutting plane (e.g., the top node cuts along dimension $d1$ at value .34) and each leaf node is a data point. During a BBF search, a query point (denoted by “+”) first looks in its containing bin (D) and then in its nearest adjacent bin (B), rather than its closest neighbor in the tree (C).

to efficiently cull down a list of candidate points that lie within a hypercube of the query point. Grauman and Darrell (2005) reweight the matches at different levels of an indexing tree, which allows their technique to be less sensitive to discretization errors in the tree construction. Nistér and Stewénius (2006) use a *metric tree*, which compares feature descriptors to a small number of prototypes at each level in a hierarchy. The resulting quantized *visual words* can then be used with classical information retrieval (document relevance) techniques to quickly winnow down a set of potential candidates from a database of millions of images (Section 7.1.4). Muja and Lowe (2009) compare a number of these approaches, introduce a new one of their own (priority search on hierarchical k-means trees), and conclude that multiple randomized k-d trees often provide the best performance. Modern libraries for computing approximate nearest neighbors include FLANN (Muja and Lowe 2014) and Faiss (Johnson, Douze, and Jégou 2021), which are discussed in Section 5.1.1 and Appendix C.2.

Feature match verification and densification

Once we have some candidate matches, we can use geometric alignment (Section 8.1) to verify which matches are *inliers* and which ones are *outliers*. For example, if we expect the whole image to be translated or rotated in the matching view, we can fit a global geometric

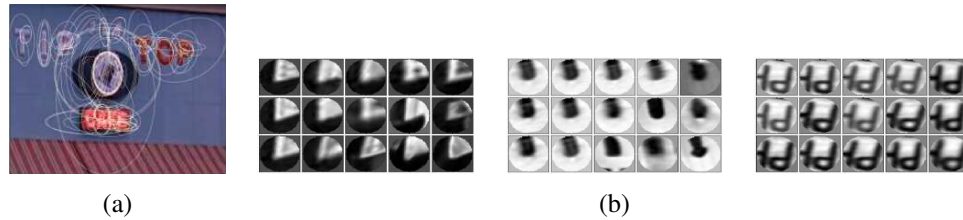


Figure 7.25 Visual words obtained from elliptical normalized affine regions (Sivic and Zisserman 2009) © 2009 IEEE. (a) Affine covariant regions are extracted from each frame and clustered into visual words using *k*-means clustering on SIFT descriptors with a learned Mahalanobis distance. (b) The central patch in each grid shows the query and the surrounding patches show the nearest neighbors.

transform and keep only those feature matches that are sufficiently close to this estimated transformation. The process of selecting a small set of seed matches and then verifying a larger set is often called *random sampling* or RANSAC (Section 8.1.4). Once an initial set of correspondences has been established, some systems look for additional matches, e.g., by looking for additional correspondences along epipolar lines (Section 12.1) or in the vicinity of estimated locations based on the global transform. It is also possible to use deep neural networks to perform feature matching and filtering, as in the SuperGlue system of Sarlin, DeTone *et al.* (2020). These topics are discussed further in Sections 8.1 and 12.2.

7.1.4 Large-scale matching and retrieval

As the number of objects in the database starts to grow (say, billions of objects or video frames), the time it takes to match a new image against each database image can become prohibitive. Instead of comparing the images one at a time, techniques are needed to quickly narrow down the search to a few likely images, which can then be compared using a more conservative verification stage.

The problem of quickly finding partial matches between documents is one of the central problems in *information retrieval* (IR) (Baeza-Yates and Ribeiro-Neto 1999; Manning, Raghavan, and Schütze 2008). In computer vision, the problem of finding a particular object in a large collection is called *content-based image retrieval* (CBIR) (Smeulders, Worring *et al.* 2000; Lew, Sebe *et al.* 2006; Vasconcelos 2007; Datta, Joshi *et al.* 2008) or *instance retrieval* (Zheng, Yang, and Tian 2018). The basic approach in fast document retrieval algorithms is to precompute an *inverted index* between individual words and the documents (or web pages or news stories) where they occur. More precisely, the *frequency* of occurrence of particular

words in a document is used to quickly find documents that match a particular query.

Sivic and Zisserman (2009) were the first to adapt IR techniques to visual search. In their Video Google system, affine invariant features are first detected in all the video frames they are indexing using both *shape adapted* regions around Harris feature points (Schaffalitzky and Zisserman 2002; Mikolajczyk and Schmid 2004) and maximally stable extremal regions (Matas, Chum *et al.* 2004; Section 7.1.1), as shown in Figure 7.25a. Next, 128-dimensional SIFT descriptors are computed from each normalized region (i.e., the patches shown in Figure 7.25b). Then, an average covariance matrix for these descriptors is estimated by accumulating statistics for features tracked from frame to frame. The feature descriptor covariance Σ is then used to define a Mahalanobis distance (5.32) between feature descriptors. In practice, feature descriptors are *whitened* by pre-multiplying them by $\Sigma^{-1/2}$ so that Euclidean distances can be used.⁵

To apply fast information retrieval techniques to images, the high-dimensional feature descriptors that occur in each image must first be mapped into discrete *visual words*. Sivic and Zisserman (2003) perform this mapping using k-means clustering, while some of the later methods (Nistér and Stewénus 2006; Philbin, Chum *et al.* 2007) use alternative techniques, such as vocabulary trees or randomized forests. To keep the clustering time manageable, only a few hundred video frames are used to learn the cluster centers, which still involves estimating several thousand clusters from about 300,000 descriptors, although subsequent work has greatly extended this capacity (Nistér and Stewénus 2006; Philbin, Chum *et al.* 2007; Mikulik, Perdoch *et al.* 2013). At visual query time, each feature in a new query region (e.g., Figure 7.25a, which is a cropped region from a larger video frame) is mapped to its corresponding visual word. To keep very common patterns from contaminating the results, a *stop list* of the most common visual words is created and such words are dropped from further consideration.

Once a query image or region has been mapped into its constituent visual words, likely matching images must then be retrieved from the database. The exact details of how this is done can be found in Sivic and Zisserman (2009), Nistér and Stewénus (2006), Philbin, Chum *et al.* (2007), Chum, Philbin *et al.* (2007), Philbin, Chum *et al.* (2008), and also in the first edition of this book (Szeliski 2010, Section 14.3.2). Because of the high efficiency in both quantizing and scoring features, the vocabulary-tree-based recognition system built by Nistér and Stewénus (2006) was able to process incoming images in real time against a database of 40,000 CD covers and at 1Hz when matching a database of one million frames

⁵Note that the computation of feature covariances from matched feature points is much more sensible than simply performing a PCA on the descriptor space (Winder and Brown 2007). This corresponds roughly to the *within-class* scatter matrix we studied in Section 5.2.3.

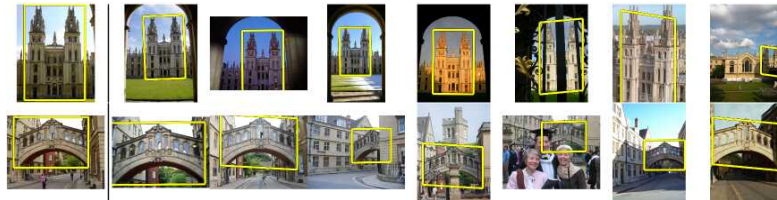


Figure 7.26 Location or building recognition using randomized trees (Philbin, Chum *et al.* 2007) © 2007 IEEE. The left image is the query, the other images are the highest-ranked results.

taken from six feature-length movies.

Instance recognition systems continued to improve rapidly in the 2000s. Philbin, Chum *et al.* (2007) showed that randomized forest of k-d trees perform better than vocabulary trees on a large location recognition task (Figure 7.26). They also compared the effects of using different 2D motion models (Section 2.1.1) in the verification stage. In follow-on work, Chum, Philbin *et al.* (2007) applied another idea from information retrieval, namely *query expansion*, which involves re-submitting top-ranked images from the initial query as additional queries to generate additional candidate results.⁶ Philbin, Chum *et al.* (2008) showed how to mitigate quantization problems in visual words selection using *soft assignment*, where each feature descriptor is mapped to a number of nearby visual words, which is similar to the multiple assignment idea proposed earlier by Jégou, Harzallah, and Schmid (2007). However, such techniques tend to reduce the sparsity of visual word vectors and increase the memory and computation costs. Jégou, Douze, and Schmid (2008) incorporated partial geometrical information and an explicit matching scheme between local descriptors in the initial large-scale image ranking stage. Taken together, these algorithms helped instance recognition algorithms perform Web-scale retrieval, matching, 3D reconstruction tasks (Agarwal, Furukawa *et al.* 2010, 2011; Frahm, Fite-Georgel *et al.* 2010; Snavely, Simon *et al.* 2010).

Since the “deep learning revolution” in 2012, researchers have started developing neural feature detectors and descriptors (Sections 7.1.1 and 7.1.2) and sometimes combining them into end-to-end matching systems.⁷ Figure 7.27 shows some of the major milestones in instance retrieval, while Figure 7.28 shows the variety of different classic and CNN-based retrieval architectures that have been considered. The survey paper by Zheng, Yang, and Tian (2018) describes and contrasts these various algorithms in more detail and also provides an

⁶An alternative to query expansion is *database-side augmentation* (Arandjelović and Zisserman 2012).

⁷But note that some popular open-source large-scale reconstruction systems such as COLMAP still use traditional features and indexing schemes (Schönberger and Frahm 2016).

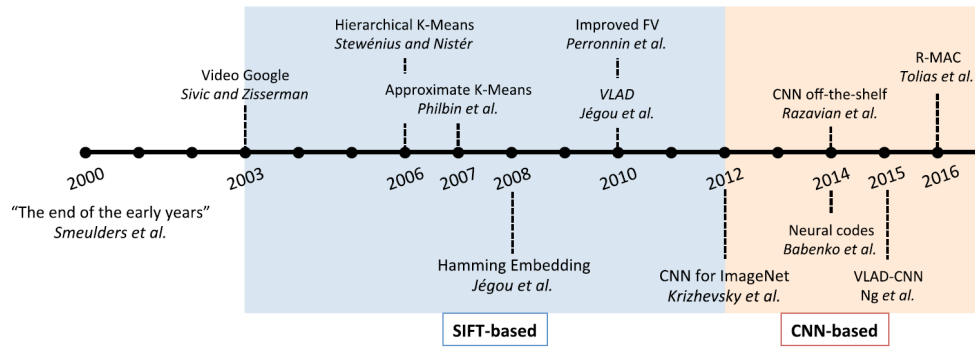


Figure 7.27 Milestones in instance retrieval (Zheng, Yang, and Tian 2018) © 2018 IEEE, showing the shift from hand-crafted feature-based retrieval to CNN-based approaches.

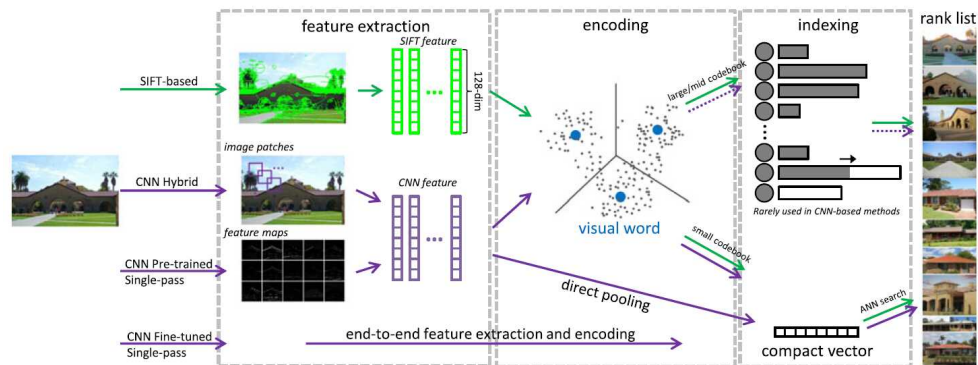


Figure 7.28 Typical pipeline for feature-based instance retrieval (Zheng, Yang, and Tian 2018) © 2018 IEEE, showing the feature extraction, encoding, and indexing portions, which are often collapsed when using a deep learning framework.

experimental comparison of some of these algorithms on image retrieval datasets. You can also find more details on related techniques and systems in Section 6.2.3 on visual similarity search, which discusses global descriptors that represent an image with a single vector (Arandjelovic, Gronat *et al.* 2016; Radenović, Tolias, and Chum 2019; Yang, Kien Nguyen *et al.* 2019; Cao, Araujo, and Sim 2020; Ng, Balntas *et al.* 2020; Tolias, Jenicek, and Chum 2020) as alternatives to bags of local features, Section 11.2.3 on location recognition, and Section 11.4.6 on large-scale 3D reconstruction from community (internet) photos.

7.1.5 Feature tracking

An alternative to independently finding features in all candidate images and then matching them is to find a set of likely feature locations in a first image and to then *search* for their corresponding locations in subsequent images. This kind of *detect then track* approach is more widely used for video tracking applications, where the expected amount of motion and appearance deformation between adjacent frames is expected to be small.

The process of selecting good features to track is closely related to selecting good features for more general recognition applications. In practice, regions containing high gradients in both directions, i.e., which have high eigenvalues in the auto-correlation matrix (7.8), provide stable locations at which to find correspondences (Shi and Tomasi 1994).

In subsequent frames, searching for locations where the corresponding patch has low squared difference (7.1) often works well enough. However, if the images are undergoing brightness change, explicitly compensating for such variations (9.9) or using *normalized cross-correlation* (9.11) may be preferable. If the search range is large, it is also often more efficient to use a *hierarchical* search strategy, which uses matches in lower-resolution images to provide better initial guesses and hence speed up the search (Section 9.1.1). Alternatives to this strategy involve learning what the appearance of the patch being tracked should be and then searching for it in the vicinity of its predicted position (Avidan 2001; Jurie and Dhome 2002; Williams, Blake, and Cipolla 2003). These topics are all covered in more detail in Section 9.1.3.

If features are being tracked over longer image sequences, their appearance can undergo larger changes. You then have to decide whether to continue matching against the originally detected patch (feature) or to re-sample each subsequent frame at the matching location. The former strategy is prone to failure, as the original patch can undergo appearance changes such as foreshortening. The latter runs the risk of the feature drifting from its original location to some other location in the image (Shi and Tomasi 1994). (Mathematically, small misregistration errors compound to create a *Markov random walk*, which leads to larger drift over time.)



Figure 7.29 Feature tracking using an affine motion model (Shi and Tomasi 1994) © 1994 IEEE, Top row: image patch around the tracked feature location. Bottom row: image patch after warping back toward the first frame using an affine deformation. Even though the speed sign gets larger from frame to frame, the affine transformation maintains a good resemblance between the original and subsequent tracked frames.

A preferable solution is to compare the original patch to later image locations using an *affine* motion model (Section 9.2). Shi and Tomasi (1994) first compare patches in neighboring frames using a translational model and then use the location estimates produced by this step to initialize an affine registration between the patch in the current frame and the base frame where a feature was first detected (Figure 7.29). In their system, features are only detected infrequently, i.e., only in regions where tracking has failed. In the usual case, an area around the current *predicted* location of the feature is searched with an incremental registration algorithm (Section 9.1.3). The resulting tracker is often called the Kanade–Lucas–Tomasi (KLT) tracker.

Since their original work on feature tracking, Shi and Tomasi’s approach has generated a plethora of follow-on papers and applications. Beardsley, Torr, and Zisserman (1996) use extended feature tracking combined with structure from motion (Chapter 11) to incrementally build up sparse 3D models from video sequences. Kang, Szeliski, and Shum (1997) tie together the corners of adjacent (regularly gridded) patches to provide some additional stability to the tracking, at the cost of poorer handling of occlusions. Tommasini, Fusiello *et al.* (1998) provide a better spurious match rejection criterion for the basic Shi and Tomasi algorithm, Collins and Liu (2003) provide improved mechanisms for feature selection and dealing with larger appearance changes over time, and Shafique and Shah (2005) develop algorithms for feature matching (data association) for videos with large numbers of moving objects or points. Lepetit and Fua (2005) and Yilmaz, Javed, and Shah (2006) survey the larger field of object tracking, which includes not only feature-based techniques but also alternative techniques based on contour and region (Section 7.3).



Figure 7.30 *Real-time head tracking using fast trained classifiers (Lepetit, Pilet, and Fua 2004) © 2004 IEEE.*

A more recent development in feature tracking is the use of learning algorithms to build special-purpose recognizers to rapidly search for matching features anywhere in an image (Lepetit, Pilet, and Fua 2006; Hinterstoisser, Benhimane *et al.* 2008; Rogez, Rihan *et al.* 2008; Özuysal, Calonder *et al.* 2010). By taking the time to train classifiers on sample patches and their affine deformations, extremely fast and reliable feature detectors can be constructed, which enables much faster motions to be supported (Figure 7.30). Coupling such features to deformable models (Pilet, Lepetit, and Fua 2008) or structure-from-motion algorithms (Klein and Murray 2008) can result in even higher stability.

While feature-based tracking is still widely used in real-time applications such as SLAM, autonomous navigation, and augmented reality (Section 11.5), a lot of current work on tracking is focused on whole *object tracking* (Chellappa, Sankaranarayanan *et al.* 2010; Smeulders, Chu *et al.* 2014), which we study in more detail in Section 9.4.4.

7.1.6 Application: Performance-driven animation

One of the most compelling applications of fast feature tracking is *performance-driven animation*, i.e., the interactive deformation of a 3D graphics model based on tracking a user's motions (Williams 1990; Litwinowicz and Williams 1994; Lepetit, Pilet, and Fua 2004).

Buck, Finkelstein *et al.* (2000) present a system that tracks a user's facial expressions and head motions and then uses them to morph among a series of hand-drawn sketches. An animator first extracts the eye and mouth regions of each sketch and draws control lines over each image (Figure 7.31a). At run time, a face-tracking system (Toyama 1998) determines the current location of these features (Figure 7.31b). The animation system decides which



Figure 7.31 Performance-driven, hand-drawn animation (Buck, Finkelstein et al. 2000) © 2000 ACM: (a) eye and mouth portions of hand-drawn sketch with their overlaid control lines; (b) an input video frame with the tracked features overlaid; (c) a different input video frame along with its (d) corresponding hand-drawn animation.

input images to morph based on nearest neighbor feature appearance matching and triangular barycentric interpolation. It also computes the global location and orientation of the head from the tracked features. The resulting morphed eye and mouth regions are then composited back into the overall head model to yield a frame of hand-drawn animation (Figure 7.31d).

In more recent work, Barnes, Jacobs *et al.* (2008) watch users animate paper cutouts on a desk and then turn the resulting motions and drawings into seamless 2D animations. Feature-based facial trackers continue to be widely used (Zollhöfer, Thies *et al.* 2018), both in the visual effects industry, as well as for real-time smartphone augmented reality effects such as Facebook’s Spark AR Face Masks.

7.2 Edges and contours

While interest points are useful for finding image locations that can be accurately matched in 2D, edge points are far more plentiful and often carry important semantic associations. For example, the boundaries of objects, which also correspond to occlusion events in 3D, are usually delineated by visible contours. Other kinds of edges correspond to shadow boundaries or crease edges, where surface orientation changes rapidly. Isolated edge points can also be grouped into longer *curves* or *contours*, as well as *straight line segments* (Section 7.4). It is interesting that even young children have no difficulty in recognizing familiar objects or animals from such simple line drawings.

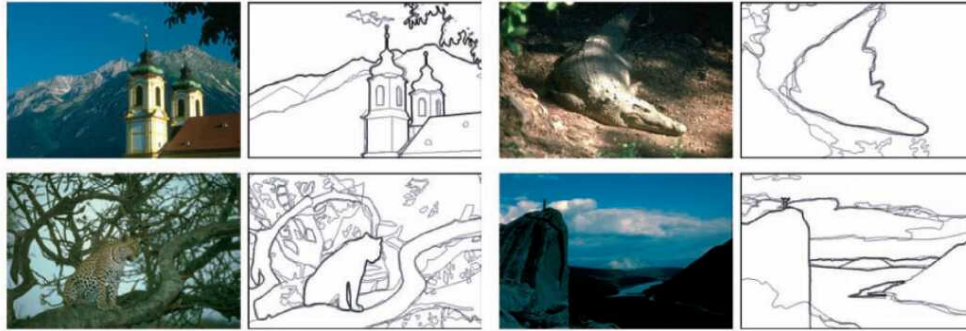


Figure 7.32 Human boundary detection (Martin, Fowlkes, and Malik 2004) © 2004 IEEE. The darkness of the edges corresponds to how many human subjects marked an object boundary at that location.

7.2.1 Edge detection

Given an image, how can we find the salient edges? Consider the color images in Figure 7.32. If someone asked you to point out the most “salient” or “strongest” edges or the object boundaries, which ones would you trace? How closely do your perceptions match the edge images shown in Figure 7.32?

Qualitatively, edges occur at boundaries between regions of different color, intensity, or texture (Martin, Fowlkes, and Malik 2004; Arbeláez, Maire *et al.* 2011; Pont-Tuset, Arbeláez *et al.* 2017). Unfortunately, segmenting an image into coherent regions is a difficult task, which we address in Section 7.5. Often, it is preferable to detect edges using only purely local information.

Under such conditions, a reasonable approach is to define an edge as a location of *rapid intensity or color variation*. Think of an image as a height field. On such a surface, edges occur at locations of *steep slopes*, or equivalently, in regions of closely packed contour lines (on a topographic map).

A mathematical way to define the slope and direction of a surface is through its gradient,

$$\mathbf{J}(\mathbf{x}) = \nabla I(\mathbf{x}) = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right) (\mathbf{x}). \quad (7.19)$$

The local gradient vector $\mathbf{J}$ points in the direction of *steepest ascent* in the intensity function. Its magnitude is an indication of the slope or strength of the variation, while its orientation points in a direction *perpendicular* to the local contour.

Unfortunately, taking image derivatives accentuates high frequencies and hence amplifies noise, as the proportion of noise to signal is larger at high frequencies. It is therefore prudent

to smooth the image with a low-pass filter prior to computing the gradient. Because we would like the response of our edge detector to be independent of orientation, a circularly symmetric smoothing filter is desirable. As we saw in Section 3.2, the Gaussian is the only separable circularly symmetric filter, so it is used in most edge detection algorithms. Canny (1986) discusses alternative filters and a number of researchers review alternative edge detection algorithms and compare their performance (Davis 1975; Nalwa and Binford 1986; Nalwa 1987; Deriche 1987; Freeman and Adelson 1991; Nalwa 1993; Heath, Sarkar *et al.* 1998; Crane 1997; Ritter and Wilson 2000; Bowyer, Kranenburg, and Dougherty 2001; Arbeláez, Maire *et al.* 2011; Pont-Tuset, Arbeláez *et al.* 2017).

Because differentiation is a linear operation, it commutes with other linear filtering operations. The gradient of the smoothed image can therefore be written as

$$\mathbf{J}_\sigma(\mathbf{x}) = \nabla[G_\sigma(\mathbf{x}) * I(\mathbf{x})] = [\nabla G_\sigma](\mathbf{x}) * I(\mathbf{x}), \quad (7.20)$$

i.e., we can convolve the image with the horizontal and vertical derivatives of the Gaussian kernel function,

$$\nabla G_\sigma(\mathbf{x}) = \left(\frac{\partial G_\sigma}{\partial x}, \frac{\partial G_\sigma}{\partial y} \right) (\mathbf{x}) = [-x \ -y] \frac{1}{\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right), \quad (7.21)$$

where the parameter σ indicates the width of the Gaussian. This is the same computation that is performed by Freeman and Adelson's (1991) first-order steerable filter, which we have already covered in Section 3.2.3.

For many applications, however, we wish to thin such a continuous gradient image to return isolated edges only, i.e., as single pixels at discrete locations along the edge contours. This can be achieved by looking for *maxima* in the edge strength (gradient magnitude) in a direction *perpendicular* to the edge orientation, i.e., along the gradient direction.

Finding this maximum corresponds to taking a directional derivative of the strength field in the direction of the gradient and then looking for zero crossings. The desired directional derivative is equivalent to the dot product between a second gradient operator and the results of the first,

$$S_\sigma(\mathbf{x}) = \nabla \cdot \mathbf{J}_\sigma(\mathbf{x}) = [\nabla^2 G_\sigma](\mathbf{x}) * I(\mathbf{x}). \quad (7.22)$$

The gradient operator dot product with the gradient is called the *Laplacian*. The convolution kernel

$$\nabla^2 G_\sigma(\mathbf{x}) = \left(\frac{x^2 + y^2}{\sigma^4} - \frac{2}{\sigma^2} \right) G_\sigma(\mathbf{x}), \quad (7.23)$$

is therefore called the *Laplacian of Gaussian* (LoG) kernel (Marr and Hildreth 1980). This kernel can be split into two separable parts,

$$\nabla^2 G_\sigma(\mathbf{x}) = \left(\frac{x^2}{2\sigma^4} - \frac{1}{\sigma^2} \right) G_\sigma(x)G_\sigma(y) + \left(\frac{y^2}{2\sigma^4} - \frac{1}{\sigma^2} \right) G_\sigma(y)G_\sigma(x) \quad (7.24)$$

(Wiejak, Buxton, and Buxton 1985), which allows for a much more efficient implementation using separable filtering (Section 3.2.1).

In practice, it is quite common to replace the Laplacian of Gaussian convolution with a difference of Gaussian (DoG) computation, since the kernel shapes are qualitatively similar (Figure 3.34). This is especially convenient if a “Laplacian pyramid” (Section 3.5) has already been computed.⁸

In fact, it is not strictly necessary to take differences between adjacent levels when computing the edge field. Think about what a zero crossing in a “generalized” difference of Gaussians image represents. The finer (smaller kernel) Gaussian is a noise-reduced version of the original image. The coarser (larger kernel) Gaussian is an estimate of the average intensity over a larger region. Thus, whenever the DoG image changes sign, this corresponds to the (slightly blurred) image going from relatively darker to relatively lighter, as compared to the average intensity in that neighborhood.

Once we have computed the sign function $S(\mathbf{x})$, we must find its *zero crossings* and convert these into edge elements (*edgels*). An easy way to detect and represent zero crossings is to look for adjacent pixel locations $\mathbf{x}_i$ and $\mathbf{x}_j$ where the sign changes value, i.e., $[S(\mathbf{x}_i) > 0] \neq [S(\mathbf{x}_j) > 0]$.

The sub-pixel location of this crossing can be obtained by computing the “ x -intercept” of the “line” connecting $S(\mathbf{x}_i)$ and $S(\mathbf{x}_j)$,

$$\mathbf{x}_z = \frac{\mathbf{x}_i S(\mathbf{x}_j) - \mathbf{x}_j S(\mathbf{x}_i)}{S(\mathbf{x}_j) - S(\mathbf{x}_i)}. \quad (7.25)$$

The orientation and strength of such edgels can be obtained by linearly interpolating the gradient values computed on the original pixel grid.

An alternative edgel representation can be obtained by linking adjacent edgels on the dual grid to form edgels that live *inside* each square formed by four adjacent pixels in the original pixel grid.⁹ The advantage of this representation is that the edgels now live on a grid offset by half a pixel from the original pixel grid and are thus easier to store and access. As before, the orientations and strengths of the edges can be computed by interpolating the gradient field or estimating these values from the difference of Gaussian image (see Exercise 7.7).

In applications where the accuracy of the edge orientation is more important, higher-order steerable filters can be used (Freeman and Adelson 1991) (see Section 3.2.3). Such filters are more selective for more elongated edges and also have the possibility of better modeling curve

⁸Recall that Burt and Adelson’s (1983a) “Laplacian pyramid” actually computes differences of Gaussian-filtered levels.

⁹This algorithm is a 2D version of the 3D *marching cubes* isosurface extraction algorithm (Lorensen and Cline 1987).

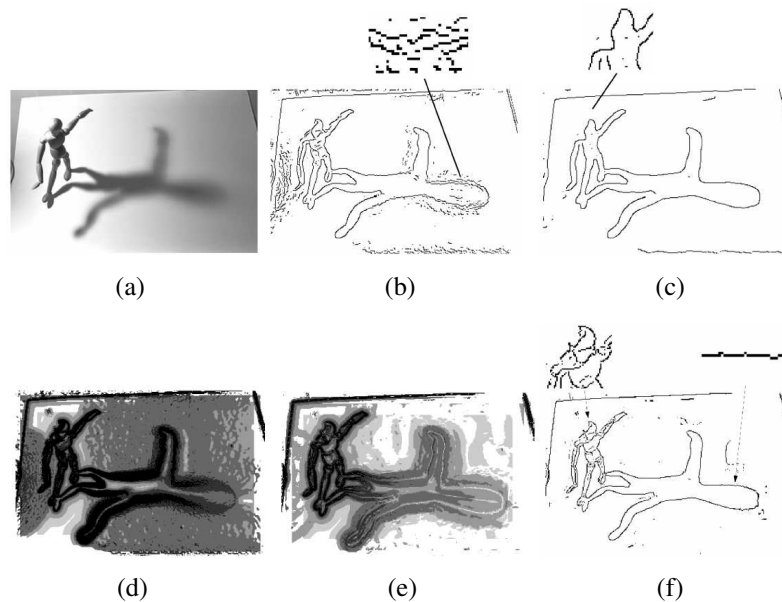


Figure 7.33 Scale selection for edge detection (Elder and Zucker 1998) © 1998 IEEE: (a) original image; (b–c) Canny/Deriche edge detector tuned to the finer (mannequin) and coarser (shadow) scales; (d) minimum reliable scale for gradient estimation; (e) minimum reliable scale for second derivative estimation; (f) final detected edges.

intersections because they can represent multiple orientations at the same pixel (Figure 3.16). Their disadvantage is that they are more expensive to compute and the directional derivative of the edge strength does not have a simple closed form solution.¹⁰

Scale selection and blur estimation

As we mentioned before, the derivative, Laplacian, and Difference of Gaussian filters (7.20–7.23) all require the selection of a spatial scale parameter σ . If we are only interested in detecting sharp edges, the width of the filter can be determined from image noise characteristics (Canny 1986; Elder and Zucker 1998). However, if we want to detect edges that occur at different resolutions (Figures 7.33b–c), a *scale-space* approach that detects and then selects edges at different scales may be necessary (Witkin 1983; Lindeberg 1994, 1998a; Nielsen, Florack, and Deriche 1997).

¹⁰In fact, the edge orientation can have a 180° ambiguity for “bar edges”, which makes the computation of zero crossings in the derivative more tricky.

Elder and Zucker (1998) present a principled approach to solving this problem. Given a known image noise level, their technique computes, for every pixel, the minimum scale at which an edge can be reliably detected (Figure 7.33d). Their approach first computes gradients densely over an image by selecting among gradient estimates computed at different scales, based on their gradient magnitudes. It then performs a similar estimate of minimum scale for directed second derivatives and uses zero crossings of this latter quantity to robustly select edges (Figures 7.33e–f). As an optional final step, the blur width of each edge can be computed from the distance between extrema in the second derivative response minus the width of the Gaussian filter.

Color edge detection

While most edge detection techniques have been developed for grayscale images, color images can provide additional information. For example, noticeable edges between *iso-luminant* colors (colors that have the same luminance) are useful cues but fail to be detected by grayscale edge operators.

One simple approach is to combine the outputs of grayscale detectors run on each color band separately.¹¹ However, some care must be taken. For example, if we simply sum up the gradients in each of the color bands, the signed gradients may actually cancel each other! (Consider, for example a pure red-to-green edge.) We could also detect edges independently in each band and then take the union of these, but this might lead to thickened or doubled edges that are hard to link.

A better approach is to compute the *oriented energy* in each band (Morrone and Burr 1988; Perona and Malik 1990a), e.g., using a second-order steerable filter (Section 3.2.3) (Freeman and Adelson 1991), and then sum up the orientation-weighted energies and find their joint best orientation. Unfortunately, the directional derivative of this energy may not have a closed form solution (as in the case of signed first-order steerable filters), so a simple zero crossing-based strategy cannot be used. However, the technique described by Elder and Zucker (1998) can be used to compute these zero crossings numerically instead.

An alternative approach is to estimate local color statistics in regions around each pixel (Ruzon and Tomasi 2001; Martin, Fowlkes, and Malik 2004). This has the advantage that more sophisticated techniques (e.g., 3D color histograms) can be used to compare regional statistics and that additional measures, such as texture, can also be considered. Figure 7.34 shows the output of such detectors.

¹¹Instead of using the raw RGB space, a more perceptually uniform color space such as $L^*a^*b^*$ (see Section 2.3.2) can be used instead. When trying to match human performance (Martin, Fowlkes, and Malik 2004), this makes sense. However, in terms of the physics of the underlying image formation and sensing, it may be a questionable strategy.

Over the years, many other approaches have been developed for detecting color edges, dating back to early work by Nevatia (1977). Ruzon and Tomasi (2001) and Gevers, van de Weijer, and Stokman (2006) provide good reviews of these approaches, which include ideas such as fusing outputs from multiple channels, using multidimensional gradients, and vector-based methods.

Combining edge feature cues

If the goal of edge detection is to match human *boundary detection* performance (Bowyer, Kranenburg, and Dougherty 2001; Martin, Fowlkes, and Malik 2004; Arbeláez, Maire *et al.* 2011; Pont-Tuset, Arbeláez *et al.* 2017), as opposed to simply finding stable features for matching, even better detectors can be constructed by combining multiple low-level cues such as brightness, color, and texture.

Martin, Fowlkes, and Malik (2004) describe a system that combines brightness, color, and texture edges to produce state-of-the-art performance on a database of hand-segmented natural color images (Martin, Fowlkes *et al.* 2001). First, they construct and train separate oriented half-disc detectors for measuring significant differences in brightness (luminance), color (a^* and b^* channels, summed responses), and texture (un-normalized filter bank responses from the work of Malik, Belongie *et al.* (2001)). Some of the responses are then sharpened using a soft non-maximal suppression technique. Finally, the outputs of the three detectors are combined using a variety of machine-learning techniques, from which logistic regression is found to have the best tradeoff between speed, space, and accuracy. The resulting system (see Figure 7.34 for some examples) is shown to outperform previously developed techniques. Maire, Arbelaez *et al.* (2008) improve on these results by combining the detector based on local appearance with a *spectral* (segmentation-based) detector (Belongie and Malik 1998). In follow-on work, Arbeláez, Maire *et al.* (2011) build a hierarchical segmentation on top of this edge detector using a variant of the watershed algorithm.

7.2.2 Contour detection

While isolated edges can be useful for a variety of applications, such as line detection (Section 7.4) and sparse stereo matching (Section 12.2), they become even more useful when linked into continuous contours.

If the edges have been detected using zero crossings of some function, linking them up is straightforward, since adjacent edgels share common endpoints. Linking the edgels into chains involves picking up an unlinked edgel and following its neighbors in both directions. Either a sorted list of edgels (sorted first by x coordinates and then by y coordinates, for



Figure 7.34 Combined brightness, color, texture boundary detector (Martin, Fowlkes, and Malik 2004) © 2004 IEEE. Successive rows show the outputs of the brightness gradient (BG), color gradient (CG), texture gradient (TG), and combined (BG+CG+TG) detectors. The final row shows human-labeled boundaries derived from a database of hand-segmented images (Martin, Fowlkes et al. 2001).

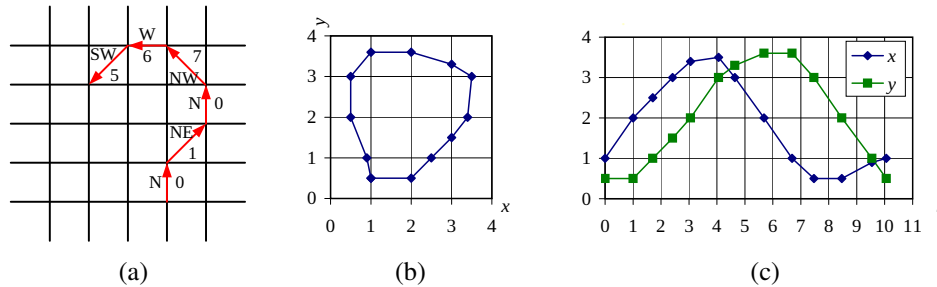


Figure 7.35 Some coding alternatives for linked contours. (a) A chain code representation of a grid-aligned linked edge chain. The code is represented as a series of direction codes, e.g., 0 1 0 7 6 5, which can further be compressed using predictive and run-length coding. (b–c) Arc-length parameterization of a contour. Discrete points along the contour (b) are first transcribed as (c) (x, y) pairs along the arc length s . This curve can then be regularly re-sampled or converted into alternative (e.g., Fourier) representations.

example) or a 2D array can be used to accelerate the neighbor finding. If edges were not detected using zero crossings, finding the continuation of an edgel can be tricky. In this case, comparing the orientation (and, optionally, phase) of adjacent edgels can be used for disambiguation. Ideas from connected component computation can also sometimes be used to make the edge linking process even faster (see Exercise 7.8).

Once the edgels have been linked into chains, we can apply an optional thresholding with hysteresis to remove low-strength contour segments (Canny 1986). The basic idea of hysteresis is to set two different thresholds and allow a curve being tracked above the higher threshold to dip in strength down to the lower threshold.

Linked edgel lists can be encoded more compactly using a variety of alternative representations. A *chain code* encodes a list of connected points lying on an $\mathcal{N}_8$ grid using a three-bit code corresponding to the eight cardinal directions (N, NE, E, SE, S, SW, W, NW) between a point and its successor (Figure 7.35a). While this representation is more compact than the original edgel list (especially if predictive variable-length coding is used), it is not very suitable for further processing.

A more useful representation is the *arc length parameterization* of a contour, $\mathbf{x}(s)$, where s denotes the arc length along a curve. Consider the linked set of edgels shown in Figure 7.35b. We start at one point (the dot at (1.0, 0.5) in Figure 7.35c) and plot it at coordinate $s = 0$ (Figure 7.35c). The next point at (2.0, 0.5) gets plotted at $s = 1$, and the next point at (2.5, 1.0) gets plotted at $s = 1.7071$, i.e., we increment s by the length of each edge segment. The resulting plot can be resampled on a regular (say, integral) s grid before further

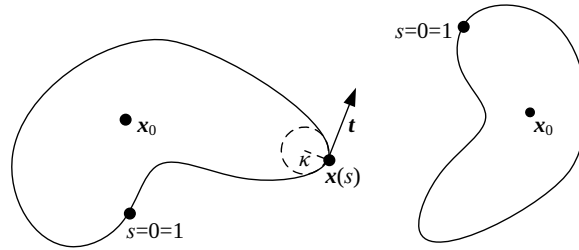


Figure 7.36 Matching two contours using their arc-length parameterization. If both curves are normalized to unit length, $s \in [0, 1]$ and centered around their centroid $\mathbf{x}_0$, they will have the same descriptor up to an overall “temporal” shift (due to different starting points for $s = 0$) and a phase (x - y) shift (due to rotation).

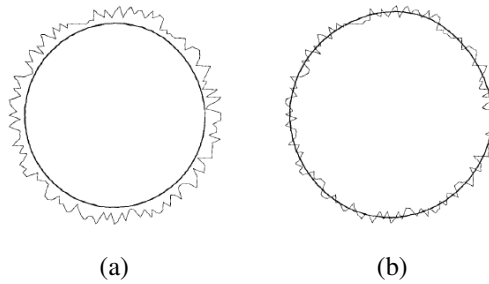


Figure 7.37 Curve smoothing with a Gaussian kernel (Lowe 1988) © 1998 IEEE: (a) without a shrinkage correction term; (b) with a shrinkage correction term.

processing.

The advantage of the arc-length parameterization is that it makes matching and processing (e.g., smoothing) operations much easier. Consider the two curves describing similar shapes shown in Figure 7.36. To compare the curves, we first subtract the average values $\mathbf{x}_0 = \int_s \mathbf{x}(s)$ from each descriptor. Next, we rescale each descriptor so that s goes from 0 to 1 instead of 0 to S , i.e., we divide $\mathbf{x}(s)$ by S . Finally, we take the Fourier transform of each normalized descriptor, treating each $\mathbf{x} = (x, y)$ value as a complex number. If the original curves are the same (up to an unknown scale and rotation), the resulting Fourier transforms should differ only by a scale change in magnitude plus a constant complex phase shift, due to rotation, and a linear phase shift in the domain, due to different starting points for s (see Exercise 7.9).

Arc-length parameterization can also be used to smooth curves to remove digitization

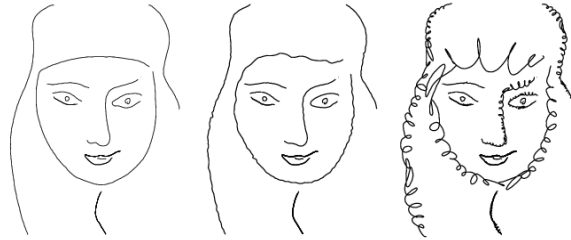


Figure 7.38 *Changing the character of a curve without affecting its sweep (Finkelstein and Salesin 1994) © 1994 ACM: higher frequency wavelets can be replaced with exemplars from a style library to effect different local appearances.*

noise. However, if we just apply a regular smoothing filter, the curve tends to shrink on itself (Figure 7.37a). Lowe (1989) and Taubin (1995) describe techniques that compensate for this shrinkage by adding an offset term based on second derivative estimates or a larger smoothing kernel (Figure 7.37b). An alternative approach, based on selectively modifying different frequencies in a wavelet decomposition, is presented by Finkelstein and Salesin (1994). In addition to controlling shrinkage without affecting its “sweep”, wavelets allow the “character” of a curve to be interactively modified, as shown in Figure 7.38.

The evolution of curves as they are smoothed and simplified is related to “grassfire” (distance) transforms and region skeletons (Section 3.3.3) (Tek and Kimia 2003), and can be used to recognize objects based on their contour shape (Sebastian and Kimia 2005). More local descriptors of curve shape such as *shape contexts* (Belongie, Malik, and Puzicha 2002) can also be used for recognition and are potentially more robust to missing parts due to occlusions.

The field of contour detection and linking continues to evolve rapidly and now includes techniques for global contour grouping, boundary completion, and junction detection (Maire, Arbeláez *et al.* 2008), as well as grouping contours into likely regions (Arbeláez, Maire *et al.* 2011) and wide-baseline correspondence (Meltzer and Soatto 2008). Some additional papers that address contour detection include Xiaofeng and Bo (2012), Lim, Zitnick, and Dollár (2013), Dollár and Zitnick (2015), Xie and Tu (2015), and Pont-Tuset, Arbeláez *et al.* (2017).

7.2.3 Application: Edge editing and enhancement

While edges can serve as components for object recognition or features for matching, they can also be used directly for image editing.

In fact, if the edge magnitude and blur estimate are kept along with each edge, a visually

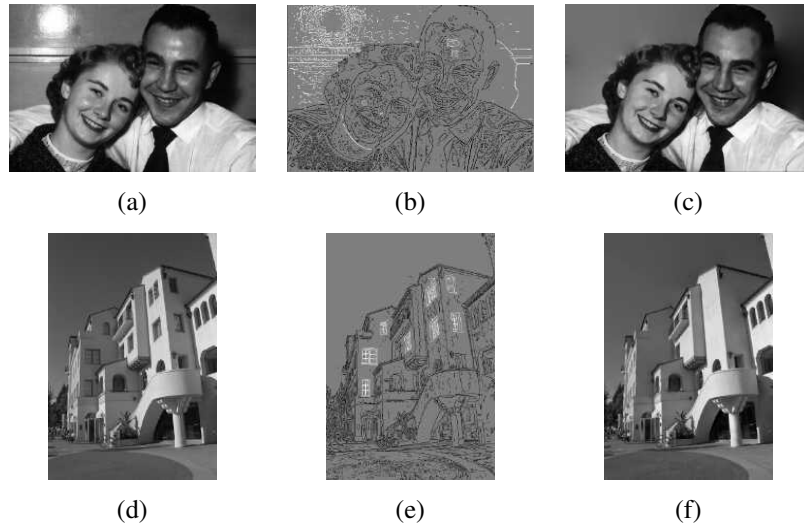


Figure 7.39 *Image editing in the contour domain (Elder and Goldberg 2001) © 2001 IEEE: (a) and (d) original images; (b) and (e) extracted edges (edges to be deleted are marked in white); (c) and (f) reconstructed edited images.*

similar image can be reconstructed from this information (Elder 1999). Based on this principle, Elder and Goldberg (2001) propose a system for “image editing in the contour domain”. Their system allows users to selectively remove edges corresponding to unwanted features such as specularities, shadows, or distracting visual elements. After reconstructing the image from the remaining edges, the undesirable visual features have been removed (Figure 7.39).

Another potential application is to enhance perceptually salient edges while simplifying the underlying image to produce a cartoon-like or “pen-and-ink” stylized image (DeCarlo and Santella 2002). This application is discussed in more detail in Section 10.5.2.

7.3 Contour tracking

While lines, vanishing points, and rectangles are commonplace in the human-made world, curves corresponding to object boundaries are even more common, especially in the natural environment. In this section, we describe some approaches to locating such boundary curves in images.

The first, originally called *snakes* by its inventors (Kass, Witkin, and Terzopoulos 1988) (Section 7.3.1), is an energy-minimizing, two-dimensional spline curve that evolves (moves)

towards image features such as strong edges. The second, *intelligent scissors* (Mortensen and Barrett 1995) (Section 7.3.1), allows the user to sketch in real time a curve that clings to object boundaries. Finally, *level set* techniques (Section 7.3.2) evolve the curve as the zero-set of a *characteristic function*, which allows them to easily change topology and incorporate region-based statistics.

All three of these are examples of *active contours* (Blake and Isard 1998; Mortensen 1999), since these boundary detectors iteratively move towards their final solution under the combination of image and optional user-guidance forces. The presentation below is heavily shortened from that presented in the first edition of this book (Szeliski 2010, Section 5.1), where interested readers can find more details.

7.3.1 Snakes and scissors

Snakes are a two-dimensional generalization of the 1D energy-minimizing splines first introduced in Section 4.2,

$$\mathcal{E}_{\text{int}} = \int \alpha(s) \|\mathbf{f}_s(s)\|^2 + \beta(s) \|\mathbf{f}_{ss}(s)\|^2 ds, \quad (7.26)$$

where s is the arc-length along the curve $\mathbf{f}(s) = (x(s), y(s))$ and $\alpha(s)$ and $\beta(s)$ are first- and second-order continuity weighting functions analogous to the $s(x, y)$ and $c(x, y)$ terms introduced in (4.24–4.25). We can discretize this energy by sampling the initial curve position evenly along its length (Figure 7.35c) to obtain

$$\begin{aligned} E_{\text{int}} = \sum_i \alpha(i) \|f(i+1) - f(i)\|^2 / h^2 \\ + \beta(i) \|f(i+1) - 2f(i) + f(i-1)\|^2 / h^4, \end{aligned} \quad (7.27)$$

where h is the step size, which can be neglected if we resample the curve along its arc-length after each iteration.

In addition to this *internal* spline energy, a snake simultaneously minimizes external image-based and constraint-based potentials. The image-based potentials are the sum of several terms

$$\mathcal{E}_{\text{image}} = w_{\text{line}} \mathcal{E}_{\text{line}} + w_{\text{edge}} \mathcal{E}_{\text{edge}} + w_{\text{term}} \mathcal{E}_{\text{term}}, \quad (7.28)$$

where the *line* term attracts the snake to dark ridges, the *edge* term attracts it to strong gradients (edges), and the *term* term attracts it to line terminations. As the snakes evolve by minimizing their energy, they often “wiggle” and “slither”, which accounts for their popular name.

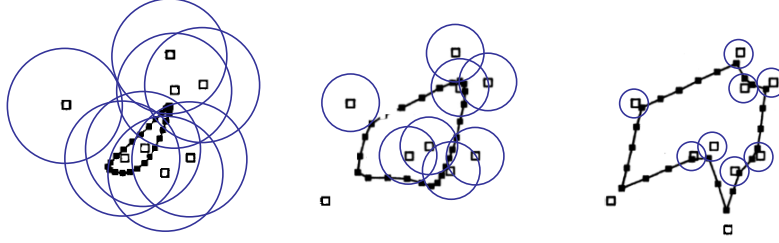


Figure 7.40 *Elastic net: The open squares indicate the cities and the closed squares linked by straight line segments are the tour points. The blue circles indicate the approximate extent of the attraction force of each city, which is reduced over time. Under the Bayesian interpretation of the elastic net, the blue circles correspond to one standard deviation of the circular Gaussian that generates each city from some unknown tour point.*

Because regular snakes have a tendency to shrink, it is usually better to initialize them by drawing the snake outside the object of interest to be tracked. Alternatively, an expansion *ballooning* force can be added to the dynamics (Cohen and Cohen 1993), essentially moving each point outwards along its normal. It is also possible to replace the energy-minimizing variational evolution equations with a deep neural network to significantly improve performance (Peng, Jiang *et al.* 2020).

Elastic nets and slippery springs

An interesting variant on snakes, first proposed by Durbin and Willshaw (1987) and later re-formulated in an energy-minimizing framework by Durbin, Szeliski, and Yuille (1989), is the *elastic net* formulation of the Traveling Salesman Problem (TSP). Recall that in a TSP, the salesman must visit each city once while minimizing the total distance traversed. A snake that is constrained to pass through each city could solve this problem (without any optimality guarantees) but it is impossible to tell ahead of time which snake control point should be associated with each city.

Instead of having a fixed constraint between snake nodes and cities, a city is assumed to pass near *some* point along the tour (Figure 7.40). In a probabilistic interpretation, each city is generated as a *mixture* of Gaussians centered at each tour point,

$$p(\mathbf{d}(j)) = \sum_i p_{ij} \quad \text{with} \quad p_{ij} = e^{-d_{ij}^2/(2\sigma^2)}, \quad (7.29)$$

where σ is the standard deviation of the Gaussian and

$$d_{ij} = \|\mathbf{f}(i) - \mathbf{d}(j)\| \quad (7.30)$$

is the Euclidean distance between a tour point $\mathbf{f}(i)$ and a city location $\mathbf{d}(j)$. The corresponding data fitting energy (negative log likelihood) is

$$E_{\text{slippery}} = - \sum_j \log p(\mathbf{d}(j)) = - \sum_j \log \left[\sum e^{-\|\mathbf{f}(i) - \mathbf{d}(j)\|^2 / 2\sigma^2} \right]. \quad (7.31)$$

This energy derives its name from the fact that, unlike a regular spring, which couples a given snake point to a given constraint, this alternative energy defines a *slippery spring* that allows the association between constraints (cities) and curve (tour) points to evolve over time (Szeliski 1989). Note that this is a soft variant of the popular *iterative closest point* data constraint that is often used in fitting or aligning surfaces to data points or to each other (Section 13.2.1) (Besl and McKay 1992; Chen and Medioni 1992; Zhang 1994).

To compute a good solution to the TSP, the slippery spring data association energy is combined with a regular first-order internal smoothness energy (7.27) to define the cost of a tour. The tour $\mathbf{f}(s)$ is initialized as a small circle around the mean of the city points and σ is progressively lowered (Figure 7.40). For large σ values, the tour tries to stay near the centroid of the points but as σ decreases each city pulls more and more strongly on its closest tour points (Durbin, Szeliski, and Yuille 1989). In the limit as $\sigma \rightarrow 0$, each city is guaranteed to capture at least one tour point and the tours between subsequent cities become straight lines.

Splines and shape priors

While snakes can be very good at capturing the fine and irregular detail in many real-world contours, they sometimes exhibit too many degrees of freedom, making it more likely that they can get trapped in local minima during their evolution.

One solution to this problem is to control the snake with fewer degrees of freedom through the use of B-spline approximations (Menet, Saint-Marc, and Medioni 1990b,a; Cipolla and Blake 1990). The resulting *B-snake* can be written as

$$\mathbf{f}(s) = \sum_k B_k(s) \mathbf{x}_k. \quad (7.32)$$

If the object being tracked or recognized has large variations in location, scale, or orientation, these can be modeled as an additional transformation on the control points, e.g., $\mathbf{x}'_k = s\mathbf{R}\mathbf{x}_k + \mathbf{t}$ (2.18), which can be estimated at the same time as the values of the control points. Alternatively, separate *detection* and *alignment* stages can be run to first localize and orient the objects of interest (Cootes, Cooper *et al.* 1995).

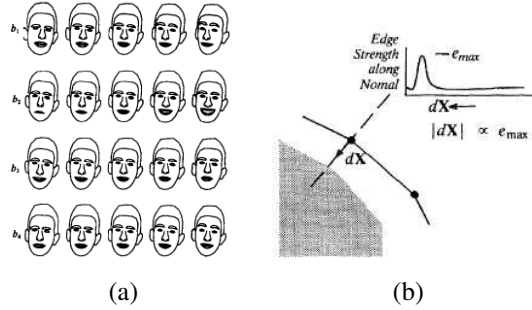


Figure 7.41 *Active Shape Model (ASM): (a) the effect of varying the first four shape parameters for a set of faces (Cootes, Taylor et al. 1993) © 1993 IEEE; (b) searching for the strongest gradient along the normal to each control point (Cootes, Cooper et al. 1995) © 1995 Elsevier.*

In a B-snake, because the snake is controlled by fewer degrees of freedom, there is less need for the internal smoothness forces used with the original snakes, although these can still be derived and implemented using finite element analysis, i.e., taking derivatives and integrals of the B-spline basis functions (Terzopoulos 1983; Bathe 2007).

In practice, it is more common to estimate a set of *shape priors* on the typical distribution of the control points $\{\mathbf{x}_k\}$ (Cootes, Cooper et al. 1995). One potential way of describing this distribution would be by the location $\bar{\mathbf{x}}_k$ and 2D covariance $\mathbf{C}_k$ of each individual point $\mathbf{x}_k$. These could then be turned into a quadratic penalty (prior energy) on the point location. In practice, however, the variation in point locations is usually highly correlated.

A preferable approach is to estimate the joint covariance of all the points simultaneously. First, concatenate all of the point locations $\{\mathbf{x}_k\}$ into a single vector $\mathbf{x}$, e.g., by interleaving the x and y locations of each point. The distribution of these vectors across all training examples can be described with a mean $\bar{\mathbf{x}}$ and a covariance

$$\mathbf{C} = \frac{1}{P} \sum_p (\mathbf{x}_p - \bar{\mathbf{x}})(\mathbf{x}_p - \bar{\mathbf{x}})^T, \quad (7.33)$$

where $\mathbf{x}_p$ are the P training examples. Using *eigenvalue analysis* (Appendix A.1.2), which is also known as *principal component analysis* (PCA) (Section 5.2.3 and Appendix B.1), the covariance matrix can be written as,

$$\mathbf{C} = \Phi \text{diag}(\lambda_0 \dots \lambda_{K-1}) \Phi^T. \quad (7.34)$$

In most cases, the likely appearance of the points can be modeled using only a few eigenvectors with the largest eigenvalues. The resulting *point distribution model* (Cootes, Taylor *et al.* 1993; Cootes, Cooper *et al.* 1995) can be written as

$$\mathbf{x} = \bar{\mathbf{x}} + \hat{\Phi} \mathbf{b}, \quad (7.35)$$

where $\mathbf{b}$ is an $M \ll K$ element *shape parameter* vector and $\hat{\Phi}$ are the first m columns of Φ . To constrain the shape parameters to reasonable values, we can use a quadratic penalty of the form

$$E_{\text{shape}} = \frac{1}{2} \mathbf{b}^T \text{diag}(\lambda_0 \dots \lambda_{M-1}) \mathbf{b} = \sum_m b_m^2 / 2\lambda_m. \quad (7.36)$$

Alternatively, the range of allowable b_m values can be limited to some range, e.g., $|b_m| \leq 3\sqrt{\lambda_m}$ (Cootes, Cooper *et al.* 1995). Alternative approaches for deriving a set of shape vectors are reviewed by Isard and Blake (1998). Varying the individual shape parameters b_m over the range $-2\sqrt{\lambda_m} \leq 2\sqrt{\lambda_m}$ can give a good indication of the expected variation in appearance, as shown in Figure 7.41a.

To align a point distribution model with an image, each control point searches in a direction normal to the contour to find the most likely corresponding image edge point (Figure 7.41b). These individual measurements can be combined with priors on the shape parameters (and, if desired, position, scale, and orientation parameters) to estimate a new set of parameters. The resulting *active shape model* (ASM) can be iteratively minimized to fit images to non-rigidly deforming objects, such as medical images, or body parts, such as hands (Cootes, Cooper *et al.* 1995). The ASM can also be combined with a PCA analysis of the underlying gray-level distribution to create an *active appearance model* (AAM) (Cootes, Edwards, and Taylor 2001), which we discussed in more detail in Section 6.2.4.

Dynamic snakes and CONDENSATION

In many applications of active contours, the object of interest is being tracked from frame to frame as it deforms and evolves. In this case, it makes sense to use estimates from the previous frame to predict and constrain the new estimates.

One way to do this is to use Kalman filtering, which results in a formulation called *Kalman snakes* (Terzopoulos and Szeliski 1992; Blake, Curwen, and Zisserman 1993). The Kalman filter is based on a linear dynamic model of shape parameter evolution,

$$\mathbf{x}_t = \mathbf{A} \mathbf{x}_{t-1} + \mathbf{w}_t, \quad (7.37)$$

where $\mathbf{x}_t$ and $\mathbf{x}_{t-1}$ are the current and previous state variables, $\mathbf{A}$ is the linear *transition matrix*, and $\mathbf{w}$ is a noise (perturbation) vector, which is often modeled as a Gaussian (Gelb

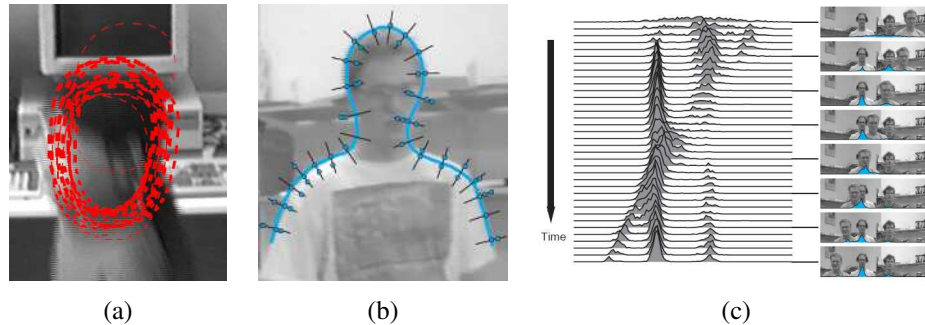


Figure 7.42 Head tracking using *CONDENSATION* (Isard and Blake 1998) © 1998 Springer: (a) sample set representation of head estimate distribution; (b) multiple measurements at each control vertex location; (c) multi-hypothesis tracking over time.

1974). The matrices $\mathbf{A}$ and the noise covariance can be learned ahead of time by observing typical sequences of the object being tracked (Blake and Isard 1998).

In many situations, however, such as when tracking in clutter, a better estimate for the contour can be obtained if we remove the assumptions that the distributions are Gaussian, which is what the Kalman filter requires. In this case, a general multi-modal distribution is propagated. To model such multi-modal distributions, Isard and Blake (1998) introduced the use of *particle filtering* to the computer vision community.¹² Particle filtering techniques represent a probability distribution using a collection of weighted point samples (Andrieu, de Freitas *et al.* 2003; Bishop 2006; Koller and Friedman 2009).

To update the locations of the samples according to the linear dynamics (deterministic drift), the centers of the samples are updated and multiple samples are generated for each point. These are then perturbed to account for the stochastic diffusion, i.e., their locations are moved by random vectors taken from the distribution of $\mathbf{w}$.¹³ Finally, the weights of these samples are multiplied by the measurement probability density, i.e., we take each sample and measure its likelihood given the current (new) measurements. Because the point samples represent and propagate conditional estimates of the multi-modal density, Isard and Blake (1998) dubbed their algorithm *CONDitional DENSITY propagation* or *CONDENSATION*.

Figure 7.42a shows what a factored sample of a head tracker might look like, drawing a red B-spline contour for each of (a subset of) the particles being tracked. Figure 7.42b shows why the measurement density itself is often multi-modal: the locations of the edges

¹²Alternatives to modeling multi-modal distributions include *mixtures of Gaussians* (Section 5.2.2) and *multiple hypothesis tracking* (Bar-Shalom and Fortmann 1988; Cham and Rehg 1999).

¹³Note that because of the structure of these steps, non-linear dynamics and non-Gaussian noise can be used.

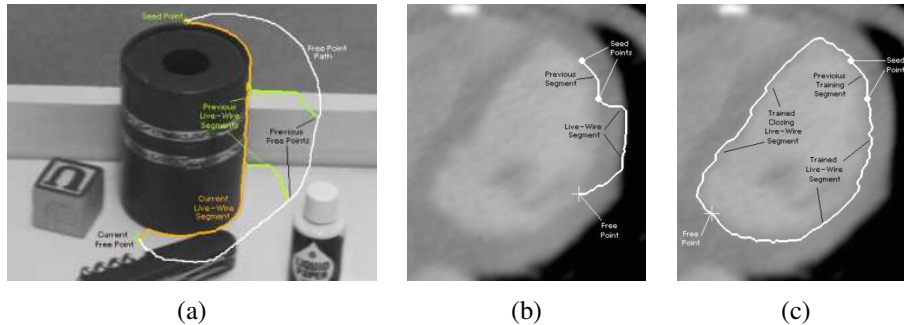


Figure 7.43 *Intelligent scissors: (a) as the mouse traces the white path, the scissors follow the orange path along the object boundary (the green curves show intermediate positions) (Mortensen and Barrett 1995) © 1995 ACM; (b) regular scissors can sometimes jump to a stronger (incorrect) boundary; (c) after training to the previous segment, similar edge profiles are preferred (Mortensen and Barrett 1998) © 1995 Elsevier.*

perpendicular to the spline curve can have multiple local maxima due to background clutter. Finally, Figure 7.42c shows the temporal evolution of the conditional density (x coordinate of the head and shoulder tracker centroid) as it tracks several people over time.

Scissors

Active contours allow a user to roughly specify a boundary of interest and have the system evolve the contour towards a more accurate location as well as track it over time. The results of this curve evolution, however, may be unpredictable and may require additional user-based hints to achieve the desired result.

An alternative approach is to have the system optimize the contour in real time as the user is drawing (Mortensen 1999). The *intelligent scissors* system developed by Mortensen and Barrett (1995) does just that. As the user draws a rough outline (the white curve in Figure 7.43a), the system computes and draws a better curve that clings to high-contrast edges (the orange curve).

To compute the optimal curve path (*live-wire*), the image is first pre-processed to associate low costs with edges (links between neighboring horizontal, vertical, and diagonal, i.e., $\mathcal{N}_8$ neighbors) that are likely to be boundary elements. Their system uses a combination of zero-crossing, gradient magnitudes, and gradient orientations to compute these costs.

Next, as the user traces a rough curve, the system continuously recomputes the lowest-cost path between the starting *seed point* and the current mouse location using Dijkstra's al-

gorithm, a breadth-first dynamic programming algorithm that terminates at the current target location.

In order to keep the system from jumping around unpredictably, the system will “freeze” the curve to date (reset the seed point) after a period of inactivity. To prevent the live wire from jumping onto adjacent higher-contrast contours, the system also “learns” the intensity profile under the current optimized curve, and uses this to preferentially keep the wire moving along the same (or a similar looking) boundary (Figure 7.43b–c).

Several extensions have been proposed to the basic algorithm, which works remarkably well even in its original form. Mortensen and Barrett (1999) use *tobogganing*, which is a simple form of watershed region segmentation, to pre-segment the image into regions whose boundaries become candidates for optimized curve paths. The resulting region boundaries are turned into a much smaller graph, where nodes are located wherever three or four regions meet. The Dijkstra algorithm is then run on this reduced graph, resulting in much faster (and often more stable) performance. Another extension to intelligent scissors is to use a probabilistic framework that takes into account the current trajectory of the boundary, resulting in a system called JetStream (Pérez, Blake, and Gangnet 2001).

Instead of re-computing an optimal curve at each time instant, a simpler system can be developed by simply “snapping” the current mouse position to the nearest likely boundary point (Gleicher 1995). Applications of these boundary extraction techniques to image cutting and pasting are presented in Section 10.4.

7.3.2 Level Sets

A limitation of active contours based on parametric curves of the form $\mathbf{f}(s)$, e.g., snakes, B-snakes, and CONDENSATION, is that it is challenging to change the topology of the curve as it evolves (McInerney and Terzopoulos 1999, 2000). Furthermore, if the shape changes dramatically, curve reparameterization may also be required.

An alternative representation for such closed contours is to use a *level set*, where the *zero-crossing(s)* of a *characteristic* (or signed distance (Section 3.3.3)) function define the curve. Level sets evolve to fit and track objects of interest by modifying the underlying *embedding function* (another name for this 2D function) $\phi(x, y)$ instead of the curve $\mathbf{f}(s)$ (Malladi, Sethian, and Vemuri 1995; Sethian 1999; Sapiro 2001; Osher and Paragios 2003). To reduce the amount of computation required, only a small strip (frontier) around the locations of the current zero-crossing needs to be updated at each step, which results in what are called *fast marching methods* (Sethian 1999).

An example of an evolution equation is the *geodesic active contour* proposed by Caselles,

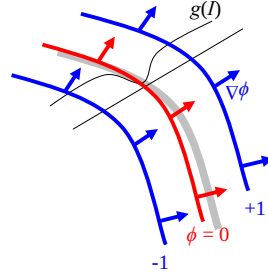


Figure 7.44 Level set evolution for a geodesic active contour. The embedding function ϕ is updated based on the curvature of the underlying surface modulated by the edge/speed function $g(I)$, as well as the gradient of $g(I)$, thereby attracting it to strong edges.

Kimmel, and Sapiro (1997) and Yezzi, Kichenassamy *et al.* (1997),

$$\begin{aligned} \frac{d\phi}{dt} &= |\nabla\phi| \operatorname{div} \left(g(I) \frac{\nabla\phi}{|\nabla\phi|} \right) \\ &= g(I) |\nabla\phi| \operatorname{div} \left(\frac{\nabla\phi}{|\nabla\phi|} \right) + \nabla g(I) \cdot \nabla\phi, \end{aligned} \quad (7.38)$$

where $g(I)$ is a generalized version of the snake edge potential. To get an intuitive sense of the curve's behavior, assume that the embedding function ϕ is a signed distance function away from the curve (Figure 7.44), in which case $|\phi| = 1$. The first term in Equation (7.38) moves the curve in the direction of its curvature, i.e., it acts to straighten the curve, under the influence of the modulation function $g(I)$. The second term moves the curve down the gradient of $g(I)$, encouraging the curve to migrate towards minima of $g(I)$.

While this level-set formulation can readily change topology, it is still susceptible to local minima, since it is based on local measurements such as image gradients. An alternative approach is to re-cast the problem in a segmentation framework, where the energy measures the consistency of the image statistics (e.g., color, texture, motion) inside and outside the segmented regions (Cremers, Rousson, and Deriche 2007; Rousson and Paragios 2008; Houhou, Thiran, and Bresson 2008). These approaches build on earlier energy-based segmentation frameworks introduced by Leclerc (1989), Mumford and Shah (1989), and Chan and Vese (2001), which are discussed in more detail in Section 4.3.2.

For more information on level sets and their applications, please see the collection of papers edited by Osher and Paragios (2003) as well as the series of Workshops on Variational and Level Set Methods in Computer Vision (Paragios, Faugeras *et al.* 2005) and Special Issues on Scale Space and Variational Methods in Computer Vision (Paragios and Sgallari

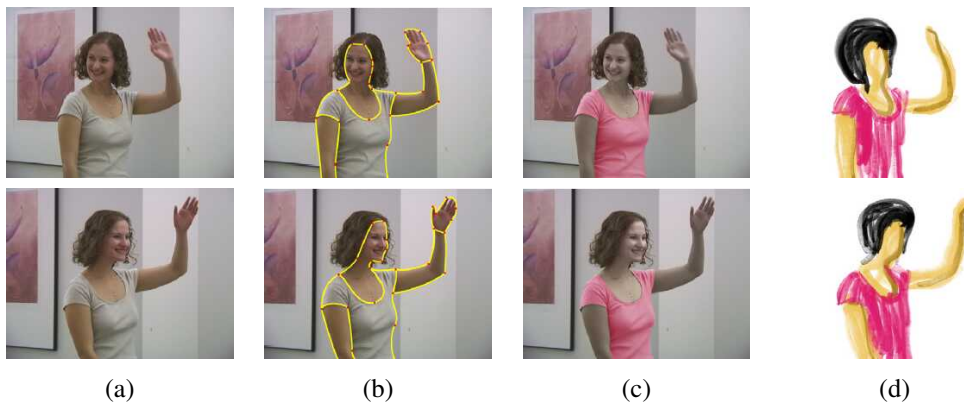


Figure 7.45 *Keyframe-based rotoscoping (Agarwala, Hertzmann et al. 2004) © 2004 ACM: (a) original frames; (b) rotoscoped contours; (c) re-colored blouse; (d) rotoscoped hand-drawn animation.*

2009).

7.3.3 Application: Contour tracking and rotoscoping

Active contours can be used in a wide variety of object-tracking applications (Blake and Isard 1998; Yilmaz, Javed, and Shah 2006). For example, they can be used to track facial features for performance-driven animation (Terzopoulos and Waters 1990; Lee, Terzopoulos, and Waters 1995; Parke and Waters 1996; Bregler, Covell, and Slaney 1997). They can also be used to track heads and people, as shown in Figure 7.42, as well as moving vehicles (Paragios and Deriche 2000). Additional applications include medical image segmentation, where contours can be tracked from slice to slice in computed tomography (Cootes and Taylor 2001), or over time, as in ultrasound scans.

An interesting application that is closer to computer animation and visual effects is *roto-scoping*, which uses the tracked contours to deform a set of hand-drawn animations (or to modify or replace the original video frames).¹⁴ Agarwala, Hertzmann et al. (2004) present a system based on tracking hand-drawn B-spline contours drawn at selected keyframes, using a combination of geometric and appearance-based criteria (Figure 7.45). They also provide an excellent review of previous rotoscoping and image-based, contour-tracking systems.

Additional applications of rotoscoping (object contour detection and segmentation), such

¹⁴The term comes from a device (a roto-scope) that projected frames of a live-action film underneath an acetate so that artists could draw animations directly over the actors' shapes.

as cutting and pasting objects from one photograph into another, are presented in Section 10.4.

7.4 Lines and vanishing points

While edges and general curves are suitable for describing the contours of natural objects, the human-made world is full of straight lines. Detecting and matching these lines can be useful in a variety of applications, including architectural modeling, pose estimation in urban environments, and the analysis of printed document layouts.

In this section, we present some techniques for extracting *piecewise linear* descriptions from the curves computed in the previous section. We begin with some algorithms for approximating a curve as a piecewise-linear polyline. We then describe the *Hough transform*, which can be used to group edgels into line segments even across gaps and occlusions. Finally, we describe how 3D lines with common *vanishing points* can be grouped together. These vanishing points can be used to calibrate a camera and to determine its orientation relative to a rectahedral scene, as described in Section 11.1.1.

7.4.1 Successive approximation

As we saw in Section 7.2.2, describing a curve as a series of 2D locations $\mathbf{x}_i = \mathbf{x}(s_i)$ provides a general representation suitable for matching and further processing. In many applications, however, it is preferable to approximate such a curve with a simpler representation, e.g., as a piecewise-linear polyline or as a B-spline curve (Farin 2002).

Many techniques have been developed over the years to perform this approximation, which is also known as *line simplification*. One of the oldest, and simplest, is the one proposed by Ramer (1972) and Douglas and Peucker (1973), who recursively subdivide the curve at the point furthest away from the line joining the two endpoints (or the current coarse polyline approximation). Hershberger and Snoeyink (1992) provide a more efficient implementation and also cite some of the other related work in this area.

Once the line simplification has been computed, it can be used to approximate the original curve. If a smoother representation or visualization is desired, either approximating or interpolating splines or curves can be used (Sections 3.5.1 and 7.3.1) (Szeliski and Ito 1986; Bartels, Beatty, and Barsky 1987; Farin 2002).

7.4.2 Hough transforms

While curve approximation with polylines can often lead to successful line extraction, lines in the real world are sometimes broken up into disconnected components or made up of many

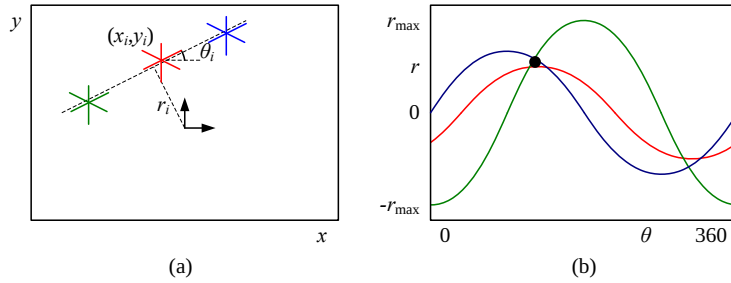


Figure 7.46 Original Hough transform: (a) each point votes for a complete family of potential lines $r_i(\theta) = x_i \cos \theta + y_i \sin \theta$; (b) each pencil of lines sweeps out a sinusoid in (r, θ) ; their intersection provides the desired line equation.

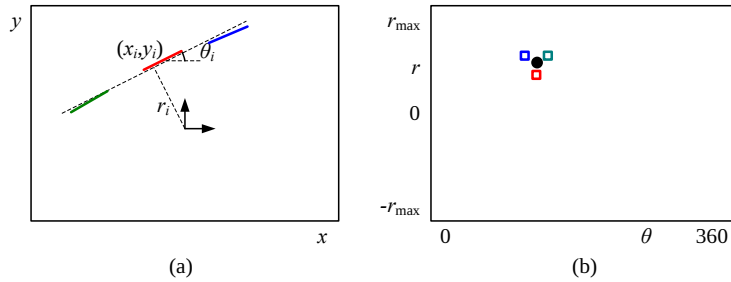


Figure 7.47 Oriented Hough transform: (a) an edgel re-parameterized in polar (r, θ) coordinates, with $\hat{\mathbf{n}}_i = (\cos \theta_i, \sin \theta_i)$ and $r_i = \hat{\mathbf{n}}_i \cdot \mathbf{x}_i$; (b) (r, θ) accumulator array, showing the votes for the three edgels marked in red, green, and blue.

collinear line segments. In many cases, it is desirable to group such collinear segments into extended lines. At a further processing stage (described in Section 7.4.3), we can then group such lines into collections with common vanishing points.

The Hough transform, named after its original inventor (Hough 1962), is a well-known technique for having edges “vote” for plausible line locations (Duda and Hart 1972; Ballard 1981; Illingworth and Kittler 1988). In its original formulation (Figure 7.46), each edge point votes for *all* possible lines passing through it, and lines corresponding to high *accumulator* or *bin* values are examined for potential line fits.¹⁵ Unless the points on a line are truly punctate, a better approach is to use the local orientation information at each edgel to vote for a *single* accumulator cell (Figure 7.47), as described below. A hybrid strategy, where each edgel votes

¹⁵The Hough transform can also be *generalized* to look for other geometric features, such as circles (Ballard 1981).

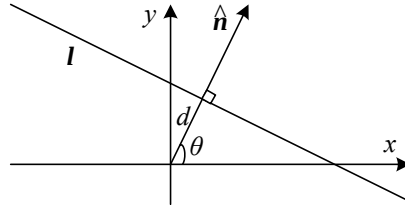


Figure 7.48 2D line equation expressed in terms of the normal $\hat{\mathbf{n}}$ and distance to the origin d .

for a number of possible orientation or location pairs centered around the estimate orientation, may be desirable in some cases.

Before we can vote for line hypotheses, we must first choose a suitable representation. Figure 7.48 (copied from Figure 2.2a) shows the normal-distance $(\hat{\mathbf{n}}, d)$ parameterization for a line. Since lines are made up of edge segments, we adopt the convention that the line normal $\hat{\mathbf{n}}$ points in the same direction (i.e., has the same sign) as the image gradient $\mathbf{J}(\mathbf{x}) = \nabla I(\mathbf{x})$ (7.19). To obtain a minimal two-parameter representation for lines, we convert the normal vector into an angle

$$\theta = \tan^{-1} n_y/n_x, \quad (7.39)$$

as shown in Figure 7.48. The range of possible (θ, d) values is $[-180^\circ, 180^\circ] \times [-\sqrt{2}, \sqrt{2}]$, assuming that we are using normalized pixel coordinates (2.61) that lie in $[-1, 1]$. The number of bins to use along each axis depends on the accuracy of the position and orientation estimate available at each edgel and the expected line density, and is best set experimentally with some test runs on sample imagery.

There are a lot of details in getting the Hough transform to work well, including using edge segment lengths or strengths during the voting process, keeping a list of constituent edgels in the accumulator array for easier post-processing, and optionally combining edges of different “polarity” into the same line segments. These are best worked out by writing an implementation and testing it out on sample data.

An alternative to the 2D polar (θ, d) representation for lines is to use the full 3D $\mathbf{m} = (\hat{\mathbf{n}}, d)$ line equation, projected onto the unit sphere. While the sphere can be parameterized using spherical coordinates (2.8),

$$\hat{\mathbf{m}} = (\cos \theta \cos \phi, \sin \theta \cos \phi, \sin \phi), \quad (7.40)$$

this does not uniformly sample the sphere and still requires the use of trigonometry.

An alternative representation can be obtained by using a *cube map*, i.e., projecting $\mathbf{m}$ onto the face of a unit cube (Figure 7.49a). To compute the cube map coordinate of a 3D vector

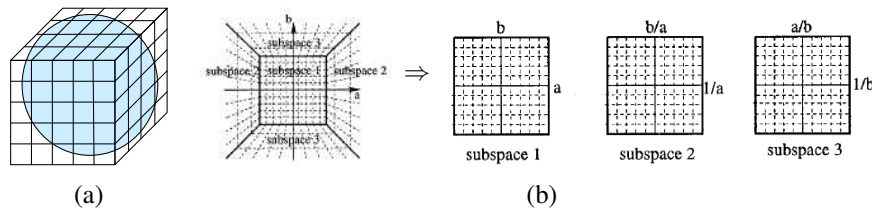


Figure 7.49 *Cube map representation for line equations and vanishing points: (a) a cube map surrounding the unit sphere; (b) projecting the half-cube onto three subspaces (Tuytelaars, Van Gool, and Proesmans 1997) © 1997 IEEE.*

$\mathbf{m}$, first find the largest (absolute value) component of $\mathbf{m}$, i.e., $m = \pm \max(|n_x|, |n_y|, |d|)$, and use this to select one of the six cube faces. Divide the remaining two coordinates by m and use these as indices into the cube face. While this avoids the use of trigonometry, it does require some decision logic.

One advantage of using the cube map, first pointed out by Tuytelaars, Van Gool, and Proesmans (1997), is that all of the lines passing through a point correspond to line segments on the cube faces, which is useful if the original (full voting) variant of the Hough transform is being used. In their work, they represent the line equation as $ax + b + y = 0$, which does not treat the x and y axes symmetrically. Note that if we restrict $d \geq 0$ by ignoring the polarity of the edge orientation (gradient sign), we can use a half-cube instead, which can be represented using only three cube faces, as shown in Figure 7.49b (Tuytelaars, Van Gool, and Proesmans 1997).

RANSAC-based line detection. Another alternative to the Hough transform is the Random Sample Consensus (RANSAC) algorithm described in more detail in Section 8.1.4. In brief, RANSAC randomly chooses pairs of edgels to form a line hypothesis and then tests how many other edgels fall onto this line. (If the edge orientations are accurate enough, a single edgel can produce this hypothesis.) Lines with sufficiently large numbers of *inliers* (matching edgels) are then selected as the desired line segments.

An advantage of RANSAC is that no accumulator array is needed, so the algorithm can be more space efficient and potentially less prone to the choice of bin size. The disadvantage is that many more hypotheses may need to be generated and tested than those obtained by finding peaks in the accumulator array.

Bottom-up grouping. Yet another approach to line segment detection is to iteratively group edgels with similar orientations into oriented rectangular *line-support regions* (Burns, Han-

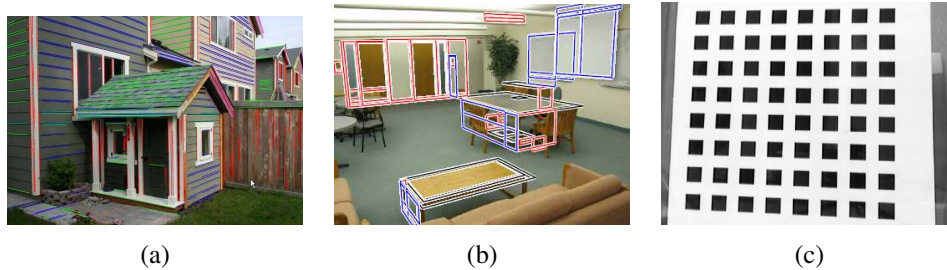


Figure 7.50 Real-world vanishing points: (a) architecture (Sinha, Steedly *et al.* 2008), (b) furniture (Mičušík, Wildenauer, and Košecká 2008) © 2008 IEEE, and (c) calibration patterns (Zhang 2000).

son, and Riseman 1986). The validity of such regions can then be determined using a statistical analysis, as described in the LSD paper by Grompone von Gioi, Jakubowicz *et al.* (2008). The resulting algorithm is quite fast, does a good job of distinguishing line segments from texture, and is widely used in practice because of its performance and open source availability. Recently, deep neural network algorithms have been developed to simultaneously extract line segments and their junctions (Huang, Wang *et al.* 2018; Zhang, Li *et al.* 2019; Huang, Qin *et al.* 2020; Lin, Pinteá, and van Gemert 2020).

In general, there is no clear consensus on which line estimation technique performs best. It is therefore a good idea to think carefully about the problem at hand and to implement several approaches (successive approximation, Hough, and RANSAC) to determine the one that works best for your application.

7.4.3 Vanishing points

In many scenes, structurally important lines have the same vanishing point because they are parallel in 3D. Examples of such lines are horizontal and vertical building edges, zebra crossings, railway tracks, the edges of furniture such as tables and dressers, and of course, the ubiquitous calibration pattern (Figure 7.50). Finding the vanishing points common to such line sets can help refine their position in the image and, in certain cases, help determine the intrinsic and extrinsic orientation of the camera (Section 11.1.1).

Over the years, a large number of techniques have been developed for finding vanishing points (Quan and Mohr 1989; Collins and Weiss 1990; Brillaut-O'Mahoney 1991; McLean and Kotturi 1995; Becker and Bove 1995; Shufelt 1999; Tuytelaars, Van Gool, and Proesmans 1997; Schaffalitzky and Zisserman 2000; Antone and Teller 2002; Rother 2002; Košecká and Zhang 2005; Denis, Elder, and Estrada 2008; Plugfelder 2008; Tardif 2009; Bazin, Seo *et al.*

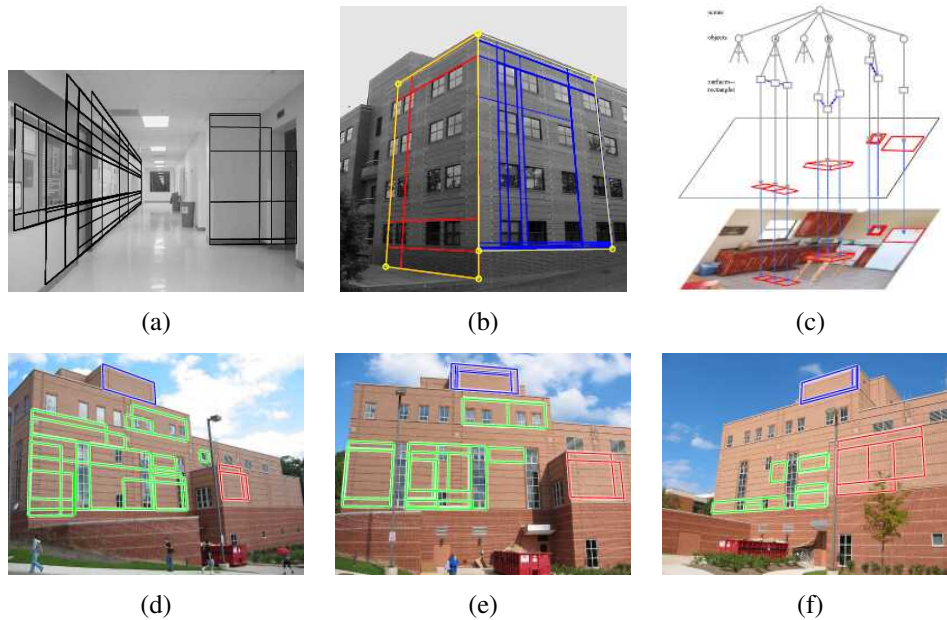


Figure 7.51 *Rectangle detection: (a) indoor corridor and (b) building exterior with grouped facades (Košecká and Zhang 2005) © 2005 Elsevier; (c) grammar-based recognition (Han and Zhu 2005) © 2005 IEEE; (d–f) rectangle matching using a plane sweep algorithm (Mičušík, Wildenauer, and Košecká 2008) © 2008 IEEE.*

2012; Antunes and Barreto 2013; Kluger, Ackermann *et al.* 2017; Zhou, Qi *et al.* 2019a)—see some of the more recent papers for additional references and alternative approaches.

In the first edition of this book (Szeliski 2010, Section 4.3.3), I presented a simple Hough technique based on having line pairs vote for potential vanishing point locations, followed by a robust least squares fitting stage. While my technique proceeds in two discrete stages, better results may be obtained by alternating between assigning lines to vanishing points and refitting the vanishing point locations (Antone and Teller 2002; Košecká and Zhang 2005; Plugfelder 2008). The results of detecting individual vanishing points can also be made more robust by simultaneously searching for pairs or triplets of mutually orthogonal vanishing points (Shufelt 1999; Antone and Teller 2002; Rother 2002; Sinha, Steedly *et al.* 2008; Li, Kim *et al.* 2020). Some results of such vanishing point detection algorithms can be seen in Figure 7.50. It is also possible to simultaneously detect line segments and their junctions using a neural network (Zhang, Li *et al.* 2019) and to then use these to construct complete 3D wireframe models (Zhou, Qi, and Ma 2019; Zhou, Qi *et al.* 2019b).

Rectangle detection

Once sets of mutually orthogonal vanishing points have been detected, it now becomes possible to search for 3D rectangular structures in the image (Figure 7.51). A variety of techniques have been developed to find such rectangles, primarily focused on architectural scenes (Kosecká and Zhang 2005; Han and Zhu 2005; Shaw and Barnes 2006; Micusík, Wildenauer, and Kosecká 2008; Schindler, Krishnamurthy *et al.* 2008).

After detecting orthogonal vanishing directions, Kosecká and Zhang (2005) refine the fitted line equations, search for corners near line intersections, and then verify rectangle hypotheses by rectifying the corresponding patches and looking for a preponderance of horizontal and vertical edges (Figures 7.51a–b). In follow-on work, Micusík, Wildenauer, and Kosecká (2008) use a Markov random field (MRF) to disambiguate between potentially overlapping rectangle hypotheses. They also use a plane sweep algorithm to match rectangles between different views (Figures 7.51d–f).

A different approach is proposed by Han and Zhu (2005), who use a grammar of potential rectangle shapes and nesting structures (between rectangles and vanishing points) to infer the most likely assignment of line segments to rectangles (Figure 7.51c). The idea of using regular, repetitive structures as part of the modeling process is now being called *holistic 3D reconstruction* (Zhou, Furukawa, and Ma 2019; Zhou, Furukawa *et al.* 2020; Pintore, Mura *et al.* 2020) and will be discussed in more detail in Section 13.6.1 on modeling 3D architecture.

7.5 Segmentation

Image segmentation is the task of finding groups of pixels that “go together”. In statistics and machine learning, this problem is known as *cluster analysis* or more simply *clustering* and is a widely studied area with hundreds of different algorithms (Jain and Dubes 1988; Kaufman and Rousseeuw 1990; Jain, Duin, and Mao 2000; Jain, Topchy *et al.* 2004; Xu and Wunsch 2005). We’ve already discussed general vector-space clustering algorithms in Section 5.2.1. The main difference between clustering and segmentation is that the former usually ignores pixel layout and neighborhoods, while the latter relies heavily on spatial cues and constraints.

In computer vision, image segmentation is one of the oldest and most widely studied problems (Brice and Fennema 1970; Pavlidis 1977; Riseman and Arbib 1977; Ohlander, Price, and Reddy 1978; Rosenfeld and Davis 1979; Haralick and Shapiro 1985). Early techniques often used region splitting or merging (Brice and Fennema 1970; Horowitz and Pavlidis 1976; Ohlander, Price, and Reddy 1978; Pavlidis and Liow 1990), which correspond to *divisive* and *agglomerative* algorithms (Jain, Topchy *et al.* 2004; Xu and Wunsch 2005), which we intro-

duced in Section 5.2.1. More recent algorithms typically optimize some global criterion, such as intra-region consistency and inter-region boundary lengths or dissimilarity (Leclerc 1989; Mumford and Shah 1989; Shi and Malik 2000; Comaniciu and Meer 2002; Felzenszwalb and Huttenlocher 2004; Cremers, Rousson, and Deriche 2007; Pont-Tuset, Arbeláez *et al.* 2017).

We have already seen examples of image segmentation using image morphology (Section 3.3.3), Markov random fields (Section 4.3), active contours (Section 7.3), and level sets (Section 7.3.2). In the recognition chapter (Section 6.4), we studied *semantic segmentation*, whose goal is to break the image up into semantically labeled regions such as sky, grass, and individual people and animals. In this section, we review some additional techniques for bottom-up general (non-semantic) image segmentation. These include algorithms based on region splitting and merging, graph-based segmentation, and probabilistic aggregation (Section 7.5.1), *mean shift* mode finding (Section 7.5.2), and *normalized cuts* splitting based on pixel similarity metrics (Section 7.5.3). Since many of these algorithms are no longer widely used, a lot of the descriptions have been considerably shortened from those found in the first edition of this book (Szeliski 2010, Chapter 5), where you can find longer descriptions.

Since the literature on image segmentation is so vast, a good way to get a handle on some of the better performing algorithms is to look at experimental comparisons on human-labeled databases (Arbeláez, Maire *et al.* 2011; Pont-Tuset, Arbeláez *et al.* 2017). The best known of these is the Berkeley Segmentation Dataset and Benchmark (Martin, Fowlkes *et al.* 2001), which consists of 1,000 images from a Corel image dataset that were hand-labeled by 30 human subjects, for which Unnikrishnan, Pantofaru, and Hebert (2007) propose new metrics for comparing segmentation algorithms, while Estrada and Jepson (2009) compare four well-known segmentation algorithms. A newer database of foreground and background segmentations, used by Alpert, Galun *et al.* (2007), is also available.

As mentioned in Section 3.3.3, the simplest possible technique for segmenting a grayscale image is to select a threshold and then compute connected components. Unfortunately, a single threshold is rarely sufficient for the whole image because of lighting and intra-object statistical variations.

Region splitting (divisive clustering). Splitting the image into successively finer regions is one of the oldest techniques in computer vision. Ohlander, Price, and Reddy (1978) present such a technique, which first computes a histogram for the whole image and then finds a threshold that best separates the large peaks in the histogram. This process is repeated until regions are either fairly uniform or below a certain size. More recent splitting algorithms often optimize some metric of intra-region similarity and inter-region dissimilarity. These are covered in Sections 4.3.2 and Sections 7.5.3.

Region merging (agglomerative clustering). Region merging techniques also date back to the beginnings of computer vision. Brice and Fennema (1970) use a dual grid for representing boundaries between pixels and merge regions based on their relative boundary lengths and the strength of the visible edges at these boundaries.

A very simple version of pixel-based merging combines adjacent regions whose average color difference is below a threshold or whose regions are too small. Segmenting the image into such *superpixels* (Mori, Ren *et al.* 2004), which are not semantically meaningful, can be a useful pre-processing stage to make higher-level algorithms such as stereo matching (Zitnick, Kang *et al.* 2004; Taguchi, Wilburn, and Zitnick 2008), optical flow (Zitnick, Jovic, and Kang 2005; Brox, Bregler, and Malik 2009), and recognition (Mori, Ren *et al.* 2004; Mori 2005; Gu, Lim *et al.* 2009; Lim, Arbeláez *et al.* 2009) both faster and more robust. It is also possible to combine both splitting and merging by starting with a medium-grain segmentation (in a quadtree representation) and then allowing both merging and splitting operations (Horowitz and Pavlidis 1976; Pavlidis and Liow 1990).

Watershed. A technique related to thresholding, since it operates on a grayscale image, is *watershed* computation (Vincent and Soille 1991). This technique segments an image into several *catchment basins*, which are the regions of an image (interpreted as a height field or landscape) where rain would flow into the same lake. An efficient way to compute such regions is to start flooding the landscape at all of the local minima and to label ridges wherever differently evolving components meet. The whole algorithm can be implemented using a priority queue of pixels and breadth-first search (Vincent and Soille 1991).¹⁶

Since images rarely have dark regions separated by lighter ridges, watershed segmentation is usually applied to a smoothed version of the gradient magnitude image, which also makes it usable with color images. As an alternative, the maximum oriented energy in a steerable filter (3.28–3.29) (Freeman and Adelson 1991) can be used as the basis of the *oriented watershed transform* developed by Arbeláez, Maire *et al.* (2011). Such techniques end up finding smooth regions separated by visible (higher gradient) boundaries. Since such boundaries are what active contours usually follow, active contour algorithms (Mortensen and Barrett 1999; Li, Sun *et al.* 2004) often precompute such a segmentation using either the watershed or the related *tobogganing* technique (Section 7.3.1).

¹⁶A related algorithm can be used to compute maximally stable extremal regions (MSERs) efficiently (Section 7.1.1) (Nistér and Stewénius 2008).

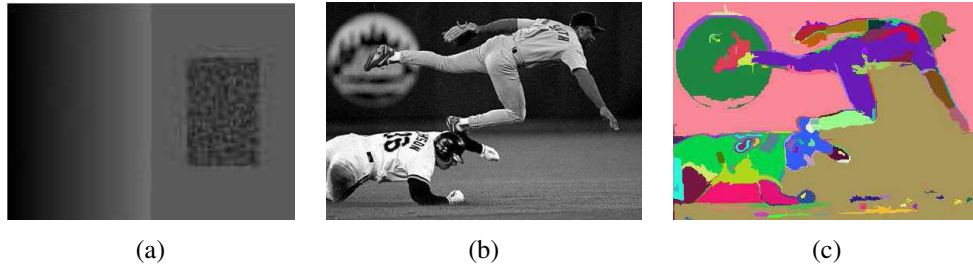


Figure 7.52 Graph-based merging segmentation (Felzenszwalb and Huttenlocher 2004)
 © 2004 Springer: (a) input grayscale image that is successfully segmented into three regions even though the variation inside the smaller rectangle is larger than the variation across the middle edge; (b) input grayscale image; (c) resulting segmentation using an $\mathcal{N}_8$ pixel neighborhood.

7.5.1 Graph-based segmentation

While many merging algorithms simply apply a fixed rule that groups pixels and regions together, Felzenszwalb and Huttenlocher (2004) present a merging algorithm that uses *relative dissimilarities* between regions to determine which ones should be merged; it produces an algorithm that provably optimizes a global grouping metric. They start with a pixel-to-pixel dissimilarity measure $w(e)$ that measures, for example, intensity differences between $\mathcal{N}_8$ neighbors. Alternatively, they can use the *joint feature space* distances introduced by Comaniciu and Meer (2002), which we discuss in Sections 7.5.2 and 7.5.3. Figure 7.52 shows two examples of images segmented using their technique.

Probabilistic aggregation

Alpert, Galun *et al.* (2007) develop a probabilistic merging algorithm based on two cues, namely gray-level similarity and texture similarity. The gray-level similarity between regions R_i and R_j is based on the *minimal external difference* from other neighboring regions, which is compared to the *average intensity difference* to compute the likelihoods p_{ij} that two regions should be merged. Merging proceeds in a hierarchical fashion inspired by algebraic multigrid techniques (Brandt 1986; Briggs, Henson, and McCormick 2000) and previously used by Alpert, Galun *et al.* (2007) in their segmentation by weighted aggregation (SWA) algorithm (Sharon, Galun *et al.* 2006). Figure 7.56 shows the segmentations produced by this algorithm compared to other popular segmentation algorithms.

7.5.2 Mean shift

Mean-shift and mode finding techniques, such as k-means and mixtures of Gaussians, model the feature vectors associated with each pixel (e.g., color and position) as samples from an unknown probability density function and then try to find clusters (modes) in this distribution.

Consider the color image shown in Figure 7.53a. How would you segment this image based on color alone? Figure 7.53b shows the distribution of pixels in $L^*u^*v^*$ space, which is equivalent to what a vision algorithm that ignores spatial location would see. To make the visualization simpler, let us only consider the L^*u^* coordinates, as shown in Figure 7.53c. How many obvious (elongated) clusters do you see? How would you go about finding these clusters?

The k-means and mixtures of Gaussians techniques we studied in Section 5.2.2 use a *parametric* model of the density function to answer this question, i.e., they assume the density is the superposition of a small number of simpler distributions (e.g., Gaussians) whose locations (centers) and shape (covariance) can be estimated. Mean shift, on the other hand, smooths the distribution and finds its peaks as well as the regions of feature space that correspond to each peak. Since a complete density is being modeled, this approach is called *non-parametric* (Bishop 2006).

The key to mean shift is a technique for efficiently finding peaks in this high-dimensional data distribution without ever computing the complete function explicitly (Fukunaga and Hostetler 1975; Cheng 1995; Comaniciu and Meer 2002). Consider once again the data points shown in Figure 7.53c, which can be thought of as having been drawn from some probability density function. If we could compute this density function, as visualized in Figure 7.53e, we could find its major peaks (*modes*) and identify regions of the input space that climb to the same peak as being part of the same region. This is the inverse of the *watershed* algorithm described in Section 7.5, which climbs downhill to find *basins of attraction*.

The first question, then, is how to estimate the density function given a sparse set of samples. One of the simplest approaches is to just smooth the data, e.g., by convolving it with a fixed kernel of width h , which, as we saw in Section 4.1.1, is the *Parzen window* approach to density estimation (Duda, Hart, and Stork 2001, Section 4.3; Bishop 2006, Section 2.5.1). Once we have computed $f(\mathbf{x})$, as shown in Figure 7.53e, we can find its local maxima using gradient ascent or some other optimization technique.

The problem with this “brute force” approach is that, for higher dimensions, it becomes computationally prohibitive to evaluate $f(\mathbf{x})$ over the complete search space. Instead, mean shift uses a variant of what is known in the optimization literature as *multiple restart gradient descent*. Starting at some guess for a local maximum, $\mathbf{y}_k$, which can be a random input data point $\mathbf{x}_i$, mean shift computes the gradient of the density estimate $f(\mathbf{x})$ at $\mathbf{y}_k$ and takes an

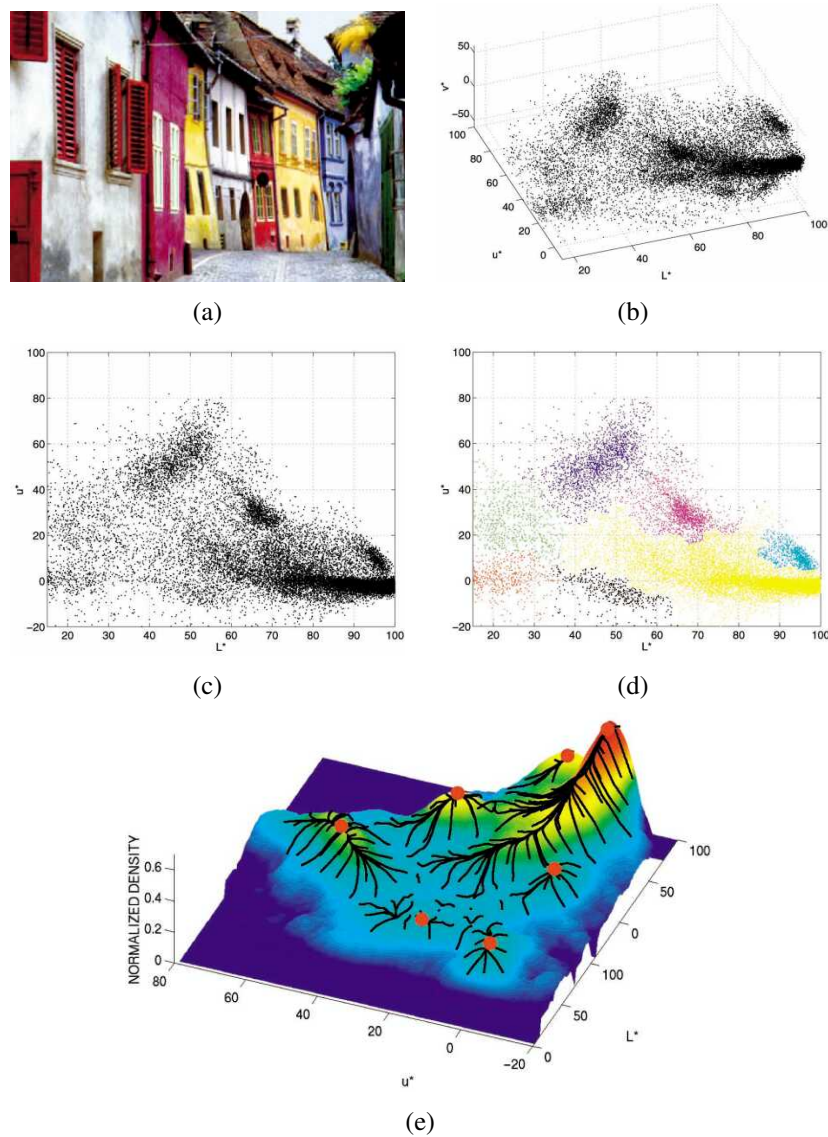


Figure 7.53 Mean-shift image segmentation (*Comaniciu and Meer 2002*) © 2002 IEEE: (a) input color image; (b) pixels plotted in $L^*u^*v^*$ space; (c) L^*u^* space distribution; (d) clustered results after 159 mean-shift procedures; (e) corresponding trajectories with peaks marked as red dots.

uphill step in that direction. Details on how this can be done efficiently can be found in papers on mean shift (Comaniciu and Meer 2002; Paris and Durand 2007) as well as the first edition of this book (Szeliski 2010, Section 5.3.2).

The color-based segmentation shown in Figure 7.53 only looks at pixel colors when determining the best clustering. It may therefore cluster together small isolated pixels that happen to have the same color, which may not correspond to a semantically meaningful segmentation of the image. Better results can usually be obtained by clustering in the *joint domain* of color and location. In this approach, the spatial coordinates of the image $\mathbf{x}_s = (x, y)$, which are called the *spatial domain*, are concatenated with the color values $\mathbf{x}_r$, which are known as the *range domain*, and mean-shift clustering is applied in this five-dimensional space $\mathbf{x}_j$. Since location and color may have different scales, the kernels are adjusted separately, just as in the bilateral filter kernel (3.34–3.37) discussed in Section 3.3.2. The difference between mean shift and bilateral filtering, however, is that in mean shift, the spatial coordinates of each pixel are adjusted along with its color values, so that the pixel migrates more quickly towards other pixels with similar colors, and can therefore later be used for clustering and segmentation.

Mean shift has been applied to a number of different problems in computer vision, including face tracking, 2D shape extraction, and texture segmentation (Comaniciu and Meer 2002), stereo matching (Wei and Quan 2004), non-photorealistic rendering (Section 10.5.2) (DeCarlo and Santella 2002), and video editing (Section 10.4.5) (Wang, Bhat *et al.* 2005). Paris and Durand (2007) provide a nice review of such applications, as well as techniques for more efficiently solving the mean-shift equations and producing hierarchical segmentations.

7.5.3 Normalized cuts

While bottom-up merging techniques aggregate regions into coherent wholes and mean-shift techniques try to find clusters of similar pixels using mode finding, the normalized cuts technique introduced by Shi and Malik (2000) examines the *affinities* (similarities) between nearby pixels and tries to separate groups that are connected by weak affinities.

Consider the simple graph shown in Figure 7.54a. The pixels in group *A* are all strongly connected with high affinities, shown as thick red lines, as are the pixels in group *B*. The connections between these two groups, shown as thinner blue lines, are much weaker. A *normalized cut* between the two groups, shown as a dashed line, separates them into two clusters.

The cut between two groups *A* and *B* is defined as the sum of all the weights being cut, where the weights between two pixels (or regions) *i* and *j* measure their similarity. Using a minimum cut as a segmentation criterion, however, does not result in reasonable clusters, since the smallest cuts usually involve isolating a single pixel.

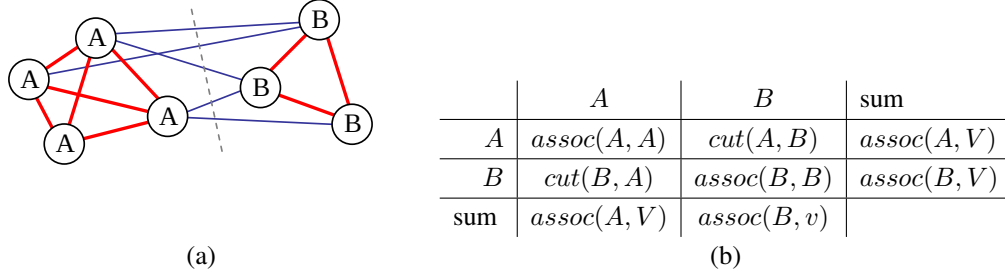


Figure 7.54 Sample weighted graph and its normalized cut: (a) a small sample graph and its smallest normalized cut; (b) tabular form of the associations and cuts for this graph. The *assoc* and *cut* entries are computed as area sums of the associated weight matrix $\mathbf{W}$. Normalizing the table entries by the row or column sums produces normalized associations and cuts $Nassoc$ and $Ncut$.

A better measure of segmentation is the normalized cut, which is defined as

$$Ncut(A, B) = \frac{cut(A, B)}{assoc(A, V)} + \frac{cut(A, B)}{assoc(B, V)}, \quad (7.41)$$

where $assoc(A, A) = \sum_{i \in A, j \in A} w_{ij}$ is the *association* (sum of all the weights) within a cluster and $assoc(A, V) = assoc(A, A) + cut(A, B)$ is the sum of *all* the weights associated with nodes in A . Figure 7.54b shows how the cuts and associations can be thought of as area sums in the weight matrix $\mathbf{W} = [w_{ij}]$, where the entries of the matrix have been arranged so that the nodes in A come first and the nodes in B come second. Dividing each of these areas by the corresponding row sum (the rightmost column of Figure 7.54b) results in the normalized cut and association values. These normalized values better reflect the fitness of a particular segmentation, since they look for collections of edges that are weak relative to all of the edges both inside and emanating from a particular region.

Unfortunately, computing the optimal normalized cut is NP-complete. Instead, Shi and Malik (2000) suggest computing a real-valued assignment of nodes to groups, using a generalized eigenvalue analysis of the *normalized* affinity matrix (Weiss 1999), as described in more detail in the normalized cuts paper and (Szeliski 2010, Section 5.4). Because these eigenvectors can be interpreted as the large modes of vibration in a spring-mass system, normalized cuts is an example of a *spectral method* for image segmentation. After the real-valued eigenvector is computed, the variables corresponding to positive and negative eigenvector values are associated with the two cut components. This process can be further repeated to hierarchically subdivide an image, as shown in Figure 7.55.

The original algorithm proposed by Shi and Malik (2000) used spatial position and image

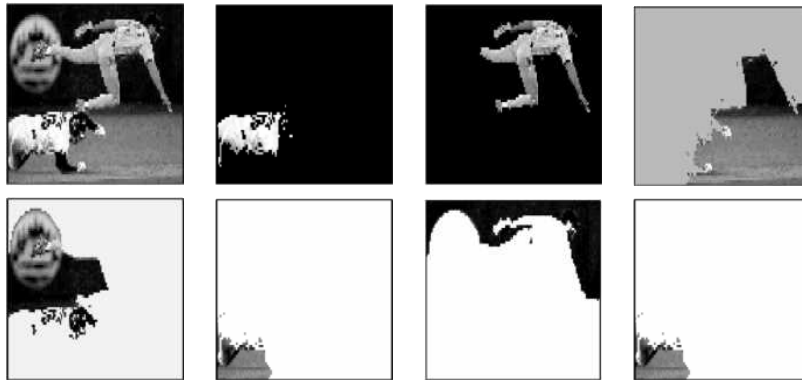


Figure 7.55 Normalized cuts segmentation (*Shi and Malik 2000*) © 2000 IEEE: The input image and the components returned by the normalized cuts algorithm.

feature differences to compute the pixel-wise affinities. In subsequent work, *Malik, Belongie et al. (2001)* look for *intervening contours* between pixels i and j to define intervening contour weights and then multiply these weights with a texon-based texture similarity metric. They then use an initial over-segmentation based purely on local pixel-wise features to re-estimate intervening contours and texture statistics in a region-based manner. Figure 7.56 shows the results of running this improved algorithm on a number of test images.

Because it requires the solution of large sparse eigenvalue problems, normalized cuts can be quite slow. *Sharon, Galun et al. (2006)* present a way to accelerate the computation of the normalized cuts using an approach inspired by algebraic multigrid (*Brandt 1986; Briggs, Henson, and McCormick 2000*).

An example of the segmentation produced by weighted aggregation (SWA) is shown in Figure 7.56, along with the most recent probabilistic bottom-up merging algorithm by *Alpert, Galun et al. (2007)*. In more recent work, *Pont-Tuset, Arbeláez et al. (2017)* speed up normalized cuts and extend it to multiple scales to obtain state-of-the-art results on both the Berkeley Segmentation Dataset as well as (at the time) object proposals on the VOC and COCO datasets.

7.6 Additional reading

One of the seminal papers on feature detection, description, and matching is by *Lowe (2004)*. Comprehensive surveys and evaluations of such techniques have been made by *Schmid, Mohr, and Bauckhage (2000)*, *Mikolajczyk and Schmid (2005)*, *Mikolajczyk, Tuytelaars et*



Figure 7.56 Comparative segmentation results (Alpert, Galun *et al.* 2007) © 2007 IEEE. “Our method” refers to the probabilistic bottom-up merging algorithm developed by Alpert *et al.*

al. (2005), and Tuytelaars and Mikolajczyk (2008), while Shi and Tomasi (1994) and Triggs (2004) also provide nice reviews.

In the area of feature detectors (Mikolajczyk, Tuytelaars *et al.* 2005), in addition to such classic approaches as Förstner–Harris (Förstner 1986; Harris and Stephens 1988) and difference of Gaussians (Lindeberg 1993, 1998b; Lowe 2004), maximally stable extremal regions (MSERs) are widely used for applications that require affine invariance (Matas, Chum *et al.* 2004; Nistér and Stewénus 2008). More recent interest point detectors are discussed by Xiao and Shah (2003), Koethe (2003), Carneiro and Jepson (2005), Kenney, Zuliani, and Manjunath (2005), Bay, Ess *et al.* (2008), Platel, Balmachnova *et al.* (2006), and Rosten, Porter, and Drummond (2010), as are techniques based on line matching (Zoghlami, Faugeras, and Deriche 1997; Bartoli, Coquerelle, and Sturm 2004) and region detection (Kadir, Zisserman, and Brady 2004; Matas, Chum *et al.* 2004; Tuytelaars and Van Gool 2004; Corso and Hager 2005). Three recent papers with nice reviews of DNN-based feature detectors are Balntas, Lenc *et al.* (2020), Barroso-Laguna, Riba *et al.* (2019), and Tian, Balntas *et al.* (2020).

A variety of local feature descriptors (and matching heuristics) are surveyed and compared by Mikolajczyk and Schmid (2005). More recent publications in this area include those by van de Weijer and Schmid (2006), Abdel-Hakim and Farag (2006), Winder and Brown (2007), and Hua, Brown, and Winder (2007) and the recent evaluations by Balntas, Lenc *et al.* (2020) and Jin, Mishkin *et al.* (2021). Techniques for efficiently matching features include k-d trees (Beis and Lowe 1999; Lowe 2004; Muja and Lowe 2009), pyramid matching kernels (Grauman and Darrell 2005), metric (vocabulary) trees (Nistér and Stewénus 2006),

variety of multi-dimensional hashing techniques (Shakhnarovich, Viola, and Darrell 2003; Torralba, Weiss, and Fergus 2008; Weiss, Torralba, and Fergus 2008; Kulis and Grauman 2009; Raginsky and Lazebnik 2009), and product quantization (Jégou, Douze, and Schmid 2010; Johnson, Douze, and Jégou 2021). A good review of large-scale systems for instance retrieval is Zheng, Yang, and Tian (2018).

The classic reference on feature detection and tracking is Shi and Tomasi (1994). More recent work in this field has focused on learning better matching functions for specific features (Avidan 2001; Jurie and Dhome 2002; Williams, Blake, and Cipolla 2003; Lepetit and Fua 2005; Lepetit, Pilet, and Fua 2006; Hinterstoisser, Benhimane *et al.* 2008; Rogez, Rihan *et al.* 2008; Özuysal, Calonder *et al.* 2010).

A highly cited and widely used edge detector is the one developed by Canny (1986). Alternative edge detectors as well as experimental comparisons can be found in publications by Nalwa and Binford (1986), Nalwa (1987), Deriche (1987), Freeman and Adelson (1991), Nalwa (1993), Heath, Sarkar *et al.* (1998), Crane (1997), Ritter and Wilson (2000), Bowyer, Kranenburg, and Dougherty (2001), Arbeláez, Maire *et al.* (2011), and Pont-Tuset, Arbeláez *et al.* (2017). The topic of scale selection in edge detection is nicely treated by Elder and Zucker (1998), while approaches to color and texture edge detection can be found in Ruzon and Tomasi (2001), Martin, Fowlkes, and Malik (2004), and Gevers, van de Weijer, and Stokman (2006). Edge detectors have also been combined with region segmentation techniques to further improve the detection of semantically salient boundaries (Maire, Arbelaez *et al.* 2008; Arbeláez, Maire *et al.* 2011; Xiaofeng and Bo 2012; Pont-Tuset, Arbeláez *et al.* 2017). Edges linked into contours can be smoothed and manipulated for artistic effect (Lowe 1989; Finkelstein and Salesin 1994; Taubin 1995) and used for recognition (Belongie, Malik, and Puzicha 2002; Tek and Kimia 2003; Sebastian and Kimia 2005).

The topic of active contours has a long history, beginning with the seminal work on snakes and other energy-minimizing variational methods (Kass, Witkin, and Terzopoulos 1988; Cootes, Cooper *et al.* 1995; Blake and Isard 1998), continuing through techniques such as intelligent scissors (Mortensen and Barrett 1995, 1999; Pérez, Blake, and Gangnet 2001), and culminating in level sets (Malladi, Sethian, and Vemuri 1995; Caselles, Kimmel, and Sapiro 1997; Sethian 1999; Paragios and Deriche 2000; Sapiro 2001; Osher and Paragios 2003; Paragios, Faugeras *et al.* 2005; Cremers, Rousson, and Deriche 2007; Rousson and Paragios 2008; Paragios and Sgallari 2009), which are currently the most widely used active contour methods.

An early, well-regarded paper on straight line extraction in images was written by Burns, Hanson, and Riseman (1986). Their idea of bottom-up *line-support regions* was extended by Grompone von Gioi, Jakubowicz *et al.* (2008) to construct the popular LSD line segment

detector. The literature on vanishing point detection is quite vast and still evolving (Quan and Mohr 1989; Collins and Weiss 1990; Brillaut-O'Mahoney 1991; McLean and Kotturi 1995; Becker and Bove 1995; Shufelt 1999; Tuytelaars, Van Gool, and Proesmans 1997; Schaffalitzky and Zisserman 2000; Antone and Teller 2002; Rother 2002; Kosecká and Zhang 2005; Denis, Elder, and Estrada 2008; Plugfelder 2008; Tardif 2009; Bazin, Seo *et al.* 2012; Antunes and Barreto 2013; Zhou, Qi *et al.* 2019a). Simultaneous line and junction detection techniques have also been developed (Huang, Wang *et al.* 2018; Zhang, Li *et al.* 2019).

The topic of image segmentation is closely related to clustering techniques, which are treated in a number of monographs and review articles (Jain and Dubes 1988; Kaufman and Rousseeuw 1990; Jain, Duin, and Mao 2000; Jain, Topchy *et al.* 2004). Some early segmentation techniques include those described by Brice and Fennema (1970), Pavlidis (1977), Riesenman and Arbib (1977), Ohlander, Price, and Reddy (1978), Rosenfeld and Davis (1979), and Haralick and Shapiro (1985), while examples of newer techniques are developed by Leclerc (1989), Mumford and Shah (1989), Shi and Malik (2000), and Felzenszwalb and Huttenlocher (2004).

Arbeláez, Maire *et al.* (2011) and Pont-Tuset, Arbeláez *et al.* (2017) provide good reviews of automatic segmentation techniques and compare their performance on the Berkeley Segmentation Dataset and Benchmark (Martin, Fowlkes *et al.* 2001).¹⁷ Additional comparison papers and databases include those by Unnikrishnan, Pantofaru, and Hebert (2007), Alpert, Galun *et al.* (2007), and Estrada and Jepson (2009).

Techniques for segmenting images based on local pixel similarities combined with aggregation or splitting methods include watersheds (Vincent and Soille 1991; Beare 2006; Arbeláez, Maire *et al.* 2011), region splitting (Ohlander, Price, and Reddy 1978), region merging (Brice and Fennema 1970; Pavlidis and Liow 1990; Jain, Topchy *et al.* 2004), as well as graph-based and probabilistic multi-scale approaches (Felzenszwalb and Huttenlocher 2004; Alpert, Galun *et al.* 2007).

Mean-shift algorithms, which find modes (peaks) in a density function representation of the pixels, are presented by Comaniciu and Meer (2002) and Paris and Durand (2007). Parametric mixtures of Gaussians can also be used to represent and segment such pixel densities (Bishop 2006; Ma, Derksen *et al.* 2007).

The seminal work on spectral (eigenvalue) methods for image segmentation is the *normalized cut* algorithm of Shi and Malik (2000). Related work includes that by Weiss (1999), Meila and Shi (2000), Meila and Shi (2001), Malik, Belongie *et al.* (2001), Ng, Jordan, and Weiss (2001), Yu and Shi (2003), Cour, Bénézit, and Shi (2005), Sharon, Galun *et al.* (2006), Tolliver and Miller (2006), and Wang and Oliensis (2010).

¹⁷<http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench>.

7.7 Exercises

Ex 7.1: Interest point detector. Implement one or more keypoint detectors and compare their performance (with your own or with a classmate's detector).

Possible detectors:

- Laplacian or Difference of Gaussian;
- Förstner–Harris Hessian (try different formula variants given in (7.9–7.11));
- oriented/steerable filter, looking for either second-order high second response or two edges in a window (Koethe 2003), as discussed in Section 7.1.1.
- any of the newer DNN-based detectors.

Other detectors are described in Mikolajczyk, Tuytelaars *et al.* (2005), Tuytelaars and Mikolajczyk (2008), and Balntas, Lenc *et al.* (2020). Additional optional steps could include:

1. Compute the detections on a sub-octave pyramid and find 3D maxima.
2. Find local orientation estimates using steerable filter responses or a gradient histogramming method.
3. Implement non-maximal suppression, such as the adaptive technique of Brown, Szeliski, and Winder (2005).
4. Vary the window shape and size (prefilter and aggregation).

To test for repeatability, download the code from <https://www.robots.ox.ac.uk/~vgg/research/affine> (Mikolajczyk, Tuytelaars *et al.* 2005; Tuytelaars and Mikolajczyk 2008) or simply rotate or shear your own test images. (Pick a domain you may want to use later, e.g., for outdoor stitching.)

Be sure to measure and report the stability of your scale and orientation estimates.

Ex 7.2: Interest point descriptor. Implement two or more descriptors from Section 7.1.2 (steered to local scale and orientation estimates, if appropriate) and compare their performance on some images of your own choosing.

You can either use the evaluation methodologies (and optionally software) described in Mikolajczyk and Schmid (2005), Balntas, Lenc *et al.* (2020), or Jin, Mishkin *et al.* (2021).

Ex 7.3: ROC curve computation. Given a pair of curves (histograms) plotting the number of matching and non-matching features as a function of Euclidean distance d as shown in

Figure 7.22b, derive an algorithm for plotting a ROC curve (Figure 7.22a). In particular, let $t(d)$ be the distribution of true matches and $f(d)$ be the distribution of (false) non-matches. Write down the equations for the ROC, i.e., TPR(FPR), and the AUC.

(Hint: Plot the cumulative distributions $T(d) = \int t(d)$ and $F(d) = \int f(d)$ and see if these help you derive the TPR and FPR at a given threshold θ .)

Ex 7.4: Feature matcher. After extracting features from a collection of overlapping or distorted images,¹⁸ match them up by their descriptors either using nearest neighbor matching or a more efficient matching strategy such as a k-d tree.

See whether you can improve the accuracy of your matches using techniques such as the nearest neighbor distance ratio.

Ex 7.5: Feature tracker. Instead of finding feature points independently in multiple images and then matching them, find features in the first image of a video or image sequence and then re-locate the corresponding points in the next frames using either search and gradient descent (Shi and Tomasi 1994) or learned feature detectors (Lepetit, Pilet, and Fua 2006; Fossati, Dimitrijevic *et al.* 2007). When the number of tracked points drops below a threshold or new regions in the image become visible, find additional points to track.

(Optional) Winnow out incorrect matches by estimating a homography (8.19–8.23) or fundamental matrix (Section 11.3.3).

(Optional) Refine the accuracy of your matches using the iterative registration algorithm described in Section 9.2 and Exercise 9.2.

Ex 7.6: Facial feature tracker. Apply your feature tracker to tracking points on a person's face, either manually initialized to interesting locations such as eye corners or automatically initialized at interest points.

(Optional) Match features between two people and use these features to perform image morphing (Exercise 3.25).

Ex 7.7: Edge detector. Implement an edge detector of your choice. Compare its performance to that of your classmates' detectors or code downloaded from the internet.

A simple but well-performing sub-pixel edge detector can be created as follows:

1. Blur the input image a little,

$$B_{\sigma}(\mathbf{x}) = G_{\sigma}(\mathbf{x}) * I(\mathbf{x}).$$

¹⁸<https://www.robots.ox.ac.uk/~vgg/research/affine>.

2. Construct a Gaussian pyramid (Exercise 3.17),

$$P = \text{Pyramid}\{B_\sigma(\mathbf{x})\}$$

3. Subtract an interpolated coarser-level pyramid image from the original resolution blurred image,

$$S(\mathbf{x}) = B_\sigma(\mathbf{x}) - P.\text{InterpolatedLevel}(L).$$

4. For each quad of pixels, $\{(i, j), (i + 1, j), (i, j + 1), (i + 1, j + 1)\}$, count the number of zero crossings along the four edges.
5. When there are exactly two zero crossings, compute their locations using (7.25) and store these edgel endpoints along with the midpoint in the edgel structure.
6. For each edgel, compute the local gradient by taking the horizontal and vertical differences between the values of S along the zero crossing edges.
7. Store the magnitude of this gradient as the edge strength and either its orientation or that of the segment joining the edgel endpoints as the edge orientation.
8. Add the edgel to a list of edgels or store it in a 2D array of edgels (addressed by pixel coordinates).

Ex 7.8: Edge linking and thresholding. Link up the edges computed in the previous exercise into chains and optionally perform thresholding with hysteresis.

The steps may include:

1. Store the edgels either in a 2D array (say, an integer image with indices into the edgel list) or pre-sort the edgel list first by (integer) x coordinates and then y coordinates, for faster neighbor finding.
2. Pick up an edgel from the list of unlinked edgels and find its neighbors in both directions until no neighbor is found or a closed contour is obtained. Flag edgels as linked as you visit them and push them onto your list of linked edgels.
3. (Optional) Perform hysteresis-based thresholding (Canny 1986). Use two thresholds “hi” and “lo” for the edge strength. A candidate edgel is considered an edge if either its strength is above the “hi” threshold or its strength is above the “lo” threshold and it is (recursively) connected to a previously detected edge.
4. (Optional) Link together contours that have small gaps but whose endpoints have similar orientations.

5. (Optional) Find junctions between adjacent contours, e.g., using some of the ideas (or references) from Maire, Arbelaez *et al.* (2008).

Ex 7.9: Contour matching. Convert a closed contour (linked edgel list) into its arc-length parameterization and use this to match object outlines.

The steps may include:

1. Walk along the contour and create a list of (x_i, y_i, s_i) triplets, using the arc-length formula

$$s_{i+1} = s_i + \|\mathbf{x}_{i+1} - \mathbf{x}_i\|. \quad (7.42)$$

2. Resample this list onto a regular set of (x_j, y_j, j) samples using linear interpolation of each segment.
3. Compute the average values of x and y , i.e., $\bar{x}$ and $\bar{y}$ and subtract them from your sampled curve points.
4. Resample the original (x_i, y_i, s_i) piecewise-linear function onto a length-independent set of samples, say $j \in [0, 1023]$. (Using a length which is a power of two makes subsequent Fourier transforms more convenient.)
5. Compute the Fourier transform of the curve, treating each (x, y) pair as a complex number.
6. To compare two curves, fit a linear equation to the phase difference between the two curves. (Careful: phase wraps around at 360° . Also, you may wish to weight samples by their Fourier spectrum magnitude—see Section 9.1.2.)
7. (Optional) Prove that the constant phase component corresponds to the temporal shift in s , while the linear component corresponds to rotation.

Of course, feel free to try any other curve descriptor and matching technique from the computer vision literature (Tek and Kimia 2003; Sebastian and Kimia 2005).

Ex 7.10: Jigsaw puzzle solver—challenging. Write a program to automatically solve a jigsaw puzzle from a set of scanned puzzle pieces. Your software may include the following components:

1. Scan the pieces (either face up or face down) on a flatbed scanner with a distinctively colored background.
2. (Optional) Scan in the box top to use as a low-resolution reference image.

3. Use color-based thresholding to isolate the pieces.
4. Extract the contour of each piece using edge finding and linking.
5. (Optional) Re-represent each contour using an arc-length or some other re-parameterization. Break up the contours into meaningful matchable pieces. (Is this hard?)
6. (Optional) Associate color values with each contour to help in the matching.
7. (Optional) Match pieces to the reference image using some rotationally invariant feature descriptors.
8. Solve a global optimization or (backtracking) search problem to snap pieces together and place them in the correct location relative to the reference image.
9. Test your algorithm on a succession of more difficult puzzles and compare your results with those of others.

For some additional ideas, have a look at [Cho, Avidan, and Freeman \(2010\)](#).

Ex 7.11: Successive approximation line detector. Implement a line simplification algorithm (Section 7.4.1) ([Ramer 1972](#); [Douglas and Peucker 1973](#)) to convert a hand-drawn curve (or linked edge image) into a small set of polylines.

(Optional) Re-render this curve using either an approximating or interpolating spline or Bezier curve ([Szeliski and Ito 1986](#); [Bartels, Beatty, and Barsky 1987](#); [Farin 2002](#)).

Ex 7.12: Line fitting uncertainty. Estimate the uncertainty (covariance) in your line fit using uncertainty analysis.

1. After determining which edgels belong to the line segment (using either successive approximation or Hough transform), re-fit the line segment using total least squares ([Van Huffel and Vandewalle 1991](#); [Van Huffel and Lemmerling 2002](#)), i.e., find the mean or centroid of the edgels and then use eigenvalue analysis to find the dominant orientation.
2. Compute the perpendicular errors (deviations) to the line and robustly estimate the variance of the fitting noise using an estimator such as MAD ([Appendix B.3](#)).
3. (Optional) re-fit the line parameters by throwing away outliers or using a robust norm or influence function.
4. Estimate the error in the perpendicular location of the line segment and its orientation.

Ex 7.13: Vanishing points. Compute the vanishing points in an image using one of the techniques described in Section 7.4.3 and optionally refine the original line equations associated with each vanishing point. Your results can be used later to track a target or reconstruct architecture (Section 13.6.1).

Ex 7.14: Vanishing point uncertainty. Perform an uncertainty analysis on your estimated vanishing points. You will need to decide how to represent your vanishing point, e.g., homogeneous coordinates on a sphere, to handle vanishing points near infinity.

See the discussion of Bingham distributions by Collins and Weiss (1990) for some ideas.

Ex 7.15: Region segmentation. Implement one of the region segmentation algorithms described in this chapter. Some popular segmentation algorithms include:

- k-means (Section 5.2.2);
- mixtures of Gaussians (Section 5.2.2);
- mean shift (Section 7.5.2);
- normalized cuts (Section 7.5.3);
- similarity graph-based segmentation (Section 7.5.1);
- binary Markov random fields solved using graph cuts (Section 4.3.2).

Apply your region segmentation to a video sequence and use it to track moving regions from frame to frame.

Alternatively, test out your segmentation algorithm on the Berkeley segmentation database (Martin, Fowlkes *et al.* 2001).

Chapter 8

Image alignment and stitching

8.1	Pairwise alignment	503
8.1.1	2D alignment using least squares	504
8.1.2	<i>Application: Panography</i>	506
8.1.3	Iterative algorithms	507
8.1.4	Robust least squares and RANSAC	510
8.1.5	3D alignment	513
8.2	Image stitching	514
8.2.1	Parametric motion models	516
8.2.2	<i>Application: Whiteboard and document scanning</i>	517
8.2.3	Rotational panoramas	519
8.2.4	Gap closing	520
8.2.5	<i>Application: Video summarization and compression</i>	522
8.2.6	Cylindrical and spherical coordinates	523
8.3	Global alignment	526
8.3.1	Bundle adjustment	527
8.3.2	Parallax removal	531
8.3.3	Recognizing panoramas	533
8.4	Compositing	536
8.4.1	Choosing a compositing surface	536
8.4.2	Pixel selection and weighting (deghosting)	538
8.4.3	<i>Application: Photomontage</i>	544
8.4.4	Blending	544
8.5	Additional reading	547
8.6	Exercises	549



Figure 8.1 Image stitching: (a) geometric alignment of 2D images for stitching (Szeliski and Shum 1997) © 1997 ACM; (b) a spherical panorama constructed from 54 photographs (Szeliski and Shum 1997) © 1997 ACM; (c) a multi-image panorama automatically assembled from an unordered photo collection; a multi-image stitch (d) without and (e) with moving object removal (Uyttendaele, Eden, and Szeliski 2001) © 2001 IEEE.

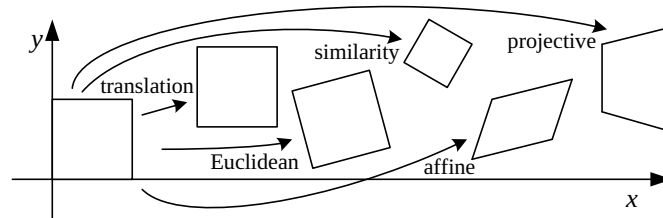


Figure 8.2 Basic set of 2D planar transformations

Once we have extracted features from images, the next stage in many vision algorithms is to match these features across different images (Section 7.1.3). An important component of this matching is to verify whether the set of matching features is geometrically consistent, e.g., whether the feature displacements can be described by a simple 2D or 3D geometric transformation. The computed motions can then be used in other applications such as image stitching (Section 8.2) or augmented reality (Section 11.2.2).

In this chapter, we look at the topic of geometric image registration, i.e., the computation of 2D and 3D transformations that map features in one image to another (Section 8.1). In Chapter 11, we look at the related problems of *pose estimation*, which is determining a camera's position relative to a known 3D object or scene, and *structure from motion*, i.e., how to simultaneously estimate 3D geometry and camera motion.

8.1 Pairwise alignment

Feature-based alignment is the problem of estimating the motion between two or more sets of matched 2D or 3D points. In this section, we restrict ourselves to global *parametric* transformations, such as those described in Section 2.1.1 and shown in Table 2.1 and Figure 8.2, or higher order transformation for curved surfaces (Shashua and Toelg 1997; Can, Stewart *et al.* 2002). Applications to non-rigid or elastic deformations (Bookstein 1989; Kambhamettu, Goldgof *et al.* 1994; Szeliski and Lavallée 1996; Torresani, Hertzmann, and Bregler 2008) are examined in Sections 9.2.2 and 13.6.4.

Transform	Matrix	Parameters $\mathbf{p}$	Jacobian $\mathbf{J}$
translation	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$	(t_x, t_y)	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
Euclidean	$\begin{bmatrix} c_\theta & -s_\theta & t_x \\ s_\theta & c_\theta & t_y \end{bmatrix}$	(t_x, t_y, θ)	$\begin{bmatrix} 1 & 0 & -s_\theta x - c_\theta y \\ 0 & 1 & c_\theta x - s_\theta y \end{bmatrix}$
similarity	$\begin{bmatrix} 1+a & -b & t_x \\ b & 1+a & t_y \end{bmatrix}$	(t_x, t_y, a, b)	$\begin{bmatrix} 1 & 0 & x & -y \\ 0 & 1 & y & x \end{bmatrix}$
affine	$\begin{bmatrix} 1+a_{00} & a_{01} & t_x \\ a_{10} & 1+a_{11} & t_y \end{bmatrix}$	$(t_x, t_y, a_{00}, a_{01}, a_{10}, a_{11})$	$\begin{bmatrix} 1 & 0 & x & y & 0 & 0 \\ 0 & 1 & 0 & 0 & x & y \end{bmatrix}$
projective	$\begin{bmatrix} 1+h_{00} & h_{01} & h_{02} \\ h_{10} & 1+h_{11} & h_{12} \\ h_{20} & h_{21} & 1 \end{bmatrix}$	$(h_{00}, h_{01}, \dots, h_{21})$	(see Section 8.1.3)

Table 8.1 Jacobians of the 2D coordinate transformations $\mathbf{x}' = \mathbf{f}(\mathbf{x}; \mathbf{p})$ shown in Table 2.1, where we have re-parameterized the motions so that they are identity for $\mathbf{p} = 0$.

8.1.1 2D alignment using least squares

Given a set of matched feature points $\{(\mathbf{x}_i, \mathbf{x}'_i)\}$ and a planar parametric transformation¹ of the form

$$\mathbf{x}' = \mathbf{f}(\mathbf{x}; \mathbf{p}), \quad (8.1)$$

how can we produce the best estimate of the motion parameters $\mathbf{p}$? The usual way to do this is to use least squares, i.e., to minimize the sum of squared residuals

$$E_{\text{LS}} = \sum_i \|\mathbf{r}_i\|^2 = \sum_i \|\mathbf{f}(\mathbf{x}_i; \mathbf{p}) - \mathbf{x}'_i\|^2, \quad (8.2)$$

where

$$\mathbf{r}_i = \mathbf{x}'_i - \mathbf{f}(\mathbf{x}_i; \mathbf{p}) = \hat{\mathbf{x}}'_i - \tilde{\mathbf{x}}'_i \quad (8.3)$$

is the *residual* between the measured location $\hat{\mathbf{x}}'_i$ and its corresponding current *predicted* location $\tilde{\mathbf{x}}'_i = \mathbf{f}(\mathbf{x}_i; \mathbf{p})$. (See Appendix A.2 for more on least squares and Appendix B.2 for a statistical justification.)

¹For examples of non-planar parametric models, such as quadrics, see the work of Shashua and Toelg (1997) and Shashua and Wexler (2001).

Many of the motion models presented in Section 2.1.1 and Table 2.1, i.e., translation, similarity, and affine, have a *linear* relationship between the amount of motion $\Delta \mathbf{x} = \mathbf{x}' - \mathbf{x}$ and the unknown parameters $\mathbf{p}$,

$$\Delta \mathbf{x} = \mathbf{x}' - \mathbf{x} = \mathbf{J}(\mathbf{x})\mathbf{p}, \quad (8.4)$$

where $\mathbf{J} = \partial \mathbf{f} / \partial \mathbf{p}$ is the *Jacobian* of the transformation $\mathbf{f}$ with respect to the motion parameters $\mathbf{p}$ (see Table 8.1). In this case, a simple *linear* regression (linear least squares problem) can be formulated as

$$E_{\text{LLS}} = \sum_i \|\mathbf{J}(\mathbf{x}_i)\mathbf{p} - \Delta \mathbf{x}_i\|^2 \quad (8.5)$$

$$= \mathbf{p}^T \left[\sum_i \mathbf{J}^T(\mathbf{x}_i)\mathbf{J}(\mathbf{x}_i) \right] \mathbf{p} - 2\mathbf{p}^T \left[\sum_i \mathbf{J}^T(\mathbf{x}_i)\Delta \mathbf{x}_i \right] + \sum_i \|\Delta \mathbf{x}_i\|^2 \quad (8.6)$$

$$= \mathbf{p}^T \mathbf{A} \mathbf{p} - 2\mathbf{p}^T \mathbf{b} + c. \quad (8.7)$$

The minimum can be found by solving the symmetric positive definite (SPD) system of *normal equations*²

$$\mathbf{A} \mathbf{p} = \mathbf{b}, \quad (8.8)$$

where

$$\mathbf{A} = \sum_i \mathbf{J}^T(\mathbf{x}_i)\mathbf{J}(\mathbf{x}_i) \quad (8.9)$$

is called the *Hessian* and $\mathbf{b} = \sum_i \mathbf{J}^T(\mathbf{x}_i)\Delta \mathbf{x}_i$. For the case of pure translation, the resulting equations have a particularly simple form, i.e., the translation is the average translation between corresponding points or, equivalently, the translation of the point centroids.

Uncertainty weighting. The above least squares formulation assumes that all feature points are matched with the same accuracy. This is often not the case, since certain points may fall into more textured regions than others. If we associate a scalar variance estimate σ_i^2 with each correspondence, we can minimize the *weighted least squares* problem instead,³

$$E_{\text{WLS}} = \sum_i \sigma_i^{-2} \|\mathbf{r}_i\|^2. \quad (8.10)$$

²For poorly conditioned problems, it is better to use QR decomposition on the set of linear equations $\mathbf{J}(\mathbf{x}_i)\mathbf{p} = \Delta \mathbf{x}_i$ instead of the normal equations (Björck 1996; Golub and Van Loan 1996). However, such conditions rarely arise in image registration.

³Problems where each measurement can have a different variance or uncertainty are called *heteroscedastic models*.



Figure 8.3 A simple panograph consisting of three images automatically aligned with a translational model and then averaged together.

As shown in Section 9.1.3, a covariance estimate for patch-based matching can be obtained by multiplying the inverse of the *patch Hessian* $\mathbf{A}_i$ (9.48) with the per-pixel noise covariance σ_n^2 (9.37). Weighting each squared residual by its inverse covariance $\Sigma_i^{-1} = \sigma_n^{-2} \mathbf{A}_i$ (which is called the *information matrix*), we obtain

$$E_{\text{CWLS}} = \sum_i \|\mathbf{r}_i\|_{\Sigma_i^{-1}}^2 = \sum_i \mathbf{r}_i^T \Sigma_i^{-1} \mathbf{r}_i = \sum_i \sigma_n^{-2} \mathbf{r}_i^T \mathbf{A}_i \mathbf{r}_i. \quad (8.11)$$

8.1.2 Application: Panography

One of the simplest (and most fun) applications of image alignment is a special form of image stitching called *panography*. In a panograph, images are translated and optionally rotated and scaled before being blended with simple averaging (Figure 8.3). This process mimics the photographic collages created by artist David Hockney, although his compositions use an opaque overlay model, being created out of regular photographs.

In most of the examples seen on the web, the images are aligned by hand for best artistic effect.⁴ However, it is also possible to use feature matching and alignment techniques to perform the registration automatically (Nomura, Zhang, and Nayar 2007; Zelnik-Manor and Perona 2007).

Consider a simple translational model. We want all the corresponding features in different images to line up as best as possible. Let $\mathbf{t}_j$ be the location of the j th image coordinate frame in the global composite frame and $\mathbf{x}_{ij}$ be the location of the i th matched feature in the j th

⁴<https://www.flickr.com/groups/panography/>.

image. In order to align the images, we wish to minimize the least squares error

$$E_{\text{PLS}} = \sum_{ij} \|(\mathbf{t}_j + \mathbf{x}_{ij}) - \mathbf{x}_i\|^2, \quad (8.12)$$

where $\mathbf{x}_i$ is the consensus (average) position of feature i in the global coordinate frame. (An alternative approach is to register each pair of overlapping images separately and then compute a consensus location for each frame—see Exercise 8.2.)

The above least squares problem is indeterminate (you can add a constant offset to all the frame and point locations $\mathbf{t}_j$ and $\mathbf{x}_i$). To fix this, either pick one frame as being at the origin or add a constraint to make the average frame offsets be 0.

The formulas for adding rotation and scale transformations are straightforward and are left as an exercise (Exercise 8.2). See if you can create some collages that you would be happy to share with others on the web.

8.1.3 Iterative algorithms

While linear least squares is the simplest method for estimating parameters, most problems in computer vision do not have a simple linear relationship between the measurements and the unknowns. In this case, the resulting problem is called *non-linear least squares* or *non-linear regression*.

Consider, for example, the problem of estimating a rigid Euclidean 2D transformation (translation plus rotation) between two sets of points. If we parameterize this transformation by the translation amount (t_x, t_y) and the rotation angle θ , as in Table 2.1, the Jacobian of this transformation, given in Table 8.1, depends on the current value of θ . Notice how in Table 8.1, we have re-parameterized the motion matrices so that they are always the identity at the origin $\mathbf{p} = 0$, which makes it easier to initialize the motion parameters.

To minimize the non-linear least squares problem, we iteratively find an update $\Delta\mathbf{p}$ to the current parameter estimate $\mathbf{p}$ by minimizing

$$E_{\text{NLS}}(\Delta\mathbf{p}) = \sum_i \|\mathbf{f}(\mathbf{x}_i; \mathbf{p} + \Delta\mathbf{p}) - \mathbf{x}'_i\|^2 \quad (8.13)$$

$$\approx \sum_i \|\mathbf{J}(\mathbf{x}_i; \mathbf{p})\Delta\mathbf{p} - \mathbf{r}_i\|^2 \quad (8.14)$$

$$= \Delta\mathbf{p}^T \left[\sum_i \mathbf{J}^T \mathbf{J} \right] \Delta\mathbf{p} - 2\Delta\mathbf{p}^T \left[\sum_i \mathbf{J}^T \mathbf{r}_i \right] + \sum_i \|\mathbf{r}_i\|^2 \quad (8.15)$$

$$= \Delta\mathbf{p}^T \mathbf{A} \Delta\mathbf{p} - 2\Delta\mathbf{p}^T \mathbf{b} + c, \quad (8.16)$$

where the “Hessian”⁵ $\mathbf{A}$ is the same as Equation (8.9) and the right-hand side vector

$$\mathbf{b} = \sum_i \mathbf{J}^T(\mathbf{x}_i) \mathbf{r}_i \quad (8.17)$$

is now a Jacobian-weighted sum of residual vectors. This makes intuitive sense, as the parameters are pulled in the direction of the prediction error with a strength proportional to the Jacobian.

Once $\mathbf{A}$ and $\mathbf{b}$ have been computed, we solve for $\Delta \mathbf{p}$ using

$$(\mathbf{A} + \lambda \text{diag}(\mathbf{A})) \Delta \mathbf{p} = \mathbf{b}, \quad (8.18)$$

and update the parameter vector $\mathbf{p} \leftarrow \mathbf{p} + \Delta \mathbf{p}$ accordingly. The parameter λ is an additional damping parameter used to ensure that the system takes a “downhill” step in energy (squared error) and is an essential component of the Levenberg–Marquardt algorithm (described in more detail in Appendix A.3). In many applications, it can be set to 0 if the system is successfully converging.

For the case of our 2D translation+rotation, we end up with a 3×3 set of normal equations in the unknowns $(\delta t_x, \delta t_y, \delta \theta)$. An initial guess for (t_x, t_y, θ) can be obtained by fitting a four-parameter similarity transform in (t_x, t_y, c, s) and then setting $\theta = \tan^{-1}(s/c)$. An alternative approach is to estimate the translation parameters using the centroids of the 2D points and to then estimate the rotation angle using polar coordinates (Exercise 8.3).

For the other 2D motion models, the derivatives in Table 8.1 are all fairly straightforward, except for the projective 2D motion (homography), which arises in image-stitching applications (Section 8.2). These equations can be re-written from (2.21) in their new parametric form as

$$x' = \frac{(1 + h_{00})x + h_{01}y + h_{02}}{h_{20}x + h_{21}y + 1} \quad \text{and} \quad y' = \frac{h_{10}x + (1 + h_{11})y + h_{12}}{h_{20}x + h_{21}y + 1}. \quad (8.19)$$

The Jacobian is therefore

$$\mathbf{J} = \frac{\partial \mathbf{f}}{\partial \mathbf{p}} = \frac{1}{D} \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -x'x & -x'y \\ 0 & 0 & 0 & x & y & 1 & -y'x & -y'y \end{bmatrix}, \quad (8.20)$$

where $D = h_{20}x + h_{21}y + 1$ is the denominator in (8.19), which depends on the current parameter settings (as do x' and y').

An initial guess for the eight unknowns $\{h_{00}, h_{01}, \dots, h_{21}\}$ can be obtained by multiplying both sides of the equations in (8.19) through by the denominator, which yields the linear

⁵The “Hessian” $\mathbf{A}$ is not the true Hessian (second derivative) of the non-linear least squares problem (8.13). Instead, it is the approximate Hessian, which neglects second (and higher) order derivatives of $\mathbf{f}(\mathbf{x}_i; \mathbf{p} + \Delta \mathbf{p})$.

set of equations,

$$\begin{bmatrix} \hat{x}' - x \\ \hat{y}' - y \end{bmatrix} = \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -\hat{x}'x & -\hat{x}'y \\ 0 & 0 & 0 & x & y & 1 & -\hat{y}'x & -\hat{y}'y \end{bmatrix} \begin{bmatrix} h_{00} \\ \vdots \\ h_{21} \end{bmatrix}. \quad (8.21)$$

However, this is not optimal from a statistical point of view, since the denominator D , which was used to multiply each equation, can vary quite a bit from point to point.⁶

One way to compensate for this is to *reweight* each equation by the inverse of the current estimate of the denominator, D ,

$$\frac{1}{D} \begin{bmatrix} \hat{x}' - x \\ \hat{y}' - y \end{bmatrix} = \frac{1}{D} \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -\hat{x}'x & -\hat{x}'y \\ 0 & 0 & 0 & x & y & 1 & -\hat{y}'x & -\hat{y}'y \end{bmatrix} \begin{bmatrix} h_{00} \\ \vdots \\ h_{21} \end{bmatrix}. \quad (8.22)$$

While this may at first seem to be the exact same set of equations as (8.21), because least squares is being used to solve the over-determined set of equations, the weightings *do* matter and produce a different set of normal equations that performs better in practice.

The most principled way to do the estimation, however, is to directly minimize the squared residual Equations (8.13) using the Gauss–Newton approximation, i.e., performing a first-order Taylor series expansion in $\mathbf{p}$, as shown in (8.14), which yields the set of equations

$$\begin{bmatrix} \hat{x}' - \tilde{x}' \\ \hat{y}' - \tilde{y}' \end{bmatrix} = \frac{1}{D} \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -\tilde{x}'x & -\tilde{x}'y \\ 0 & 0 & 0 & x & y & 1 & -\tilde{y}'x & -\tilde{y}'y \end{bmatrix} \begin{bmatrix} \Delta h_{00} \\ \vdots \\ \Delta h_{21} \end{bmatrix}. \quad (8.23)$$

While these look similar to (8.22), they differ in two important respects. First, the left-hand side consists of unweighted *prediction errors* rather than point displacements and the solution vector is a *perturbation* to the parameter vector $\mathbf{p}$. Second, the quantities inside $\mathbf{J}$ involve *predicted* feature locations $(\tilde{x}', \tilde{y}')$ instead of *sensed* feature locations $(\hat{x}', \hat{y}')$. Both of these differences are subtle and yet they lead to an algorithm that, when combined with proper checking for downhill steps (as in the Levenberg–Marquardt algorithm), will converge to a local minimum. Note that iterating Equations (8.22) is not guaranteed to converge, since it is not minimizing a well-defined energy function.

⁶Hartley and Zisserman (2004) call this strategy of forming linear equations from rational equations the *direct linear transform*, but that term is more commonly associated with pose estimation (Section 11.2). Note also that our definition of the h_{ij} parameters differs from that used in their book, since we define h_{ii} to be the *difference* from unity and we do not leave h_{22} as a free parameter, which means that we cannot handle certain extreme homographies.

Equation (8.23) is analogous to the *additive* algorithm for direct intensity-based registration (Section 9.2), since the change to the full transformation is being computed. If we prepend an incremental homography to the current homography instead, i.e., we use a *compositional* algorithm (described in Section 9.2), we get $D = 1$ (since $\mathbf{p} = 0$) and the above formula simplifies to

$$\begin{bmatrix} \hat{x}' - x \\ \hat{y}' - y \end{bmatrix} = \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -x^2 & -xy \\ 0 & 0 & 0 & x & y & 1 & -xy & -y^2 \end{bmatrix} \begin{bmatrix} \Delta h_{00} \\ \vdots \\ \Delta h_{21} \end{bmatrix}, \quad (8.24)$$

where we have replaced $(\tilde{x}', \tilde{y}')$ with (x, y) for conciseness.

8.1.4 Robust least squares and RANSAC

While regular least squares is the method of choice for measurements where the noise follows a normal (Gaussian) distribution, more robust versions of least squares are required when there are outliers among the correspondences (as there almost always are). In this case, it is preferable to use an *M-estimator* (Huber 1981; Hampel, Ronchetti *et al.* 1986; Black and Rangarajan 1996; Stewart 1999), which involves applying a robust penalty function $\rho(r)$ to the residuals

$$E_{\text{RLS}}(\Delta \mathbf{p}) = \sum_i \rho(\|\mathbf{r}_i\|) \quad (8.25)$$

instead of squaring them.⁷

We can take the derivative of this function with respect to $\mathbf{p}$ and set it to 0,

$$\sum_i \psi(\|\mathbf{r}_i\|) \frac{\partial \|\mathbf{r}_i\|}{\partial \mathbf{p}} = \sum_i \frac{\psi(\|\mathbf{r}_i\|)}{\|\mathbf{r}_i\|} \mathbf{r}_i^T \frac{\partial \mathbf{r}_i}{\partial \mathbf{p}} = 0, \quad (8.26)$$

where $\psi(r) = \rho'(r)$ is the derivative of ρ and is called the *influence function*. If we introduce a *weight function*, $w(r) = \psi(r)/r$, we observe that finding the stationary point of (8.25) using (8.26) is equivalent to minimizing the *iteratively reweighted least squares* (IRLS) problem

$$E_{\text{IRLS}} = \sum_i w(\|\mathbf{r}_i\|) \|\mathbf{r}_i\|^2, \quad (8.27)$$

where the $w(\|\mathbf{r}_i\|)$ play the same local weighting role as σ_i^{-2} in (8.10). The IRLS algorithm alternates between computing the influence functions $w(\|\mathbf{r}_i\|)$ and solving the resulting weighted least squares problem (with fixed w values). Other incremental robust least squares

⁷The plots for some commonly used robust penalty functions ρ can be found in Figure 4.7.

algorithms can be found in the work of Sawhney and Ayer (1996), Black and Anandan (1996), Black and Rangarajan (1996), and Baker, Gross *et al.* (2003) and in textbooks and tutorials on robust statistics (Huber 1981; Hampel, Ronchetti *et al.* 1986; Rousseeuw and Leroy 1987; Stewart 1999).

While M-estimators can definitely help reduce the influence of outliers, in some cases, starting with too many outliers will prevent IRLS (or other gradient descent algorithms) from converging to the global optimum. A better approach is often to find a starting set of *inlier* correspondences, i.e., points that are consistent with a dominant motion estimate.⁸

Two widely used approaches to this problem are called RANdom Sample Consensus, or RANSAC for short (Fischler and Bolles 1981), and *least median of squares* (LMS) (Rousseeuw 1984). Both techniques start by selecting (at random) a subset of k correspondences, which is then used to compute an initial estimate for $\mathbf{p}$. The *residuals* of the full set of correspondences are then computed as

$$\mathbf{r}_i = \tilde{\mathbf{x}}'_i(\mathbf{x}_i; \mathbf{p}) - \hat{\mathbf{x}}'_i, \quad (8.28)$$

where $\tilde{\mathbf{x}}'_i$ are the *estimated* (mapped) locations and $\hat{\mathbf{x}}'_i$ are the sensed (detected) feature point locations.⁹

The RANSAC technique then counts the number of *inliers* that are within ϵ of their predicted location, i.e., whose $\|\mathbf{r}_i\| \leq \epsilon$. (The ϵ value is application dependent but is often around 1–3 pixels.) Least median of squares finds the median value of the $\|\mathbf{r}_i\|^2$ values. The random selection process is repeated S times and the sample set with the largest number of inliers (or with the smallest median residual) is kept as the final solution. Either the initial parameter guess $\mathbf{p}$ or the full set of computed inliers is then passed on to the next data fitting stage.

When the number of measurements is quite large, it may be preferable to only score a subset of the measurements in an initial round that selects the most plausible hypotheses for additional scoring and selection. This modification of RANSAC, which can significantly speed up its performance, is called *Preemptive RANSAC* (Nistér 2003). In another variant on RANSAC called PROSAC (PROgressive Sample Consensus), random samples are initially added from the most “confident” matches, thereby speeding up the process of finding a (statistically) likely good set of inliers (Chum and Matas 2005). Raguram, Chum *et al.* (2012) provide a unified framework from which most of these techniques can be derived as well as a nice experimental comparison.

Additional variants on RANSAC include MLESAC (Torr and Zisserman 2000), DSAC

⁸For pixel-based alignment methods (Section 9.1.1), hierarchical (coarse-to-fine) techniques are often used to lock onto the *dominant motion* in a scene.

⁹For problems such as epipolar geometry estimation, the residual may be the distance between a point and a line.

k	p	S
3	0.5	35
6	0.6	97
6	0.5	293

Table 8.2 Number of trials S to attain a 99% probability of success (Stewart 1999).

(Brachmann, Krull *et al.* 2017), Graph-Cut RANSAC (Barath and Matas 2018), MAGSAC (Barath, Matas, and Nuskova 2019), and ESAC (Brachmann and Rother 2019). Some of these algorithms, such as DSAC (Differentiable RANSAC), are designed to be differentiable so they can be used in end-to-end training of feature detection and matching pipelines (Section 7.1). The MAGSAC++ paper by Barath, Nuskova *et al.* (2020) compares many of these variants. Yang, Antonante *et al.* (2020) claim that using a robust penalty function with a decreasing outlier parameter, i.e., *graduated non-convexity* (Blake and Zisserman 1987; Barron 2019), can outperform RANSAC in many geometric correspondence and pose estimation problems. To ensure that the random sampling has a good chance of finding a true set of inliers, a sufficient number of trials S must be evaluated. Let p be the probability that any given correspondence is valid and P be the probability of success after S trials. The likelihood in one trial that all k random samples are inliers is p^k . Therefore, the likelihood that S such trials will all fail is

$$1 - P = (1 - p^k)^S \quad (8.29)$$

and the required minimum number of trials is

$$S = \frac{\log(1 - P)}{\log(1 - p^k)}. \quad (8.30)$$

Stewart (1999) gives examples of the required number of trials S to attain a 99% probability of success. As you can see from Table 8.2, the number of trials grows quickly with the number of sample points used. This provides a strong incentive to use the *minimum* number of sample points k possible for any given trial, which is how RANSAC is normally used in practice.

Uncertainty modeling

In addition to robustly computing a good alignment, some applications require the computation of uncertainty (see Appendix B.6). For linear problems, this estimate can be obtained by inverting the Hessian matrix (8.9) and multiplying it by the feature position noise, if these

have not already been used to weight the individual measurements, as in Equations (8.10) and (8.11). In statistics, the Hessian, which is the inverse covariance, is sometimes called the (Fisher) *information matrix* (Appendix B.1).

When the problem involves non-linear least squares, the inverse of the Hessian matrix provides the *Cramer–Rao lower bound* on the covariance matrix, i.e., it provides the *minimum* amount of covariance in a given solution, which can actually have a wider spread (“longer tails”) if the energy flattens out away from the local minimum where the optimal solution is found.

8.1.5 3D alignment

Instead of aligning 2D sets of image features, many computer vision applications require the alignment of 3D points. In the case where the 3D transformations are linear in the motion parameters, e.g., for translation, similarity, and affine, regular least squares (8.5) can be used.

The case of rigid (Euclidean) motion,

$$E_{\text{R3D}} = \sum_i \|\mathbf{x}'_i - \mathbf{R}\mathbf{x}_i - \mathbf{t}\|^2, \quad (8.31)$$

which arises more frequently and is often called the *absolute orientation* problem (Horn 1987), requires slightly different techniques. If only scalar weightings are being used (as opposed to full 3D per-point anisotropic covariance estimates), the weighted centroids of the two point clouds $\mathbf{c}$ and $\mathbf{c}'$ can be used to estimate the translation $\mathbf{t} = \mathbf{c}' - \mathbf{R}\mathbf{c}$.¹⁰ We are then left with the problem of estimating the rotation between two sets of points $\{\hat{\mathbf{x}}_i = \mathbf{x}_i - \mathbf{c}\}$ and $\{\hat{\mathbf{x}}'_i = \mathbf{x}'_i - \mathbf{c}'\}$ that are both centered at the origin.

One commonly used technique is called the *orthogonal Procrustes algorithm* (Golub and Van Loan 1996, p. 601) and involves computing the singular value decomposition (SVD) of the 3×3 correlation matrix

$$\mathbf{C} = \sum_i \hat{\mathbf{x}}'_i \hat{\mathbf{x}}_i^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T. \quad (8.32)$$

The rotation matrix is then obtained as $\mathbf{R} = \mathbf{U}\mathbf{V}^T$. (Verify this for yourself when $\hat{\mathbf{x}}' = \mathbf{R}\hat{\mathbf{x}}$.)

Another technique is the absolute orientation algorithm (Horn 1987) for estimating the unit quaternion corresponding to the rotation matrix $\mathbf{R}$, which involves forming a 4×4 matrix from the entries in $\mathbf{C}$ and then finding the eigenvector associated with its largest positive eigenvalue.

¹⁰When full covariances are used, they are transformed by the rotation, so a closed-form solution for translation is not possible.

Lorusso, Eggert, and Fisher (1995) experimentally compare these two techniques to two additional techniques proposed in the literature, but find that the difference in accuracy is negligible (well below the effects of measurement noise).

In situations where these closed-form algorithms are not applicable, e.g., when full 3D covariances are being used or when the 3D alignment is part of some larger optimization, the incremental rotation update introduced in Section 2.1.3 (2.35–2.36), which is parameterized by an instantaneous rotation vector ω , can be used (See Section 8.2.3 for an application to image stitching.)

In some situations, e.g., when merging range data maps, the correspondence between data points is not known *a priori*. In this case, iterative algorithms that start by matching nearby points and then update the most likely correspondence can be used (Besl and McKay 1992; Zhang 1994; Szeliski and Lavallée 1996; Gold, Rangarajan *et al.* 1998; David, DeMenthon *et al.* 2004; Li and Hartley 2007; Enqvist, Josephson, and Kahl 2009). These techniques are discussed in more detail in Section 13.2.1.

8.2 Image stitching

Algorithms for aligning images and stitching them into seamless photo-mosaics are among the oldest and most widely used in computer vision (Milgram 1975; Peleg 1981). Image stitching algorithms create the high-resolution photo-mosaics used to produce today's digital maps and satellite photos. They are also now a standard mode in smartphone cameras and can be used to create beautiful ultra wide-angle panoramas.

Image stitching originated in the photogrammetry community, where more manually intensive methods based on surveyed *ground control points* or manually registered *tie points* have long been used to register aerial photos into large-scale photo-mosaics (Slama 1980). One of the key advances in this community was the development of *bundle adjustment* algorithms (Section 11.4.2), which could simultaneously solve for the locations of all of the camera positions, thus yielding globally consistent solutions (Triggs, McLauchlan *et al.* 1999). Another recurring problem in creating photo-mosaics is the elimination of visible seams, for which a variety of techniques have been developed over the years (Milgram 1975, 1977; Peleg 1981; Davis 1998; Agarwala, Dontcheva *et al.* 2004)

In film photography, special cameras were developed in the 1990s to take ultra-wide-angle panoramas, often by exposing the film through a vertical slit as the camera rotated on its axis (Meehan 1990). In the mid-1990s, image alignment techniques started being applied to the construction of wide-angle seamless panoramas from regular hand-held cameras (Mann and Picard 1994; Chen 1995; Szeliski 1996). Subsequent algorithms addressed the need to

compute globally consistent alignments (Szeliski and Shum 1997; Sawhney and Kumar 1999; Shum and Szeliski 2000), to remove “ghosts” due to parallax and object movement (Davis 1998; Shum and Szeliski 2000; Uyttendaele, Eden, and Szeliski 2001; Agarwala, Dontcheva *et al.* 2004), and to deal with varying exposures (Mann and Picard 1994; Uyttendaele, Eden, and Szeliski 2001; Levin, Zomet *et al.* 2004; Eden, Uyttendaele, and Szeliski 2006; Kopf, Uyttendaele *et al.* 2007).¹¹

While early techniques worked by directly minimizing pixel-to-pixel dissimilarities, today’s algorithms extract a sparse set of features and match them to each other, as described in Chapter 7. Such feature-based approaches (Zoghiani, Faugeras, and Deriche 1997; Capel and Zisserman 1998; Cham and Cipolla 1998; Badra, Qumsieh, and Dudek 1998; McLauchlan and Jaenicke 2002; Brown and Lowe 2007) have the advantage of being more robust against scene movement and are usually faster,¹² Their biggest advantage, however, is the ability to “recognize panoramas”, i.e., to automatically discover the adjacency (overlap) relationships among an unordered set of images, which makes them ideally suited for fully automated stitching of panoramas taken by casual users (Brown and Lowe 2007).

What, then, are the essential problems in image stitching? As with image alignment, we must first determine the appropriate mathematical model relating pixel coordinates in one image to pixel coordinates in another; Section 8.2.1 reviews the basic models we have studied and presents some new motion models related specifically to panoramic image stitching. Next, we must somehow estimate the correct alignments relating various pairs (or collections) of images. Chapter 7 discusses how distinctive features can be found in each image and then efficiently matched to rapidly establish correspondences between pairs of images. Chapter 9 discusses how direct pixel-to-pixel comparisons combined with gradient descent (and other optimization techniques) can also be used to estimate these parameters. When multiple images exist in a panorama, global optimization techniques can be used to compute a globally consistent set of alignments and to efficiently discover which images overlap one another. In Section 8.3, we look at how each of these previously developed techniques can be modified to take advantage of the imaging setups commonly used to create panoramas.

Once we have aligned the images, we must choose a final compositing surface for warping the aligned images (Section 8.4.1). We also need algorithms to seamlessly cut and blend overlapping images, even in the presence of parallax, lens distortion, scene motion, and exposure differences (Section 8.4.2–8.4.4).

¹¹A collection of some of these papers was compiled by Benosman and Kang (2001) and they are surveyed by Szeliski (2006a).

¹²See a discussion of the pros and cons of direct vs. feature-based techniques in (Triggs, Zisserman, and Szeliski 2000) and in the first edition of this book (Szeliski 2010, Section 8.3.4).

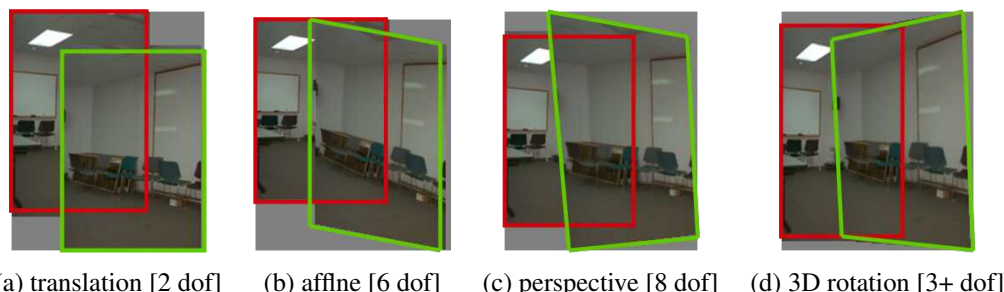


Figure 8.4 *Two-dimensional motion models and how they can be used for image stitching.*

8.2.1 Parametric motion models

Before we can register and align images, we need to establish the mathematical relationships that map pixel coordinates from one image to another. A variety of such *parametric motion models* are possible, from simple 2D transforms, to planar perspective models, 3D camera rotations, lens distortions, and mapping to non-planar (e.g., cylindrical) surfaces.

We already covered several of these models in Sections 2.1 and 8.1. In particular, we saw in Section 2.1.4 how the parametric motion describing the deformation of a planar surface as viewed from different positions can be described with an eight-parameter homography (2.71) (Mann and Picard 1994; Szeliski 1996). We also saw how a camera undergoing a pure rotation induces a different kind of homography (2.72).

In this section, we review both of these models and show how they can be applied to different stitching situations. We also introduce spherical and cylindrical compositing surfaces and show how, under favorable circumstances, they can be used to perform alignment using pure translations (Section 8.2.6). Deciding which alignment model is most appropriate for a given situation or set of data is a *model selection* problem (Torr 2002; Bishop 2006; Robert 2007; Hastie, Tibshirani, and Friedman 2009; Murphy 2012), an important topic we do not cover in this book.

Planar perspective motion

The simplest possible motion model to use when aligning images is to simply translate and rotate them in 2D (Figure 8.4a). This is exactly the same kind of motion that you would use if you had overlapping photographic prints. It is also the kind of technique favored by David Hockney to create the collages that he calls *joiners* (Zelnik-Manor and Perona 2007; Nomura, Zhang, and Nayar 2007). Creating such collages, which show visible seams and inconsistencies that add to the artistic effect, is popular on websites such as Flickr, where they

more commonly go under the name *panography* (Section 8.1.2). Translation and rotation are also usually adequate motion models to compensate for small camera motions in applications such as photo and video stabilization and merging (Exercise 8.1 and Section 9.2.1).

In Section 2.1.4, we saw how the mapping between two cameras viewing a common plane can be described using a 3×3 homography (2.71). Consider the matrix $\mathbf{M}_{10}$ that arises when mapping a pixel in one image to a 3D point and then back onto a second image,

$$\tilde{\mathbf{x}}_1 \sim \tilde{\mathbf{P}}_1 \tilde{\mathbf{P}}_0^{-1} \tilde{\mathbf{x}}_0 = \mathbf{M}_{10} \tilde{\mathbf{x}}_0. \quad (8.33)$$

When the last row of the $\mathbf{P}_0$ matrix is replaced with a plane equation $\hat{\mathbf{n}}_0 \cdot \mathbf{p} + c_0$ and points are assumed to lie on this plane, i.e., their disparity is $d_0 = 0$, we can ignore the last column of $\mathbf{M}_{10}$ and also its last row, since we do not care about the final z-buffer depth. The resulting homography matrix $\tilde{\mathbf{H}}_{10}$ (the upper left 3×3 sub-matrix of $\mathbf{M}_{10}$) describes the mapping between pixels in the two images,

$$\tilde{\mathbf{x}}_1 \sim \tilde{\mathbf{H}}_{10} \tilde{\mathbf{x}}_0. \quad (8.34)$$

This observation formed the basis of some of the earliest automated image stitching algorithms (Mann and Picard 1994; Szeliski 1994, 1996). Because reliable feature matching techniques had not yet been developed, these algorithms used direct pixel value matching, i.e., direct parametric motion estimation, as described in Section 9.2 and Equations (8.19–8.20).

More recent stitching algorithms first extract features and then match them up, often using robust techniques such as RANSAC (Section 8.1.4) to compute a good set of inliers. The final computation of the homography (8.34), i.e., the solution of the least squares fitting problem given pairs of corresponding features,

$$x_1 = \frac{(1 + h_{00})x_0 + h_{01}y_0 + h_{02}}{h_{20}x_0 + h_{21}y_0 + 1} \quad \text{and} \quad (8.35)$$

$$y_1 = \frac{h_{10}x_0 + (1 + h_{11})y_0 + h_{12}}{h_{20}x_0 + h_{21}y_0 + 1}, \quad (8.36)$$

uses iterative least squares, as described in Section 8.1.3 and Equations (8.21–8.23).

8.2.2 Application: Whiteboard and document scanning

The simplest image-stitching application is to stitch together a number of image scans taken on a flatbed scanner. Say you have a large map, or a piece of child's artwork, that is too large to fit on your scanner. Simply take multiple scans of the document, making sure to overlap the scans by a large enough amount to ensure that there are enough common features. Next, take successive pairs of images that you know overlap, extract features, match them up, and

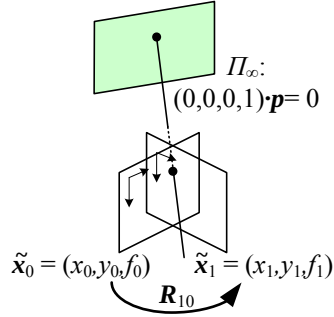


Figure 8.5 *Pure 3D camera rotation. The form of the homography (mapping) is particularly simple and depends only on the 3D rotation matrix and focal lengths.*

estimate the 2D rigid transform (2.16),

$$\mathbf{x}_{k+1} = \mathbf{R}_k \mathbf{x}_k + \mathbf{t}_k, \quad (8.37)$$

that best matches the features, using two-point RANSAC, if necessary, to find a good set of inliers. Then, on a final compositing surface (aligned with the first scan, for example), resample your images (Section 3.6.1) and average them together. Can you see any potential problems with this scheme?

One complication is that a 2D rigid transformation is non-linear in the rotation angle θ , so you will have to either use non-linear least squares or constrain $\mathbf{R}$ to be orthonormal, as described in Section 8.1.3.

A bigger problem lies in the pairwise alignment process. As you align more and more pairs, the solution may drift so that it is no longer globally consistent. In this case, a global optimization procedure, as described in Section 8.3, may be required. Such global optimization often requires a large system of non-linear equations to be solved, although in some cases, such as linearized homographies (Section 8.2.3) or similarity transforms (Section 8.1.2), regular least squares may be an option.

A slightly more complex scenario is when you take multiple overlapping handheld pictures of a whiteboard or other large planar object (He and Zhang 2005; Zhang and He 2007). Here, the natural motion model to use is a homography, although a more complex model that estimates the 3D rigid motion relative to the plane (plus the focal length, if unknown), could in principle be used.

8.2.3 Rotational panoramas

The most typical case for panoramic image stitching is when the camera undergoes a pure rotation. Think of standing at the rim of the Grand Canyon. Relative to the distant geometry in the scene, as you snap away, the camera is undergoing a pure rotation, which is equivalent to assuming that all points are very far from the camera, i.e., on the *plane at infinity* (Figure 8.5).¹³ Setting $\mathbf{t}_0 = \mathbf{t}_1 = 0$, we get the simplified 3×3 homography

$$\tilde{\mathbf{H}}_{10} = \mathbf{K}_1 \mathbf{R}_1 \mathbf{R}_0^{-1} \mathbf{K}_0^{-1} = \mathbf{K}_1 \mathbf{R}_{10} \mathbf{K}_0^{-1}, \quad (8.38)$$

where $\mathbf{K}_k = \text{diag}(f_k, f_k, 1)$ is the simplified camera intrinsic matrix (2.59), assuming that $c_x = c_y = 0$, i.e., we are indexing the pixels starting from the image center (Szeliski 1996). This can also be re-written as

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \sim \begin{bmatrix} f_1 & & \\ & f_1 & \\ & & 1 \end{bmatrix} \mathbf{R}_{10} \begin{bmatrix} f_0^{-1} & & \\ & f_0^{-1} & \\ & & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \quad (8.39)$$

or

$$\begin{bmatrix} x_1 \\ y_1 \\ f_1 \end{bmatrix} \sim \mathbf{R}_{10} \begin{bmatrix} x_0 \\ y_0 \\ f_0 \end{bmatrix}, \quad (8.40)$$

which reveals the simplicity of the mapping equations and makes all of the motion parameters explicit. Thus, instead of the general eight-parameter homography relating a pair of images, we get the three-, four-, or five-parameter *3D rotation* motion models corresponding to the cases where the focal length f is known, fixed, or variable (Szeliski and Shum 1997).¹⁴ Estimating the 3D rotation matrix (and, optionally, focal length) associated with each image is intrinsically more stable than estimating a homography with a full eight degrees of freedom, which makes this the method of choice for large-scale image stitching algorithms (Szeliski and Shum 1997; Shum and Szeliski 2000; Brown and Lowe 2007).

Given this representation, how do we update the rotation matrices to best align two overlapping images? Given a current estimate for the homography $\tilde{\mathbf{H}}_{10}$ in (8.38), the best way to

¹³In a more general (e.g., indoor) scene, if we want to ensure that there is no *parallax* (visible relative movement between objects at different depths), we need to rotate the camera around the lens's front *no-parallax point* (Littlefield 2006). This can be achieved by using a specialized panoramic rotation head with a built-in translation stage (Houghton 2013) or by determining the front nodal point using observations of collinear points—see Debevec, Wenger *et al.* (2002) and Szeliski (2010, Figure 6.7).

¹⁴An initial estimate of the focal lengths can be obtained using the intrinsic calibration techniques described in Section 11.1.3 or from EXIF tags.

update $\mathbf{R}_{10}$ is to prepend an *incremental* rotation matrix $\mathbf{R}(\boldsymbol{\omega})$ to the current estimate $\mathbf{R}_{10}$ (Szeliski and Shum 1997; Shum and Szeliski 2000),

$$\tilde{\mathbf{H}}(\boldsymbol{\omega}) = \mathbf{K}_1 \mathbf{R}(\boldsymbol{\omega}) \mathbf{R}_{10} \mathbf{K}_0^{-1} = [\mathbf{K}_1 \mathbf{R}(\boldsymbol{\omega}) \mathbf{K}_1^{-1}] [\mathbf{K}_1 \mathbf{R}_{10} \mathbf{K}_0^{-1}] = \mathbf{D} \tilde{\mathbf{H}}_{10}. \quad (8.41)$$

Note that here we have written the update rule in the *compositional* form, where the incremental update $\mathbf{D}$ is *prepended* to the current homography $\tilde{\mathbf{H}}_{10}$. Using the small-angle approximation to $\mathbf{R}(\boldsymbol{\omega})$ given in (2.35), we can write the incremental update matrix as

$$\mathbf{D} = \mathbf{K}_1 \mathbf{R}(\boldsymbol{\omega}) \mathbf{K}_1^{-1} \approx \mathbf{K}_1 (\mathbf{I} + [\boldsymbol{\omega}]_{\times}) \mathbf{K}_1^{-1} = \begin{bmatrix} 1 & -\omega_z & f_1 \omega_y \\ \omega_z & 1 & -f_1 \omega_x \\ -\omega_y/f_1 & \omega_x/f_1 & 1 \end{bmatrix}. \quad (8.42)$$

Notice how there is now a nice one-to-one correspondence between the entries in the $\mathbf{D}$ matrix and the $h_{00}, \dots, h_{21}$ parameters used in Table 8.1 and Equation (8.19), i.e.,

$$(h_{00}, h_{01}, h_{02}, h_{10}, h_{11}, h_{12}, h_{20}, h_{21}) = (0, -\omega_z, f_1 \omega_y, \omega_z, 0, -f_1 \omega_x, -\omega_y/f_1, \omega_x/f_1). \quad (8.43)$$

We can therefore apply the chain rule to Equations (8.24 and 8.43) to obtain

$$\begin{bmatrix} \hat{x}' - x \\ \hat{y}' - y \end{bmatrix} = \begin{bmatrix} -xy/f_1 & f_1 + x^2/f_1 & -y \\ -(f_1 + y^2/f_1) & xy/f_1 & x \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}, \quad (8.44)$$

which give us the linearized update equations needed to estimate $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z)$.¹⁵ Notice that this update rule depends on the focal length f_1 of the *target* view and is independent of the focal length f_0 of the *template* view. This is because the compositional algorithm essentially makes small perturbations to the target. Once the incremental rotation vector $\boldsymbol{\omega}$ has been computed, the $\mathbf{R}_1$ rotation matrix can be updated using $\mathbf{R}_1 \leftarrow \mathbf{R}(\boldsymbol{\omega}) \mathbf{R}_1$.

The formulas for updating the focal length estimates are a little more involved and are given in Shum and Szeliski (2000). We will not repeat them here, since an alternative update rule, based on minimizing the difference between back-projected 3D rays, is given in Section 8.3.1. Figure 8.1a shows the alignment of four images under the 3D rotation motion model.

8.2.4 Gap closing

The techniques presented in this section can be used to estimate a series of rotation matrices and focal lengths, which can be chained together to create large panoramas. Unfortunately,

¹⁵This is the same as the rotational component of instantaneous rigid flow (Bergen, Anandan *et al.* 1992) and the update equations given by Szeliski and Shum (1997) and Shum and Szeliski (2000).

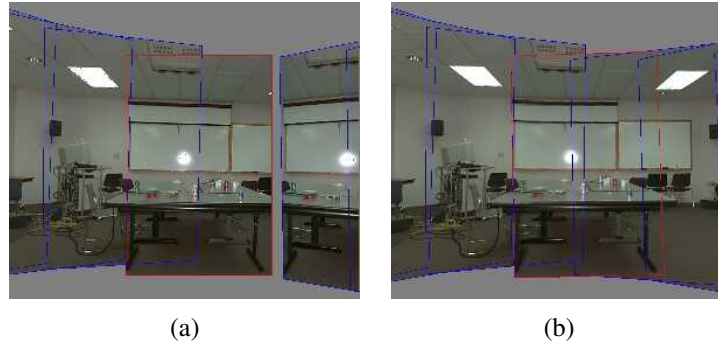


Figure 8.6 Gap closing (Szeliski and Shum 1997) © 1997 ACM: (a) A gap is visible when the focal length is wrong ($f = 510$). (b) No gap is visible for the correct focal length ($f = 468$).

because of accumulated errors, this approach will rarely produce a closed 360° panorama. Instead, there will invariably be either a gap or an overlap (Figure 8.6).

We can solve this problem by matching the first image in the sequence with the last one. The difference between the two rotation matrix estimates associated with the repeated first image indicates the amount of misregistration. This error can be distributed evenly across the whole sequence by taking the quotient of the two quaternions associated with these rotations and dividing this “error quaternion” by the number of images in the sequence (Szeliski and Shum 1997). We can also update the estimated focal length based on the amount of misregistration. To do this, we first convert the error quaternion into a *gap angle*, θ_g and then update the focal length using the equation $f' = f(1 - \theta_g/360^\circ)$.

Figure 8.6a shows the end of registered image sequence and the first image. There is a big gap between the last image and the first, which are in fact the same image. The gap is 32° because the wrong estimate of focal length ($f = 510$) was used. Figure 8.6b shows the registration after closing the gap with the correct focal length ($f = 468$). Notice that both mosaics show very little visual misregistration (except at the gap), yet Figure 8.6a has been computed using a focal length that has 9% error. Related approaches have been developed by Hartley (1994b), McMillan and Bishop (1995), Stein (1995), and Kang and Weiss (1997) to solve the focal length estimation problem using pure panning motion and cylindrical images.

Unfortunately, this gap-closing heuristic only works for the kind of “one-dimensional” panorama where the camera is continuously turning in the same direction. In Section 8.3, we describe a different approach to removing gaps and overlaps that works for arbitrary camera motions.

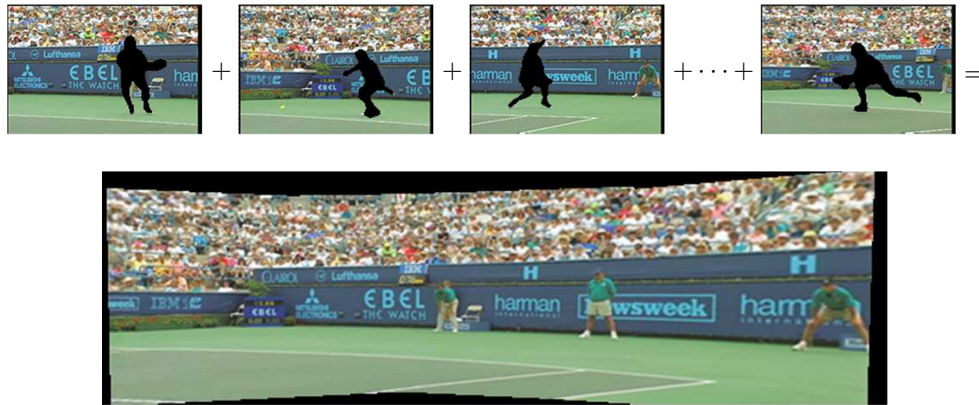


Figure 8.7 Video stitching the background scene to create a single sprite image that can be transmitted and used to re-create the background in each frame (Lee, Chen et al. 1997) © 1997 IEEE.

8.2.5 Application: Video summarization and compression

An interesting application of image stitching is the ability to summarize and compress videos taken with a panning camera. This application was first suggested by Teodosio and Bender (1993), who called their mosaic-based summaries *salient stills*. These ideas were then extended by Irani, Hsu, and Anandan (1995) and Irani and Anandan (1998) to additional applications, such as video compression and video indexing. While these early approaches used affine motion models and were therefore restricted to long focal lengths, the techniques were generalized by Lee, Chen et al. (1997) to full eight-parameter homographies and incorporated into the MPEG-4 video compression standard, where the stitched background layers were called *video sprites* (Figure 8.7).

While video stitching is in many ways a straightforward generalization of multiple-image stitching (Steedly, Pal, and Szeliski 2005; Baudisch, Tan et al. 2006), the potential presence of large amounts of independent motion, camera zoom, and the desire to visualize dynamic events impose additional challenges. For example, moving foreground objects can often be removed using *median filtering*. Alternatively, foreground objects can be extracted into a separate layer (Sawhney and Ayer 1996) and later composited back into the stitched panoramas, sometimes as multiple instances to give the impressions of a “Chronophotograph” (Massey and Bender 1996) and sometimes as video overlays (Irani and Anandan 1998). Videos can also be used to create animated *panoramic video textures* (Section 14.5.2), in which different portions of a panoramic scene are animated with independently moving video loops (Agar-

wala, Zheng *et al.* 2005; Rav-Acha, Pritch *et al.* 2005; Joshi, Mehta *et al.* 2012; Yan, Liu, and Furukawa 2017; He, Liao *et al.* 2017; Oh, Joo *et al.* 2017), or to shine “video flashlights” onto a composite mosaic of a scene (Sawhney, Arpa *et al.* 2002).

Video can also provide an interesting source of content for creating panoramas taken from moving cameras. While this invalidates the usual assumption of a single point of view (optical center), interesting results can still be obtained. For example, the VideoBrush system of Sawhney, Kumar *et al.* (1998) uses thin strips taken from the center of the image to create a panorama taken from a horizontally moving camera. This idea can be generalized to other camera motions and compositing surfaces using the concept of mosaics on an adaptive manifold (Peleg, Rousso *et al.* 2000), and also used to generate panoramic stereograms (Ishiguro, Yamamoto, and Tsuji 1992; Peleg, Ben-Ezra, and Pritch 2001).¹⁶ Related ideas have been used to create panoramic matte paintings for multiplane cel animation (Wood, Finkelstein *et al.* 1997), for creating stitched images of scenes with parallax (Kumar, Anandan *et al.* 1995), and as 3D representations of more complex scenes using *multiple-center-of-projection images* (Rademacher and Bishop 1998) and *multi-perspective panoramas* (Román, Garg, and Levoy 2004; Román and Lensch 2006; Agarwala, Agrawala *et al.* 2006; Kopf, Chen *et al.* 2010).

Another interesting variant on video-based panoramas is *concentric mosaics* (Section 14.3.3) (Shum and He 1999). Here, rather than trying to produce a single panoramic image, the complete original video is kept and used to re-synthesize views (from different camera origins) using ray remapping (light field rendering), thus endowing the panorama with a sense of 3D depth. The same dataset can also be used to explicitly reconstruct the depth using multi-baseline stereo (Ishiguro, Yamamoto, and Tsuji 1992; Peleg, Ben-Ezra, and Pritch 2001; Li, Shum *et al.* 2004; Zheng, Kang *et al.* 2007).

8.2.6 Cylindrical and spherical coordinates

An alternative to using homographies or 3D motions to align images is to first warp the images into *cylindrical* coordinates and then use a pure translational model to align them (Chen 1995; Szeliski 1996). Unfortunately, this only works if the images are all taken with a level camera or with a known tilt angle.

Assume for now that the camera is in its canonical position, i.e., its rotation matrix is the identity, $\mathbf{R} = \mathbf{I}$, so that the optical axis is aligned with the z -axis and the y -axis is aligned vertically. The 3D ray corresponding to an (x, y) pixel is therefore (x, y, f) .

¹⁶A similar technique was likely used in the Google Cardboard Camera, <https://blog.google/products/google-vr/cardboard-camera-ios>.

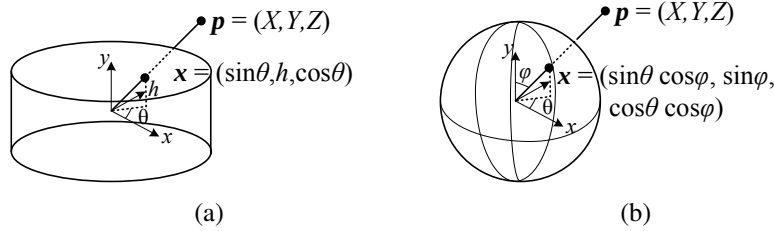


Figure 8.8 Projection from 3D to (a) cylindrical and (b) spherical coordinates.

We wish to project this image onto a *cylindrical surface* of unit radius (Szeliski 1996). Points on this surface are parameterized by an angle θ and a height h , with the 3D cylindrical coordinates corresponding to (θ, h) given by

$$(\sin \theta, h, \cos \theta) \propto (x, y, f), \quad (8.45)$$

as shown in Figure 8.8a. From this correspondence, we can compute the formula for the *warped* or *mapped* coordinates (Szeliski and Shum 1997),

$$x' = s\theta = s \tan^{-1} \frac{x}{f}, \quad (8.46)$$

$$y' = sh = s \frac{y}{\sqrt{x^2 + f^2}}, \quad (8.47)$$

where s is an arbitrary scaling factor (sometimes called the *radius* of the cylinder) that can be set to $s = f$ to minimize the distortion (scaling) near the center of the image.¹⁷ The inverse of this mapping equation is given by

$$x = f \tan \theta = f \tan \frac{x'}{s}, \quad (8.48)$$

$$y = h \sqrt{x^2 + f^2} = \frac{y'}{s} f \sqrt{1 + \tan^2 x'/s} = f \frac{y'}{s} \sec \frac{x'}{s}. \quad (8.49)$$

Images can also be projected onto a *spherical surface* (Szeliski and Shum 1997), which is useful if the final panorama includes a full sphere or hemisphere of views, instead of just a cylindrical strip. In this case, the sphere is parameterized by two angles (θ, ϕ) , with 3D spherical coordinates given by

$$(\sin \theta \cos \phi, \sin \phi, \cos \theta \cos \phi) \propto (x, y, f), \quad (8.50)$$

¹⁷The scale can also be set to a larger or smaller value for the final compositing surface, depending on the desired output panorama resolution—see Section 8.4.



Figure 8.9 A cylindrical panorama (Szeliski and Shum 1997) © 1997 ACM: (a) two cylindrically warped images related by a horizontal translation; (b) part of a cylindrical panorama composited from a sequence of images.

as shown in Figure 8.8b.¹⁸ The correspondence between coordinates is now given by (Szeliski and Shum 1997):

$$x' = s\theta = s \tan^{-1} \frac{x}{f}, \quad (8.51)$$

$$y' = s\phi = s \tan^{-1} \frac{y}{\sqrt{x^2 + f^2}}, \quad (8.52)$$

while the inverse is given by

$$x = f \tan \theta = f \tan \frac{x'}{s}, \quad (8.53)$$

$$y = \sqrt{x^2 + f^2} \tan \phi = \tan \frac{y'}{s} f \sqrt{1 + \tan^2 x'/s} = f \tan \frac{y'}{s} \sec \frac{x'}{s}. \quad (8.54)$$

Note that it may be simpler to generate a scaled (x, y, z) direction from Equation (8.50) followed by a perspective division by z and a scaling by f .

Cylindrical image stitching algorithms are most commonly used when the camera is known to be level and only rotating around its vertical axis (Chen 1995). Under these conditions, images at different rotations are related by a pure horizontal translation.¹⁹ This makes it attractive as an initial class project in an introductory computer vision course, since the full complexity of the perspective alignment algorithm (Sections 8.1, 9.2, and 8.2.3) can be avoided. Figure 8.9 shows how two cylindrically warped images from a leveled rotational panorama are related by a pure translation (Szeliski and Shum 1997).

¹⁸Note that these are not the usual spherical coordinates, first presented in Equation (2.8). Here, the y -axis points at the north pole instead of the z -axis, since we are used to viewing images taken horizontally, i.e., with the y -axis pointing in the direction of the gravity vector.

¹⁹Small vertical tilts can sometimes be compensated for with vertical translations.

Professional panoramic photographers often use pan-tilt heads that make it easy to control the tilt and to stop at specific *detents* in the rotation angle. Motorized rotation heads are also sometimes used for the acquisition of larger panoramas (Kopf, Uyttendaele *et al.* 2007).²⁰ Not only do they ensure a uniform coverage of the visual field with a desired amount of image overlap but they also make it possible to stitch the images using cylindrical or spherical coordinates and pure translations. In this case, pixel coordinates (x, y, f) must first be rotated using the known tilt and panning angles before being projected into cylindrical or spherical coordinates (Chen 1995). Having a roughly known panning angle also makes it easier to compute the alignment, as the rough relative positioning of all the input images is known ahead of time, enabling a reduced search range for alignment. Figure 8.1b shows a full 3D rotational panorama unwrapped onto the surface of a sphere (Szeliski and Shum 1997).

One final coordinate mapping worth mentioning is the *polar* mapping, where the north pole lies along the optical axis rather than the vertical axis,

$$(\cos \theta \sin \phi, \sin \theta \sin \phi, \cos \phi) = s(x, y, z). \quad (8.55)$$

In this case, the mapping equations become

$$x' = s\phi \cos \theta = s \frac{x}{r} \tan^{-1} \frac{r}{z}, \quad (8.56)$$

$$y' = s\phi \sin \theta = s \frac{y}{r} \tan^{-1} \frac{r}{z}, \quad (8.57)$$

where $r = \sqrt{x^2 + y^2}$ is the *radial distance* in the (x, y) plane and $s\phi$ plays a similar role in the (x', y') plane. This mapping provides an attractive visualization surface for certain kinds of wide-angle panoramas and is also a good model for the distortion induced by *fish-eye lenses*, as discussed in Section 2.1.5. Note how for small values of (x, y) , the mapping equations reduce to $x' \approx sx/z$, which suggests that s plays a role similar to the focal length f .

8.3 Global alignment

So far, we have discussed how to register pairs of images using a variety of motion models. In most applications, we are given more than a single pair of images to register. The goal is then to find a globally consistent set of alignment parameters that minimize the misregistration between all pairs of images (Szeliski and Shum 1997; Shum and Szeliski 2000; Sawhney and Kumar 1999; Coorg and Teller 2000).

²⁰See also <https://gigapan.org>.

In this section, we extend the pairwise matching criteria (8.2, 9.1, and 9.43) to a global energy function that involves all of the per-image pose parameters (Section 8.3.1). Once we have computed the global alignment, we often need to perform *local adjustments*, such as *parallax removal*, to reduce double images and blurring due to local misregistrations (Section 8.3.2). Finally, if we are given an unordered set of images to register, we need to discover which images go together to form one or more panoramas. This process of *panorama recognition* is described in Section 8.3.3.

8.3.1 Bundle adjustment

One way to register a large number of images is to add new images to the panorama one at a time, aligning the most recent image with the previous ones already in the collection (Szeliski and Shum 1997) and discovering, if necessary, which images it overlaps (Sawhney and Kumar 1999). In the case of 360° panoramas, accumulated error may lead to the presence of a gap (or excessive overlap) between the two ends of the panorama, which can be fixed by stretching the alignment of all the images using a process called *gap closing* (Section 8.2.4). However, a better alternative is to simultaneously align all the images using a least-squares framework to correctly distribute any misregistration errors.

The process of simultaneously adjusting pose parameters and 3D point locations for a large collection of overlapping images is called *bundle adjustment* in the photogrammetry community (Triggs, McLauchlan *et al.* 1999). In computer vision, it was first applied to the general structure from motion problem (Szeliski and Kang 1994) and then later specialized for panoramic image stitching (Shum and Szeliski 2000; Sawhney and Kumar 1999; Coorg and Teller 2000).

In this section, we formulate the problem of global alignment using a feature-based approach, since this results in a simpler system. An equivalent direct approach can be obtained either by dividing images into patches and creating a virtual feature correspondence for each one (Shum and Szeliski 2000) or by replacing the per-feature error metrics with per-pixel metrics (Irani and Anandan 1999).

Before we describe this in more details, we should mention that a simpler, although less accurate, approach is to compute pairwise rotation estimates between overlapping images, and to then use a *rotation averaging* approach to estimate a global rotation for each camera (Hartley, Trumpf *et al.* 2013). However, since the measurement errors in each feature point location are not being counted correctly, as is the case in bundle adjustment, the solution will not have the same theoretical optimality.

Consider the feature-based alignment problem given in Equation (8.2), i.e.,

$$E_{\text{pairwise-LS}} = \sum_i \|\mathbf{r}_i\|^2 = \|\tilde{\mathbf{x}}'_i(\mathbf{x}_i; \mathbf{p}) - \hat{\mathbf{x}}'_i\|^2. \quad (8.58)$$

For multi-image alignment, instead of having a single collection of pairwise feature correspondences, $\{(\mathbf{x}_i, \hat{\mathbf{x}}'_i)\}$, we have a collection of n features, with the location of the i th feature point in the j th image denoted by $\mathbf{x}_{ij}$ and its scalar confidence (i.e., inverse variance) denoted by c_{ij} .²¹ Each image also has some associated pose parameters.

In this section, we assume that this pose consists of a rotation matrix $\mathbf{R}_j$ and a focal length f_j , although formulations in terms of homographies are also possible (Szeliski and Shum 1997; Sawhney and Kumar 1999). The equation mapping a 3D point $\mathbf{x}_i$ into a point $\mathbf{x}_{ij}$ in frame j can be re-written from Equations (2.68) and (8.38) as

$$\tilde{\mathbf{x}}_{ij} \sim \mathbf{K}_j \mathbf{R}_j \mathbf{x}_i \quad \text{and} \quad \mathbf{x}_i \sim \mathbf{R}_j^{-1} \mathbf{K}_j^{-1} \tilde{\mathbf{x}}_{ij}, \quad (8.59)$$

where $\mathbf{K}_j = \text{diag}(f_j, f_j, 1)$ is the simplified form of the calibration matrix. The motion mapping a point $\mathbf{x}_{ij}$ from frame j into a point $\mathbf{x}_{ik}$ in frame k is similarly given by

$$\tilde{\mathbf{x}}_{ik} \sim \tilde{\mathbf{H}}_{kj} \tilde{\mathbf{x}}_{ij} = \mathbf{K}_k \mathbf{R}_k \mathbf{R}_j^{-1} \mathbf{K}_j^{-1} \tilde{\mathbf{x}}_{ij}. \quad (8.60)$$

Given an initial set of $\{(\mathbf{R}_j, f_j)\}$ estimates obtained from chaining pairwise alignments, how do we refine these estimates?

One approach is to directly extend the pairwise energy $E_{\text{pairwise-LS}}$ (8.58) to a multiview formulation,

$$E_{\text{all-pairs-2D}} = \sum_i \sum_{jk} c_{ij} c_{ik} \|\tilde{\mathbf{x}}_{ik}(\tilde{\mathbf{x}}_{ij}; \mathbf{R}_j, f_j, \mathbf{R}_k, f_k) - \hat{\mathbf{x}}_{ik}\|^2, \quad (8.61)$$

where the $\tilde{\mathbf{x}}_{ik}$ function is the *predicted* location of feature i in frame k given by (8.60), $\hat{\mathbf{x}}_{ij}$ is the *observed* location, and the “2D” in the subscript indicates that an image-plane error is being minimized (Shum and Szeliski 2000). Note that since $\tilde{\mathbf{x}}_{ik}$ depends on the $\hat{\mathbf{x}}_{ij}$ observed value, we actually have an *errors-in-variable* problem, which in principle requires more sophisticated techniques than least squares to solve (Van Huffel and Lemmerling 2002; Matei and Meer 2006). However, in practice, if we have enough features, we can directly minimize the above quantity using regular non-linear least squares and obtain an accurate multi-frame alignment.

While this approach works pretty well, it suffers from two potential disadvantages. First, because a summation is taken over all pairs with corresponding features, features that are

²¹ Features that are not seen in image j have $c_{ij} = 0$. We can also use 2×2 inverse covariance matrices Σ_{ij}^{-1} in place of c_{ij} , as shown in Equation (8.11).

observed many times are overweighted in the final solution. (In effect, a feature observed m times gets counted $\binom{m}{2}$ times instead of m times.) Second, the derivatives of $\tilde{\mathbf{x}}_{ik}$ with respect to the $\{(\mathbf{R}_j, f_j)\}$ are a little cumbersome, although using the incremental correction to $\mathbf{R}_j$ introduced in Section 8.2.3 makes this more tractable.

An alternative way to formulate the optimization is to use true bundle adjustment, i.e., to solve not only for the pose parameters $\{(\mathbf{R}_j, f_j)\}$ but also for the 3D point positions $\{\mathbf{x}_i\}$,

$$E_{\text{BA-2D}} = \sum_i \sum_j c_{ij} \|\tilde{\mathbf{x}}_{ij}(\mathbf{x}_i; \mathbf{R}_j, f_j) - \hat{\mathbf{x}}_{ij}\|^2, \quad (8.62)$$

where $\tilde{\mathbf{x}}_{ij}(\mathbf{x}_i; \mathbf{R}_j, f_j)$ is given by (8.59). The disadvantage of full bundle adjustment is that there are more variables to solve for, so each iteration and also the overall convergence may be slower. (Imagine how the 3D points need to “shift” each time some rotation matrices are updated.) However, the computational complexity of each linearized Gauss–Newton step can be reduced using sparse matrix techniques (Section 11.4.3) (Szeliski and Kang 1994; Triggs, McLauchlan *et al.* 1999; Hartley and Zisserman 2004).

An alternative formulation is to minimize the error in 3D projected ray directions (Shum and Szeliski 2000), i.e.,

$$E_{\text{BA-3D}} = \sum_i \sum_j c_{ij} \|\tilde{\mathbf{x}}_i(\hat{\mathbf{x}}_{ij}; \mathbf{R}_j, f_j) - \mathbf{x}_i\|^2, \quad (8.63)$$

where $\tilde{\mathbf{x}}_i(\hat{\mathbf{x}}_{ij}; \mathbf{R}_j, f_j)$ is given by the second half of (8.59). This has no particular advantage over (8.62). In fact, since errors are being minimized in 3D ray space, there is a bias towards estimating longer focal lengths, since the angles between rays become smaller as f increases.

However, if we eliminate the 3D rays $\mathbf{x}_i$, we can derive a pairwise energy formulated in 3D ray space (Shum and Szeliski 2000),

$$E_{\text{all-pairs-3D}} = \sum_i \sum_{jk} c_{ij} c_{ik} \|\tilde{\mathbf{x}}_i(\hat{\mathbf{x}}_{ij}; \mathbf{R}_j, f_j) - \tilde{\mathbf{x}}_i(\hat{\mathbf{x}}_{ik}; \mathbf{R}_k, f_k)\|^2. \quad (8.64)$$

This results in the simplest set of update equations (Shum and Szeliski 2000), since the f_k can be folded into the creation of the homogeneous coordinate vector as in Equation (8.40). Thus, even though this formula over-weights features that occur more frequently, it is the method used by Shum and Szeliski (2000) and Brown, Szeliski, and Winder (2005). To reduce the bias towards longer focal lengths, we multiply each residual (3D error) by $\sqrt{f_j f_k}$, which is similar to projecting the 3D rays into a “virtual camera” of intermediate focal length.

Up vector selection. As mentioned above, there exists a global ambiguity in the pose of the 3D cameras computed by the above methods. While this may not appear to matter, people

prefer that the final stitched image is “upright” rather than twisted or tilted. More concretely, people are used to seeing photographs displayed so that the vertical (gravity) axis points straight up in the image. Consider how you usually shoot photographs: while you may pan and tilt the camera any which way, you usually keep the horizontal edge of your camera (its x -axis) parallel to the ground plane (perpendicular to the world gravity direction).

Mathematically, this constraint on the rotation matrices can be expressed as follows. Recall from Equation (8.59) that the 3D to 2D projection is given by

$$\tilde{\mathbf{x}}_{ik} \sim \mathbf{K}_k \mathbf{R}_k \mathbf{x}_i. \quad (8.65)$$

We wish to post-multiply each rotation matrix $\mathbf{R}_k$ by a global rotation $\mathbf{R}_G$ such that the projection of the global y -axis, $\hat{j} = (0, 1, 0)$ is perpendicular to the image x -axis, $\hat{i} = (1, 0, 0)$.²²

This constraint can be written as

$$\hat{i}^T \mathbf{R}_k \mathbf{R}_G \hat{j} = 0 \quad (8.66)$$

(note that the scaling by the calibration matrix is irrelevant here). This is equivalent to requiring that the first row of $\mathbf{R}_k$, $\mathbf{r}_{k0} = \hat{i}^T \mathbf{R}_k$ be perpendicular to the second column of $\mathbf{R}_G$, $\mathbf{r}_{G1} = \mathbf{R}_G \hat{j}$. This set of constraints (one per input image) can be written as a least squares problem,

$$\mathbf{r}_{G1} = \arg \min_{\mathbf{r}} \sum_k (\mathbf{r}^T \mathbf{r}_{k0})^2 = \arg \min_{\mathbf{r}} \mathbf{r}^T \left[\sum_k \mathbf{r}_{k0} \mathbf{r}_{k0}^T \right] \mathbf{r}. \quad (8.67)$$

Thus, $\mathbf{r}_{G1}$ is the smallest eigenvector of the *scatter* or *moment* matrix spanned by the individual camera rotation x -vectors, which should generally be of the form $(c, 0, s)$ when the cameras are upright.

To fully specify the $\mathbf{R}_G$ global rotation, we need to specify one additional constraint. This is related to the *view selection* problem discussed in Section 8.4.1. One simple heuristic is to prefer the average z -axis of the individual rotation matrices, $\bar{\mathbf{k}} = \sum_k \hat{\mathbf{k}}^T \mathbf{R}_k$ to be close to the world z -axis, $\mathbf{r}_{G2} = \mathbf{R}_G \hat{\mathbf{k}}$. We can therefore compute the full rotation matrix $\mathbf{R}_G$ in three steps:

1. $\mathbf{r}_{G1} = \min \text{eigenvector} (\sum_k \mathbf{r}_{k0} \mathbf{r}_{k0}^T)$;
2. $\mathbf{r}_{G0} = \mathcal{N}((\sum_k \mathbf{r}_{k2}) \times \mathbf{r}_{G1})$;
3. $\mathbf{r}_{G2} = \mathbf{r}_{G0} \times \mathbf{r}_{G1}$,

where $\mathcal{N}(\mathbf{v}) = \mathbf{v} / \|\mathbf{v}\|$ normalizes a vector $\mathbf{v}$.

²²Note that here we use the convention common in computer graphics that the vertical world axis corresponds to y . This is a natural choice if we wish the rotation matrix associated with a “regular” image taken horizontally to be the identity, rather than a 90° rotation around the x -axis.

8.3.2 Parallax removal

Once we have optimized the global orientations and focal lengths of our cameras, we may find that the images are still not perfectly aligned, i.e., the resulting stitched image looks blurry or ghosted in some places. This can be caused by a variety of factors, including unmodeled radial distortion, 3D parallax (failure to rotate the camera around its front nodal point), small scene motions such as waving tree branches, and large-scale scene motions such as people moving in and out of pictures.

Each of these problems can be treated with a different approach. Radial distortion can be estimated (potentially ahead of time) using one of the techniques discussed in Section 2.1.5. For example, the *plumb-line method* (Brown 1971; Kang 2001; El-Melegy and Farag 2003) adjusts radial distortion parameters until slightly curved lines become straight, while mosaic-based approaches adjust them until misregistration is reduced in image overlap areas (Stein 1997; Sawhney and Kumar 1999).

3D parallax can be handled by doing a full 3D bundle adjustment, i.e., by replacing the projection Equation (8.59) used in Equation (8.62) with Equation (2.68), which models camera translations. The 3D positions of the matched feature points and cameras can then be simultaneously recovered, although this can be significantly more expensive than parallax-free image registration. Once the 3D structure has been recovered, the scene could (in theory) be projected to a single (central) viewpoint that contains no parallax. However, to do this, dense *stereo* correspondence needs to be performed (Section 12.3) (Li, Shum *et al.* 2004; Zheng, Kang *et al.* 2007), which may not be possible if the images contain only partial overlap. In that case, it may be necessary to correct for parallax only in the overlap areas, which can be accomplished using a *multi-perspective plane sweep* (MPPS) algorithm (Kang, Szeliski, and Uyttendaele 2004; Uyttendaele, Criminisi *et al.* 2004).

When the motion in the scene is very large, i.e., when objects appear and disappear completely, a sensible solution is to simply *select* pixels from only one image at a time as the source for the final composite (Milgram 1977; Davis 1998; Agarwala, Dontcheva *et al.* 2004), as discussed in Section 8.4.2. However, when the motion is reasonably small (on the order of a few pixels), general 2D motion estimation (optical flow) can be used to perform an appropriate correction before blending using a process called *local alignment* (Shum and Szeliski 2000; Kang, Uyttendaele *et al.* 2003). This same process can also be used to compensate for radial distortion and 3D parallax, although it uses a weaker motion model than explicitly modeling the source of error and may, therefore, fail more often or introduce unwanted distortions.

The local alignment technique introduced by Shum and Szeliski (2000) starts with the global bundle adjustment (8.64) used to optimize the camera poses. Once these have been

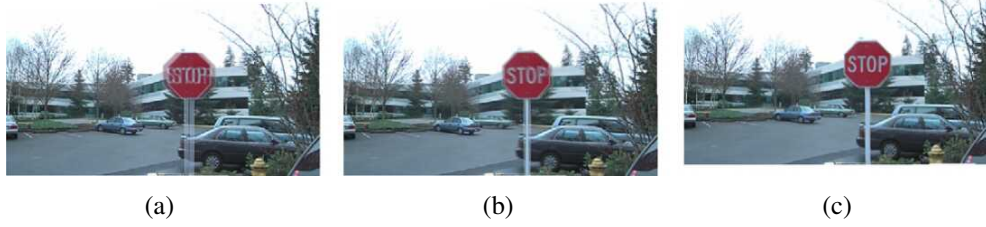


Figure 8.10 *Deghosting a mosaic with motion parallax (Shum and Szeliski 2000) © 2000 IEEE: (a) composite with parallax; (b) after a single deghosting step (patch size 32); (c) after multiple steps (sizes 32, 16 and 8).*

estimated, the *desired* location of a 3D point $\mathbf{x}_i$ can be estimated as the *average* of the back-projected 3D locations,

$$\bar{\mathbf{x}}_i \sim \sum_j c_{ij} \tilde{\mathbf{x}}_i(\tilde{\mathbf{x}}_{ij}; \mathbf{R}_j, f_j) / \sum_j c_{ij}, \quad (8.68)$$

which can be projected into each image j to obtain a *target location* $\bar{\mathbf{x}}_{ij}$. The difference between the target locations $\bar{\mathbf{x}}_{ij}$ and the original features $\mathbf{x}_{ij}$ provide a set of local motion estimates

$$\mathbf{u}_{ij} = \bar{\mathbf{x}}_{ij} - \mathbf{x}_{ij}, \quad (8.69)$$

which can be interpolated to form a dense correction field $\mathbf{u}_j(\mathbf{x}_j)$. In their system, Shum and Szeliski (2000) use an *inverse warping* algorithm where the sparse $-\mathbf{u}_{ij}$ values are placed at the new target locations $\bar{\mathbf{x}}_{ij}$, interpolated using bilinear kernel functions (Nielson 1993) and then added to the original pixel coordinates when computing the warped (corrected) image. To get a reasonably dense set of features to interpolate, Shum and Szeliski (2000) place a feature point at the center of each patch (the patch size controls the smoothness in the local alignment stage), rather than relying on features extracted using an interest operator (Figure 8.10).

An alternative approach to motion-based deghosting was proposed by Kang, Uyttendaele *et al.* (2003), who estimate dense optical flow between each input image and a central *reference* image. The accuracy of the flow vector is checked using a photo-consistency measure before a given warped pixel is considered valid and is used to compute a high dynamic range radiance estimate, which is the goal of their overall algorithm. The requirement for a reference image makes their approach less applicable to general image mosaicing, although an extension to this case could certainly be envisaged.

The idea of combining *global* parametric warps with *local* mesh-based warps or multiple motion models to compensate for parallax has been refined in a number of more recent papers

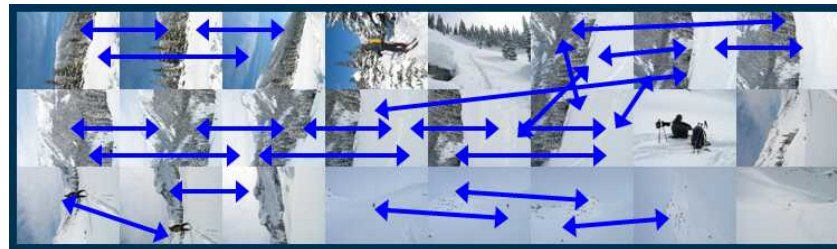
(Zaragoza, Chin *et al.* 2013; Zhang and Liu 2014; Lin, Pankanti *et al.* 2015; Lin, Jiang *et al.* 2016; Herrmann, Wang *et al.* 2018b; Lee and Sim 2020). Some of these papers use *content-preserving warps* (Liu, Gleicher *et al.* 2009) for their local deformations, while others include a rolling shutter model (Zhuang and Tran 2020).

8.3.3 Recognizing panoramas

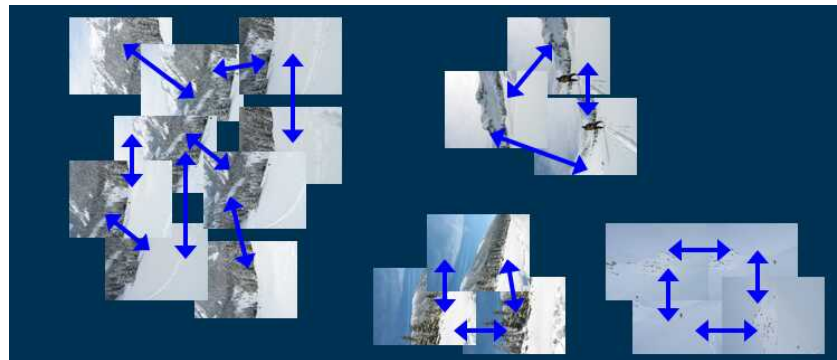
The final piece needed to perform fully automated image stitching is a technique to recognize which images actually go together, which Brown and Lowe (2007) call *recognizing panoramas*. If the user takes images in sequence so that each image overlaps its predecessor and also specifies the first and last images to be stitched, bundle adjustment combined with the process of *topology inference* can be used to automatically assemble a panorama (Sawhney and Kumar 1999). However, users often jump around when taking panoramas, e.g., they may start a new row on top of a previous one, jump back to take a repeat shot, or create 360° panoramas where end-to-end overlaps need to be discovered. Furthermore, the ability to discover multiple panoramas taken by a user over an extended period of time can be a big convenience.

To recognize panoramas, Brown and Lowe (2007) first find all pairwise image overlaps using a feature-based method and then find connected components in the overlap graph to “recognize” individual panoramas (Figure 8.11). The feature-based matching stage first extracts scale invariant feature transform (SIFT) feature locations and feature descriptors (Lowe 2004) from all the input images and places them in an indexing structure, as described in Section 7.1.3. For each image pair under consideration, the nearest matching neighbor is found for each feature in the first image, using the indexing structure to rapidly find candidates and then comparing feature descriptors to find the best match. RANSAC is used to find a set of *inlier* matches; pairs of matches are used to hypothesize similarity motion models that are then used to count the number of inliers. A RANSAC algorithm tailored specifically for rotational panoramas is described by Brown, Hartley, and Nistér (2007).

In practice, the most difficult part of getting a fully automated stitching algorithm to work is deciding which pairs of images actually correspond to the same parts of the scene. Repeated structures such as windows (Figure 8.12) can lead to false matches when using a feature-based approach. One way to mitigate this problem is to perform a direct pixel-based comparison between the registered images to determine if they actually are different views of the same scene. Unfortunately, this heuristic may fail if there are moving objects in the scene (Figure 8.13). While there is no magic bullet for this problem, short of full scene understanding, further improvements can likely be made by applying domain-specific heuristics, such as priors on typical camera motions as well as machine learning techniques



(a)



(b)



(c)

Figure 8.11 *Recognizing panoramas (Brown, Szeliski, and Winder 2005), figures courtesy of Matthew Brown: (a) input images with pairwise matches; (b) images grouped into connected components (panoramas); (c) individual panoramas registered and blended into stitched composites.*

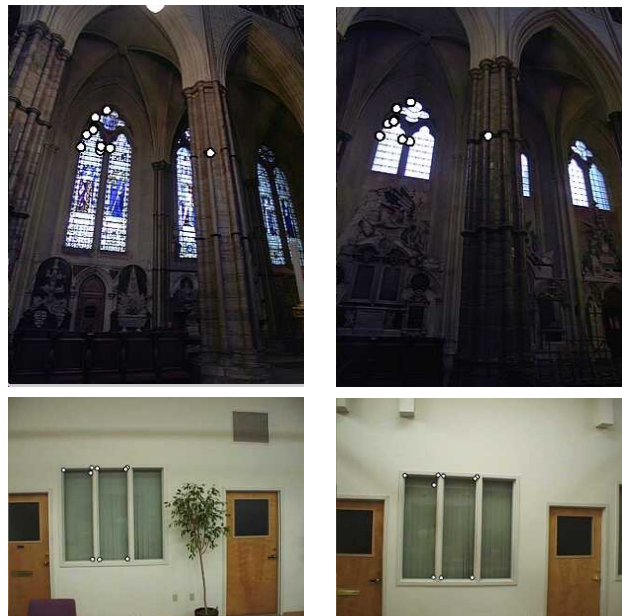


Figure 8.12 Matching errors (*Brown, Szeliski, and Winder 2004*): accidental matching of several features can lead to matches between pairs of images that do not actually overlap.



Figure 8.13 Validation of image matches by direct pixel error comparison can fail when the scene contains moving objects (*Uyttendaele, Eden, and Szeliski 2001*) © 2001 IEEE.

applied to the problem of match validation.

8.4 Compositing

Once we have registered all of the input images with respect to each other, we need to decide how to produce the final stitched mosaic image. This involves selecting a final compositing surface (flat, cylindrical, spherical, etc.) and view (reference image). It also involves selecting which pixels contribute to the final composite and how to optimally blend these pixels to minimize visible seams, blur, and ghosting.

In this section, we review techniques that address these problems, namely compositing surface parameterization, pixel and seam selection, blending, and exposure compensation. Our emphasis is on fully automated approaches to the problem. Since the creation of high-quality panoramas and composites is as much an artistic endeavor as a computational one, various interactive tools have been developed to assist this process (Agarwala, Dontcheva *et al.* 2004; Li, Sun *et al.* 2004; Rother, Kolmogorov, and Blake 2004). Some of these are covered in more detail in Section 10.4.

8.4.1 Choosing a compositing surface

The first choice to be made is how to represent the final image. If only a few images are stitched together, a natural approach is to select one of the images as the *reference* and to then warp all of the other images into its reference coordinate system. The resulting composite is sometimes called a *flat* panorama, since the projection onto the final surface is still a perspective projection, and hence straight lines remain straight (which is often a desirable attribute).²³

For larger fields of view, however, we cannot maintain a flat representation without excessively stretching pixels near the border of the image. (In practice, flat panoramas start to look severely distorted once the field of view exceeds 90° or so.) The usual choice for compositing larger panoramas is to use a cylindrical (Chen 1995; Szeliski 1996) or spherical (Szeliski and Shum 1997) projection, as described in Section 8.2.6. In fact, any surface used for *environment mapping* in computer graphics can be used, including a *cube map*, which represents the full viewing sphere with the six square faces of a cube (Greene 1986; Szeliski and Shum 1997). Cartographers have also developed a number of alternative methods for representing the globe (Bugayevskiy and Snyder 1995).

²³Techniques have also been developed to straighten curved lines in cylindrical and spherical panoramas (Carroll, Agrawala, and Agarwala 2009; Kopf, Lischinski *et al.* 2009; Carroll, Agarwala, and Agrawala 2010).

The choice of parameterization is somewhat application-dependent and involves a trade-off between keeping the local appearance undistorted (e.g., keeping straight lines straight) and providing a reasonably uniform sampling of the environment. Automatically making this selection and smoothly transitioning between representations based on the extent of the panorama is discussed in Kopf, Uyttendaele *et al.* (2007). A recent trend in panoramic photography has been the use of stereographic projections looking down at the ground (in an outdoor scene) to create “little planet” renderings.²⁴

View selection. Once we have chosen the output parameterization, we still need to determine which part of the scene will be *centered* in the final view. As mentioned above, for a flat composite, we can choose one of the images as a reference. Often, a reasonable choice is the one that is geometrically most central. For example, for rotational panoramas represented as a collection of 3D rotation matrices, we can choose the image whose z -axis is closest to the average z -axis (assuming a reasonable field of view). Alternatively, we can use the average z -axis (or quaternion, but this is trickier) to define the reference rotation matrix.

For larger, e.g., cylindrical or spherical, panoramas, we can use the same heuristic if a subset of the viewing sphere has been imaged. In the case of full 360° panoramas, a better choice is to choose the middle image from the sequence of inputs, or sometimes the first image, assuming this contains the object of greatest interest. In all of these cases, having the user control the final view is often highly desirable. If the “up vector” computation described in Section 8.3.1 is working correctly, this can be as simple as panning over the image or setting a vertical “center line” for the final panorama.

Coordinate transformations. After selecting the parameterization and reference view, we still need to compute the mappings between the input and output pixels coordinates.

If the final compositing surface is flat (e.g., a single plane or the face of a cube map) and the input images have no radial distortion, the coordinate transformation is the simple homography described by Equation (8.38). This kind of warping can be performed in graphics hardware by appropriately setting texture mapping coordinates and rendering a single quadrilateral.

If the final composite surface has some other analytic form (e.g., cylindrical or spherical), we need to convert every pixel in the final panorama into a viewing ray (3D point) and then map it back into each image according to the projection (and optionally radial distortion) equations. This process can be made more efficient by precomputing some lookup tables,

²⁴These are inspired by *The Little Prince* by Antoine De Saint-Exupéry. Go to <https://www.flickr.com> and search for “little planet projection”.

e.g., the partial trigonometric functions needed to map cylindrical or spherical coordinates to 3D coordinates or the radial distortion field at each pixel. It is also possible to accelerate this process by computing exact pixel mappings on a coarser grid and then interpolating these values.

When the final compositing surface is a texture-mapped polyhedron, a slightly more sophisticated algorithm must be used. Not only do the 3D and texture map coordinates have to be properly handled, but a small amount of *overdraw* outside the triangle footprints in the texture map is necessary, to ensure that the texture pixels being interpolated during 3D rendering have valid values (Szeliski and Shum 1997).

Sampling issues. While the above computations can yield the correct (fractional) pixel addresses in each input image, we still need to pay attention to sampling issues. For example, if the final panorama has a lower resolution than the input images, prefiltering the input images is necessary to avoid aliasing. These issues have been extensively studied in both the image processing and computer graphics communities. The basic problem is to compute the appropriate prefilter, which depends on the distance (and arrangement) between neighboring samples in a source image. As discussed in Sections 3.5.2 and 3.6.1, various approximate solutions, such as MIP mapping (Williams 1983) or elliptically weighted Gaussian averaging (Greene and Heckbert 1986) have been developed in the graphics community. For highest visual quality, a higher order (e.g., cubic) interpolator combined with a spatially adaptive prefilter may be necessary (Wang, Kang *et al.* 2001). Under certain conditions, it may also be possible to produce images with a higher resolution than the input images using the process of *super-resolution* (Section 10.3).

8.4.2 Pixel selection and weighting (deghosting)

Once the source pixels have been mapped onto the final composite surface, we must still decide how to blend them in order to create an attractive-looking panorama. If all of the images are in perfect registration and identically exposed, this is an easy problem, i.e., any pixel or combination will do. However, for real images, visible seams (due to exposure differences), blurring (due to misregistration), or ghosting (due to moving objects) can occur.

Creating clean, pleasing-looking panoramas involves both deciding which pixels to use and how to weight or blend them. The distinction between these two stages is a little fluid, since per-pixel weighting can be thought of as a combination of selection and blending. In this section, we discuss spatially varying weighting, pixel selection (seam placement), and then more sophisticated blending.

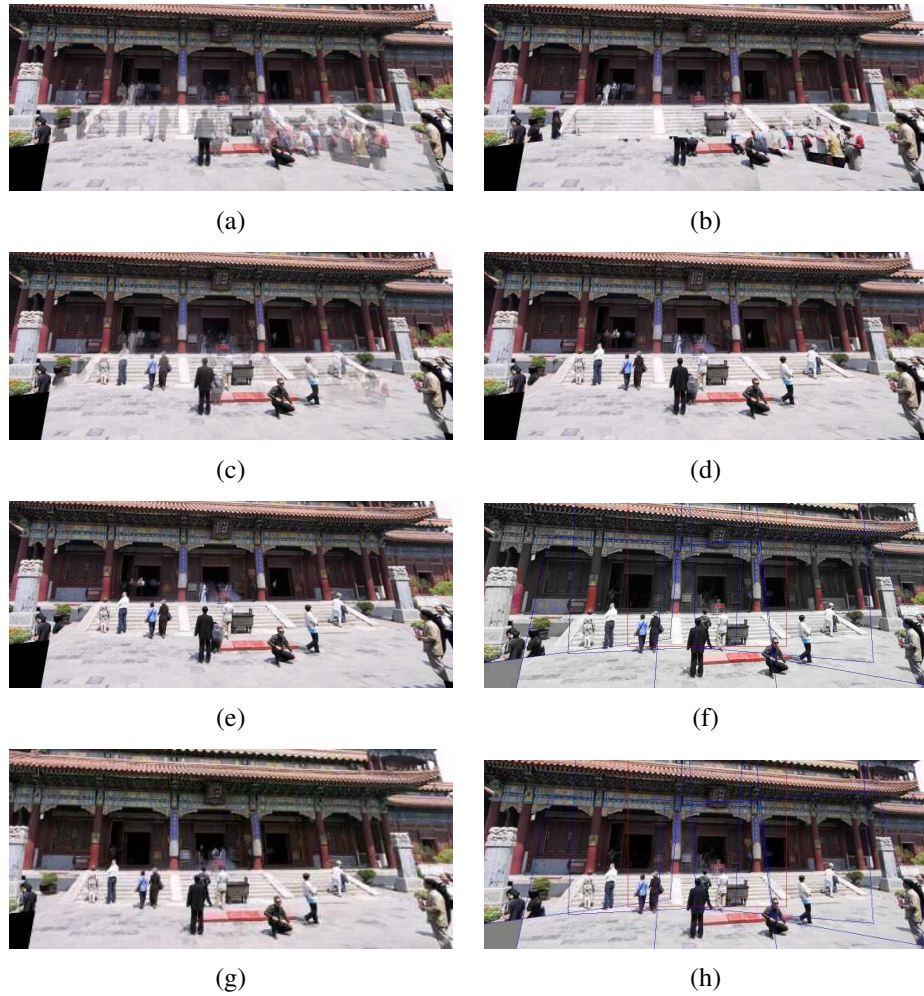


Figure 8.14 Final composites computed by a variety of algorithms (Szeliski 2006a): (a) average, (b) median, (c) feathered average, (d) p-norm $p = 10$, (e) Voronoi, (f) weighted ROD vertex cover with feathering, (g) graph cut seams with Poisson blending, and (h) with pyramid blending.

Feathering and center-weighting. The simplest way to create a final composite is to simply take an *average* value at each pixel,

$$C(\mathbf{x}) = \sum_k w_k(\mathbf{x}) \tilde{I}_k(\mathbf{x}) / \sum_k w_k(\mathbf{x}) , \quad (8.70)$$

where $\tilde{I}_k(\mathbf{x})$ are the *warped* (re-sampled) images and $w_k(\mathbf{x})$ is 1 at valid pixels and 0 elsewhere. On computer graphics hardware, this kind of summation can be performed in an *accumulation buffer* (using the *A* channel as the weight).

Simple averaging usually does not work very well, since exposure differences, misregistrations, and scene movement are all very visible (Figure 8.14a). If rapidly moving objects are the only problem, taking a *median* filter (which is a kind of pixel selection operator) can often be used to remove them (Figure 8.14b) (Irani and Anandan 1998). Conversely, center-weighting (discussed below) and *minimum likelihood* selection (Agarwala, Dontcheva *et al.* 2004) can sometimes be used to retain multiple copies of a moving object (Figure 8.17).

A better approach to averaging is to weight pixels near the center of the image more heavily and to down-weight pixels near the edges. When an image has some cutout regions, down-weighting pixels near the edges of both cutouts and the image is preferable. This can be done by computing a *distance map* or *grassfire transform*,

$$w_k(\mathbf{x}) = \arg \min_{\mathbf{y}} \{ \|\mathbf{y}\| \mid \tilde{I}_k(\mathbf{x} + \mathbf{y}) \text{ is invalid} \} , \quad (8.71)$$

where each valid pixel is tagged with its Euclidean distance to the nearest invalid pixel (Section 3.3.3). The Euclidean distance map can be efficiently computed using a two-pass raster algorithm (Danielsson 1980; Borgefors 1986).

Weighted averaging with a distance map is often called *feathering* (Szeliski and Shum 1997; Chen and Klette 1999; Uyttendaele, Eden, and Szeliski 2001) and does a reasonable job of blending over exposure differences. However, blurring and ghosting can still be problems (Figure 8.14c). Note that weighted averaging is *not* the same as compositing the individual images with the classic *over* operation (Porter and Duff 1984; Blinn 1994a), even when using the weight values (normalized to sum up to one) as *alpha* (translucency) channels. This is because the over operation attenuates the values from more distant surfaces and, hence, is not equivalent to a direct sum.

One way to improve feathering is to raise the distance map values to some large power, i.e., to use $w_k^p(\mathbf{x})$ in Equation (8.70). The weighted averages then become dominated by the larger values, i.e., they act somewhat like a *p-norm*. The resulting composite can often provide a reasonable tradeoff between visible exposure differences and blur (Figure 8.14d).

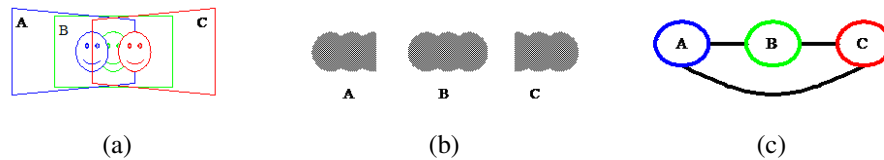


Figure 8.15 Computation of regions of difference (RODs) (Uyttendaele, Eden, and Szeliski 2001) © 2001 IEEE: (a) three overlapping images with a moving face; (b) corresponding RODs; (c) graph of coincident RODs.

In the limit as $p \rightarrow \infty$, only the pixel with the maximum weight is selected. This hard pixel selection process produces a visibility mask-sensitive variant of the familiar *Voronoi diagram*, which assigns each pixel to the nearest image center in the set (Wood, Finkelstein *et al.* 1997; Peleg, Rousso *et al.* 2000). The resulting composite, while useful for artistic guidance and in high-overlap panoramas (*manifold mosaics*) tends to have very hard edges with noticeable seams when the exposures vary (Figure 8.14e).

Xiong and Turkowski (1998) use this Voronoi idea (local maximum of the grassfire transform) to select seams for Laplacian pyramid blending (which is discussed below). However, since the seam selection is performed sequentially as new images are added in, some artifacts can occur.

Optimal seam selection. Computing the Voronoi diagram is one way to select the *seams* between regions where different images contribute to the final composite. However, Voronoi images totally ignore the local image structure underlying the seam. A better approach is to place the seams in regions where the images agree, so that transitions from one source to another are not visible. In this way, the algorithm avoids “cutting through” moving objects where a seam would look unnatural (Davis 1998). For a pair of images, this process can be formulated as a simple dynamic program starting from one edge of the overlap region and ending at the other (Milgram 1975, 1977; Davis 1998; Efros and Freeman 2001).

When multiple images are being composited, the dynamic program idea does not readily generalize. (For square texture tiles being composited sequentially, Efros and Freeman (2001) run a dynamic program along each of the four tile sides.)

To overcome this problem, Uyttendaele, Eden, and Szeliski (2001) observed that, for well-registered images, moving objects produce the most visible artifacts, namely translucent looking *ghosts*. Their system therefore decides which objects to keep and which ones to erase. First, the algorithm compares all overlapping input image pairs to determine *regions of difference* (RODs) where the images disagree. Next, a graph is constructed with the



Figure 8.16 *Photomontage (Agarwala, Dontcheva et al. 2004) © 2004 ACM. From a set of five source images (of which four are shown on the left), Photomontage quickly creates a composite family portrait in which everyone is smiling and looking at the camera (right). Users simply flip through the stack and coarsely draw strokes using the designated source image objective over the people they wish to add to the composite. The user-applied strokes and computed regions (middle) are color-coded by the borders of the source images on the left.*

RODs as vertices and edges representing ROD pairs that overlap in the final composite (Figure 8.15). Since the presence of an edge indicates an area of disagreement, vertices (regions) must be removed from the final composite until no edge spans a pair of remaining vertices. The smallest such set can be computed using a *vertex cover* algorithm. Since several such covers may exist, a *weighted vertex cover* is used instead, where the vertex weights are computed by summing the feather weights in the ROD (Uyttendaele, Eden, and Szeliski 2001). The algorithm therefore prefers removing regions that are near the edge of the image, which reduces the likelihood that partially visible objects will appear in the final composite. (It is also possible to infer which object in a region of difference is the foreground object by the “edginess” (pixel differences) across the ROD boundary, which should be higher when an object is present (Herley 2005).) Once the desired excess regions of difference have been removed, the final composite can be created by feathering (Figure 8.14f).

A different approach to pixel selection and seam placement is described by Agarwala, Dontcheva et al. (2004). Their system computes the label assignment that optimizes the sum of two objective functions. The first is a per-pixel *image objective* that determines which pixels are likely to produce good composites,

$$E_D = \sum_{\mathbf{x}} D(\mathbf{x}, l(\mathbf{x})), \quad (8.72)$$

where $D(\mathbf{x}, l)$ is the *data penalty* associated with choosing image l at pixel $\mathbf{x}$. In their system, users can select which pixels to use by “painting” over an image with the desired object or



Figure 8.17 Set of five photos tracking a snowboarder’s jump stitched together into a seamless composite. Because the algorithm prefers pixels near the center of the image, multiple copies of the boarder are retained.

appearance, which sets $D(\mathbf{x}, l)$ to a large value for all labels l other than the one selected by the user (Figure 8.16). Alternatively, automated selection criteria can be used, such as *maximum likelihood*, which prefers pixels that occur repeatedly in the background (for object removal), or *minimum likelihood* for objects that occur infrequently, i.e., for moving object retention. Using a more traditional center-weighted data term tends to favor objects that are centered in the input images (Figure 8.17).

The second term is a *seam objective* that penalizes differences in labels between adjacent images,

$$E_S = \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{N}} S(\mathbf{x}, \mathbf{y}, l(\mathbf{x}), l(\mathbf{y})), \quad (8.73)$$

where $S(\mathbf{x}, \mathbf{y}, l_x, l_y)$ is the image-dependent *interaction penalty* or *seam cost* of placing a seam between pixels $\mathbf{x}$ and $\mathbf{y}$, and $\mathcal{N}$ is the set of $\mathcal{N}_4$ neighboring pixels. For example, the simple color-based seam penalty used in Kwatra, Schödl *et al.* (2003) and Agarwala, Dontcheva *et al.* (2004) can be written as

$$S(\mathbf{x}, \mathbf{y}, l_x, l_y) = \|\tilde{I}_{l_x}(\mathbf{x}) - \tilde{I}_{l_y}(\mathbf{x})\| + \|\tilde{I}_{l_x}(\mathbf{y}) - \tilde{I}_{l_y}(\mathbf{y})\|. \quad (8.74)$$

More sophisticated seam penalties can also look at image gradients or the presence of image edges (Agarwala, Dontcheva *et al.* 2004). Seam penalties are widely used in other computer vision applications such as stereo matching (Boykov, Veksler, and Zabih 2001) to give the labeling function its *coherence* or *smoothness*. An alternative approach, which places seams along strong consistent edges in overlapping images using a watershed computation is described by Soille (2006).

The sum of these two objective functions gives rise to a *Markov random field* (MRF), for which good optimization algorithms are described in Sections 4.3 and 4.3.2 and Appendix B.5. For label computations of this kind, the α -*expansion* algorithm developed by Boykov, Veksler, and Zabih (2001) works particularly well (Szeliski, Zabih *et al.* 2008).

For the result shown in Figure 8.14g, Agarwala, Dontcheva *et al.* (2004) use a large data penalty for invalid pixels and 0 for valid pixels. Notice how the seam placement algorithm avoids regions of difference, including those that border the image and that might result in objects being cut off. Graph cuts (Agarwala, Dontcheva *et al.* 2004) and vertex cover (Uyttendaele, Eden, and Szeliski 2001) often produce similar looking results, although the former is significantly slower since it optimizes over all pixels, while the latter is more sensitive to the thresholds used to determine regions of difference. More recent approaches to seam selection include SEAGULL (Lin, Jiang *et al.* 2016), which jointly optimizes local alignment and seam selection, and object-centered image stitching (Herrmann, Wang *et al.* 2018a), which uses an off-the-shelf object detector to avoid cutting through objects.

8.4.3 Application: Photomontage

While image stitching is normally used to composite partially overlapping photographs, it can also be used to composite repeated shots of a scene taken with the aim of obtaining the best possible composition and appearance of each element.

Figure 8.16 shows the *Photomontage* system developed by Agarwala, Dontcheva *et al.* (2004), where users draw strokes over a set of pre-aligned images to indicate which regions they wish to keep from each image. Once the system solves the resulting multi-label graph cut (8.72–8.73), the various pieces taken from each source photo are blended together using a variant of Poisson image blending (8.75–8.77). Their system can also be used to automatically composite an all-focus image from a series of bracketed focus images (Hasinoff, Kutulakos *et al.* 2009) or to remove wires and other unwanted elements from sets of photographs. Exercise 8.14 has you implement this system and try out some of its variants.

8.4.4 Blending

Once the seams between images have been determined and unwanted objects removed, we still need to blend the images to compensate for exposure differences and other misalignments. The spatially varying weighting (feathering) previously discussed can often be used to accomplish this. However, it is difficult in practice to achieve a pleasing balance between smoothing out low-frequency exposure variations and retaining sharp enough transitions to prevent blurring (although using a high exponent in feathering can help).

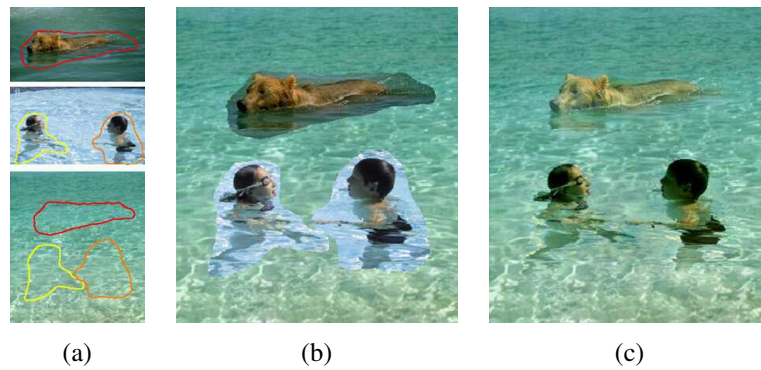


Figure 8.18 *Poisson image editing (Pérez, Gangnet, and Blake 2003) © 2003 ACM: (a) The dog and the two children are chosen as source images to be pasted into the destination swimming pool. (b) Simple pasting fails to match the colors at the boundaries, whereas (c) Poisson image blending masks these differences.*

Laplacian pyramid blending. An attractive solution to this problem is the Laplacian pyramid blending technique developed by Burt and Adelson (1983b), which we discussed in Section 3.5.5. Instead of using a single transition width, a frequency-adaptive width is used by creating a band-pass (Laplacian) pyramid and making the transition widths within each level a function of the level, i.e., the same width in pixels. In practice, a small number of levels, i.e., as few as two (Brown and Lowe 2007), may be adequate to compensate for differences in exposure. The result of applying this pyramid blending is shown in Figure 8.14h.

Gradient domain blending. An alternative approach to multi-band image blending is to perform the operations in the *gradient domain*. Reconstructing images from their gradient fields has a long history in computer vision (Horn 1986), starting originally with work in brightness constancy (Horn 1974), shape from shading (Horn and Brooks 1989), and photometric stereo (Woodham 1981). Related ideas have also been used for reconstructing images from their edges (Elder and Goldberg 2001), removing shadows from images (Weiss 2001), separating reflections from a single image (Levin, Zomet, and Weiss 2004; Levin and Weiss 2007), and *tone mapping* high dynamic range images by reducing the magnitude of image edges (gradients) (Fattal, Lischinski, and Werman 2002).

Pérez, Gangnet, and Blake (2003) show how gradient domain reconstruction can be used to do seamless object insertion in image editing applications (Figure 8.18). Rather than copying pixels, the *gradients* of the new image fragment are copied instead. The actual pixel values for the copied area are then computed by solving a *Poisson equation* that locally matches the

gradients while obeying the fixed *Dirichlet* (exact matching) conditions at the seam boundary. Pérez, Gangnet, and Blake (2003) show that this is equivalent to computing an additive *membrane* interpolant of the mismatch between the source and destination images along the boundary.²⁵ In earlier work, Peleg (1981) also proposed adding a smooth function to enforce consistency along the seam curve.

Agarwala, Dontcheva *et al.* (2004) extended this idea to a multi-source formulation, where it no longer makes sense to talk of a destination image whose exact pixel values must be matched at the seam. Instead, *each* source image contributes its own gradient field and the Poisson equation is solved using *Neumann* boundary conditions, i.e., dropping any equations that involve pixels outside the boundary of the image.

Rather than solving the Poisson partial differential equations, Agarwala, Dontcheva *et al.* (2004) directly minimize a *variational problem*,

$$\min_{C(\mathbf{x})} \|\nabla C(\mathbf{x}) - \nabla \tilde{I}_{l(\mathbf{x})}(\mathbf{x})\|^2. \quad (8.75)$$

The discretized form of this equation is a set of gradient constraint equations

$$C(\mathbf{x} + \hat{i}) - C(\mathbf{x}) = \tilde{I}_{l(\mathbf{x})}(\mathbf{x} + \hat{i}) - \tilde{I}_{l(\mathbf{x})}(\mathbf{x}) \quad \text{and} \quad (8.76)$$

$$C(\mathbf{x} + \hat{j}) - C(\mathbf{x}) = \tilde{I}_{l(\mathbf{x})}(\mathbf{x} + \hat{j}) - \tilde{I}_{l(\mathbf{x})}(\mathbf{x}), \quad (8.77)$$

where $\hat{i} = (1, 0)$ and $\hat{j} = (0, 1)$ are unit vectors in the x and y directions.²⁶ They then solve the associated sparse least squares problem. Since this system of equations is only defined up to an additive constraint, Agarwala, Dontcheva *et al.* (2004) ask the user to select the value of one pixel. In practice, a better choice might be to weakly bias the solution towards reproducing the original color values.

In order to accelerate the solution of this sparse linear system, Fattal, Lischinski, and Werman (2002) use multigrid, whereas Agarwala, Dontcheva *et al.* (2004) use hierarchical basis preconditioned conjugate gradient descent (Szeliski 1990b, 2006b; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013) (Appendix A.5). In subsequent work, Agarwala (2007) shows how using a quadtree representation for the solution can further accelerate the computation with minimal loss in accuracy, while Szeliski, Uyttendaele, and Steedly (2008) show how representing the per-image offset fields using coarser splines is even faster. This latter work also argues that blending in the log domain, i.e., using multiplicative rather than additive offsets, is preferable, as it more closely matches texture contrasts across seam boundaries. The resulting seam blending works very well in practice (Figure 8.14h), although care

²⁵The membrane interpolant is known to have nicer interpolation properties for arbitrary-shaped constraints than frequency-domain interpolants (Nielson 1993).

²⁶At seam locations, the right-hand side is replaced by the average of the gradients in the two source images.

must be taken when copying large gradient values near seams so that a “double edge” is not introduced.

Copying gradients directly from the source images after seam placement is just one approach to gradient domain blending. The paper by [Levin, Zomet *et al.* \(2004\)](#) examines several different variants of this approach, which they call *Gradient-domain Image STitching* (GIST). The techniques they examine include feathering (blending) the gradients from the source images, as well as using an L_1 norm in performing the reconstruction of the image from the gradient field, rather than using an L_2 norm as in Equation (8.75). Their preferred technique is the L_1 optimization of a feathered (blended) cost function on the original image gradients (which they call GIST1- l_1). Since L_1 optimization using linear programming can be slow, they develop a faster iterative median-based algorithm in a multigrid framework. Visual comparisons between their preferred approach and what they call *optimal seam on the gradients* (which is equivalent to the approach of [Agarwala, Dontcheva *et al.* \(2004\)](#)) show similar results, while significantly improving on pyramid blending and feathering algorithms.

Exposure compensation. Pyramid and gradient domain blending can do a good job of compensating for moderate amounts of exposure differences between images. However, when the exposure differences become large, alternative approaches may be necessary.

[Uyttendaele, Eden, and Szeliski \(2001\)](#) iteratively estimate a local correction between each source image and a blended composite. First, a block-based quadratic transfer function is fit between each source image and an initial feathered composite. Next, transfer functions are averaged with their neighbors to get a smoother mapping and per-pixel transfer functions are computed by *splining* (interpolating) between neighboring block values. Once each source image has been smoothly adjusted, a new feathered composite is computed and the process is repeated (typically three times). The results shown by [Uyttendaele, Eden, and Szeliski \(2001\)](#) demonstrate that this does a better job of exposure compensation than simple feathering and can handle local variations in exposure due to effects such as lens vignetting.

Ultimately, however, the most principled way to deal with exposure differences is to stitch images in the radiance domain, i.e., to convert each image into a radiance image using its exposure value and then create a stitched, high dynamic range image, as discussed in Section 10.2 and [Eden, Uyttendaele, and Szeliski \(2006\)](#).

8.5 Additional reading

[Hartley and Zisserman \(2004\)](#) provide a wonderful introduction to the topics of feature-based alignment and optimal motion estimation. Techniques for robust estimation are discussed

in more detail in Appendix B.3 and in monographs and review articles on this topic (Huber 1981; Hampel, Ronchetti *et al.* 1986; Rousseeuw and Leroy 1987; Black and Rangarajan 1996; Stewart 1999). The most commonly used robust initialization technique in computer vision is RANdom SAMple Consensus (RANSAC) (Fischler and Bolles 1981), which has spawned a series of more efficient variants (Torr and Zisserman 2000; Nistér 2003; Chum and Matas 2005; Raguram, Chum *et al.* 2012; Brachmann, Krull *et al.* 2017; Barath and Matas 2018; Barath, Matas, and Noskova 2019; Brachmann and Rother 2019). The MAGSAC++ paper by Barath, Noskova *et al.* (2020) compares many of these variants.

The literature on image stitching dates back to work in the photogrammetry community in the 1970s (Milgram 1975, 1977; Slama 1980). In computer vision, papers started appearing in the early 1980s (Peleg 1981), while the development of fully automated techniques came about a decade later (Mann and Picard 1994; Chen 1995; Szeliski 1996; Szeliski and Shum 1997; Sawhney and Kumar 1999; Shum and Szeliski 2000). Those techniques used direct pixel-based alignment but feature-based approaches are now the norm (Zoghlami, Faugeras, and Deriche 1997; Capel and Zisserman 1998; Cham and Cipolla 1998; Badra, Qumsieh, and Dudek 1998; McLauchlan and Jaenicke 2002; Brown and Lowe 2007). A collection of some of these papers can be found in the book by Benosman and Kang (2001). Szeliski (2006a) provides a comprehensive survey of image stitching, on which the material in this chapter is based. More recent publications include Zaragoza, Chin *et al.* (2013), Zhang and Liu (2014), Lin, Pankanti *et al.* (2015), Lin, Jiang *et al.* (2016), Herrmann, Wang *et al.* (2018b), Lee and Sim (2020), and Zhuang and Tran (2020).

High-quality techniques for optimal seam finding and blending are another important component of image stitching systems. Important developments in this field include work by Milgram (1977), Burt and Adelson (1983b), Davis (1998), Uyttendaele, Eden, and Szeliski (2001), Pérez, Gangnet, and Blake (2003), Levin, Zomet *et al.* (2004), Agarwala, Dontcheva *et al.* (2004), Eden, Uyttendaele, and Szeliski (2006), Kopf, Uyttendaele *et al.* (2007), Lin, Jiang *et al.* (2016), and Herrmann, Wang *et al.* (2018a).

In addition to the merging of multiple overlapping photographs taken for aerial or terrestrial panoramic image creation, stitching techniques can be used for automated white-board scanning (He and Zhang 2005; Zhang and He 2007), scanning with a mouse (Nakao, Kashitani, and Kaneyoshi 1998), and retinal image mosaics (Can, Stewart *et al.* 2002). They can also be applied to video sequences (Teodosio and Bender 1993; Irani, Hsu, and Anandan 1995; Kumar, Anandan *et al.* 1995; Sawhney and Ayer 1996; Massey and Bender 1996; Irani and Anandan 1998; Sawhney, Arpa *et al.* 2002; Agarwala, Zheng *et al.* 2005; Rav-Acha, Pritch *et al.* 2005; Steedly, Pal, and Szeliski 2005; Baudisch, Tan *et al.* 2006) and can even be used for video compression (Lee, Chen *et al.* 1997).

8.6 Exercises

Ex 8.1: Feature-based image alignment for flip-book animations. Take a set of photos of an action scene or portrait (preferably in burst shooting mode) and align them to make a composite or flip-book animation.

1. Extract features and feature descriptors using some of the techniques described in Sections 7.1.1–7.1.2.
2. Match your features using nearest neighbor matching with a nearest neighbor distance ratio test (7.18).
3. Compute an optimal 2D translation and rotation between the first image and all subsequent images, using least squares (Section 8.1.1) with optional RANSAC for robustness (Section 8.1.4).
4. Resample all of the images onto the first image’s coordinate frame (Section 3.6.1) using either bilinear or bicubic resampling and optionally crop them to their common area.
5. Convert the resulting images into an animated GIF (using software available from the web) or optionally implement cross-dissolves to turn them into a “slo-mo” video.
6. (Optional) Combine this technique with feature-based (Exercise 3.25) morphing.

Ex 8.2: Panography. Create the kind of panograph discussed in Section 8.1.2 and commonly found on the web.

1. Take a series of interesting overlapping photos.
2. Use the feature detector, descriptor, and matcher developed in Exercises 7.1–7.4 (or existing software) to match features among the images.
3. Turn each connected component of matching features into a *track*, i.e., assign a unique index i to each track, discarding any tracks that are inconsistent (contain two different features in the same image).
4. Compute a global translation for each image using Equation (8.12).
5. Since your matches probably contain errors, turn the above least square metric into a robust metric (8.25) and re-solve your system using iteratively reweighted least squares.
6. Compute the size of the resulting composite canvas and resample each image into its final position on the canvas. (Keeping track of bounding boxes will make this more efficient.)

7. Average all of the images, or choose some kind of ordering and implement translucent *over* compositing (3.8).
8. (Optional) Extend your parametric motion model to include rotations and scale, i.e., the similarity transform given in Table 8.1. Discuss how you could handle the case of translations and rotations only (no scale).
9. (Optional) Write a simple tool to let the user adjust the ordering and opacity, and add or remove images.
10. (Optional) Write down a different least squares problem that involves pairwise matching of images. Discuss why this might be better or worse than the global matching formula given in (8.12).

Ex 8.3: 2D rigid/Euclidean matching. Several alternative approaches are given in Section 8.1.3 for estimating a 2D rigid (Euclidean) alignment.

1. Implement the various alternatives and compare their accuracy on synthetic data, i.e., random 2D point clouds with noisy feature positions.
2. One approach is to estimate the translations from the centroids and then estimate rotation in polar coordinates. Do you need to weight the angles obtained from a polar decomposition in some way to get the statistically correct estimate?
3. How can you modify your techniques to take into account either scalar (8.10) or full two-dimensional point covariance weightings (8.11)? Do all of the previously developed “shortcuts” still work or does full weighting require iterative optimization?

Ex 8.4: 2D match move/augmented reality. Replace a picture in a magazine or a book with a different image or video.

1. Take a picture of a magazine or book page.
2. Outline a figure or picture on the page with a rectangle, i.e., draw over the four sides as they appear in the image.
3. Match features in this area with each new image frame.
4. Replace the original image with an “advertising” insert, warping the new image with the appropriate homography.
5. Try your approach on a clip from a sporting event (e.g., indoor or outdoor soccer) to implement a billboard replacement.

Ex 8.5: Direct pixel-based alignment. Take a pair of images, compute a coarse-to-fine affine alignment (Exercise 9.2) and then blend them using either averaging (Exercise 8.2) or a Laplacian pyramid (Exercise 3.18). Extend your motion model from affine to perspective (homography) to better deal with rotational mosaics and planar surfaces seen under arbitrary motion.

Ex 8.6: Featured-based stitching. Extend your feature-based alignment technique from Exercise 8.2 to use a full perspective model and then blend the resulting mosaic using either averaging or more sophisticated distance-based feathering (Exercise 8.13).

Ex 8.7: Cylindrical strip panoramas. To generate cylindrical or spherical panoramas from a horizontally panning (rotating) camera, it is best to use a tripod. Set your camera up to take a series of 50% overlapped photos and then use the following steps to create your panorama:

1. Estimate the amount of radial distortion by taking some pictures with lots of long straight lines near the edges of the image and then using the plumb-line method from Exercise 11.5.
2. Compute the focal length either by using a ruler and paper (Debevec, Wenger *et al.* 2002) or by rotating your camera on the tripod, overlapping the images by exactly 0% and counting the number of images it takes to make a 360° panorama.
3. Convert each of your images to cylindrical coordinates using (8.45–8.49).
4. Line up the images with a translational motion model using either a direct pixel-based technique, such as coarse-to-fine incremental or an FFT, or a feature-based technique.
5. (Optional) If doing a complete 360° panorama, align the first and last images. Compute the amount of accumulated vertical misregistration and re-distribute this among the images.
6. Blend the resulting images using feathering or some other technique.

Ex 8.8: Coarse alignment. Use FFT or phase correlation (Section 9.1.2) to estimate the initial alignment between successive images. How well does this work? Over what range of overlaps? If it does not work, does aligning sub-sections (e.g., quarters) do better?

Ex 8.9: Automated mosaicing. Use feature-based alignment with four-point RANSAC for homographies (Section 8.1.3, Equations (8.19–8.23)) or three-point RANSAC for rotational motions (Brown, Hartley, and Nistér 2007) to match up all pairs of overlapping images.

Merge these pairwise estimates together by finding a spanning tree of pairwise relations. Visualize the resulting global alignment, e.g., by displaying a blend of each image with all other images that overlap it.

For greater robustness, try multiple spanning trees (perhaps randomly sampled based on the confidence in pairwise alignments) to see if you can recover from bad pairwise matches (Zach, Klopschitz, and Pollefeys 2010). As a measure of fitness, count how many pairwise estimates are consistent with the global alignment.

Ex 8.10: Global optimization. Use the initialization from the previous algorithm to perform a full bundle adjustment over all of the camera rotations and focal lengths, as described in Section 11.4.2 and by Shum and Szeliski (2000). Optionally, estimate radial distortion parameters as well or support fisheye lenses (Section 2.1.5).

As in the previous exercise, visualize the quality of your registration by creating composites of each input image with its neighbors, optionally blinking between the original image and the composite to better see misalignment artifacts.

Ex 8.11: Deghosting. Use the results of the previous bundle adjustment to predict the location of each feature in a consensus geometry. Use the difference between the predicted and actual feature locations to correct for small misregistrations, as described in Section 8.3.2 (Shum and Szeliski 2000).

Ex 8.12: Compositing surface. Choose a compositing surface (Section 8.4.1), e.g., a single reference image extended to a larger plane, a sphere represented using cylindrical or spherical coordinates, a stereographic “little planet” projection, or a cube map.

Project all of your images onto this surface and blend them with equal weighting, for now (just to see where the original image seams are).

Ex 8.13: Feathering and blending. Compute a feather (distance) map for each warped source image and use these maps to blend the warped images.

Alternatively, use Laplacian pyramid blending (Exercise 3.18) or gradient domain blending.

Ex 8.14: Photomontage and object removal. Implement a “Photomontage” system in which users can indicate desired or unwanted regions in pre-registered images using strokes or other primitives (such as bounding boxes).

(Optional) Devise an automatic moving objects remover (or “keeper”) by analyzing which inconsistent regions are more or less typical given some consensus (e.g., median filtering) of the aligned images. Figure 8.17 shows an example where the moving object was kept. Try

to make this work for sequences with large amounts of overlaps and consider averaging the images to make the moving object look more ghosted.

Chapter 9

Motion estimation

9.1	Translational alignment	558
9.1.1	Hierarchical motion estimation	562
9.1.2	Fourier-based alignment	563
9.1.3	Incremental refinement	566
9.2	Parametric motion	570
9.2.1	<i>Application:</i> Video stabilization	573
9.2.2	Spline-based motion	575
9.2.3	<i>Application:</i> Medical image registration	577
9.3	Optical flow	578
9.3.1	Deep learning approaches	584
9.3.2	<i>Application:</i> Rolling shutter wobble removal	587
9.3.3	Multi-frame motion estimation	587
9.3.4	<i>Application:</i> Video denoising	589
9.4	Layered motion	589
9.4.1	<i>Application:</i> Frame interpolation	593
9.4.2	Transparent layers and reflections	594
9.4.3	Video object segmentation	597
9.4.4	Video object tracking	598
9.5	Additional reading	600
9.6	Exercises	602

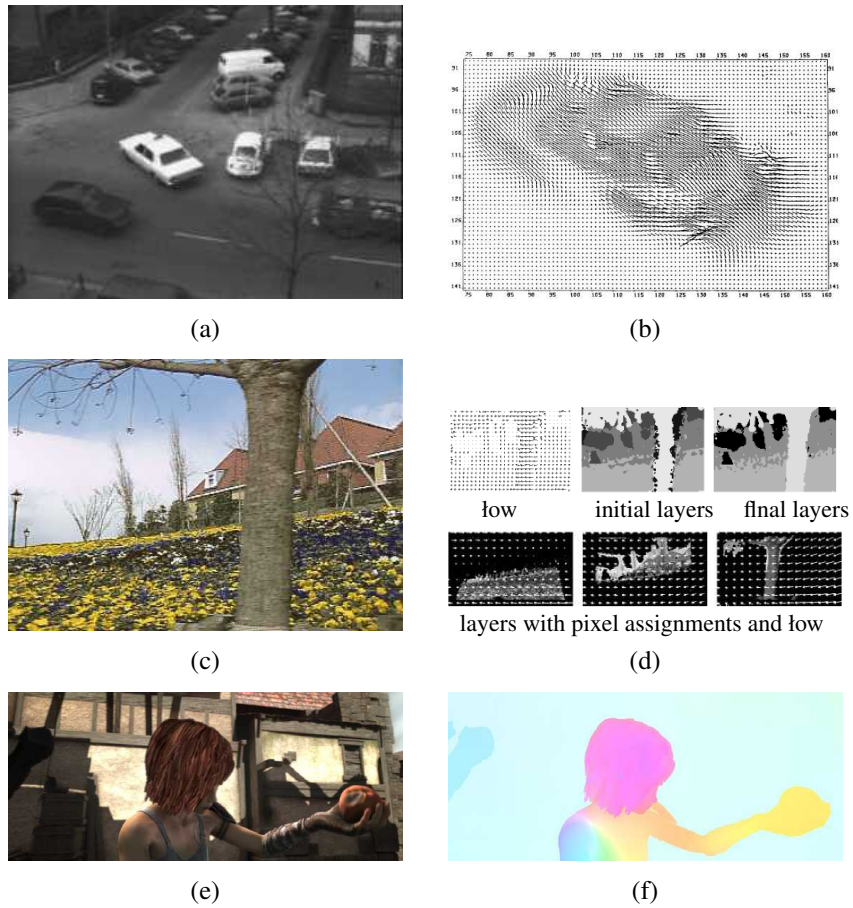


Figure 9.1 Motion estimation: (a–b) regularization-based optical flow (Nagel and Enkelmann 1986) © 1986 IEEE; (c–d) layered motion estimation (Wang and Adelson 1994) © 1994 IEEE; (e–f) sample image and ground truth flow from evaluation database (Butler, Wulff et al. 2012) © 2012 Springer.

Algorithms for aligning images and estimating motion in video sequences are among the most widely used in computer vision. For example, frame-rate image alignment is widely used in digital cameras to implement their image stabilization (IS) feature.

An early example of a widely used image registration algorithm is the patch-based translational alignment (optical flow) technique developed by Lucas and Kanade (1981). Variants of this algorithm are used in almost all motion-compensated video compression schemes such as MPEG/H.263 (Le Gall 1991) and HEVC/H.265 (Sullivan, Ohm *et al.* 2012). Similar parametric motion estimation algorithms have found a wide variety of applications, including video summarization (Teodosio and Bender 1993; Irani and Anandan 1998), video stabilization (Hansen, Anandan *et al.* 1994; Srinivasan, Chellappa *et al.* 2005; Matsushita, Ofek *et al.* 2006), and video compression (Irani, Hsu, and Anandan 1995; Lee, Chen *et al.* 1997). More sophisticated image registration algorithms have also been developed for medical imaging and remote sensing. Image registration techniques are surveyed by Brown (1992), Zitov'aa and Flusser (2003), Goshtasby (2005), and Szeliski (2006a).

To estimate the motion between two or more images, a suitable *error metric* must first be chosen to compare the images (Section 9.1). Once this has been established, a suitable *search* technique must be devised. The simplest technique is to exhaustively try all possible alignments, i.e., to do a *full search*. In practice, this may be too slow, so *hierarchical* coarse-to-fine techniques (Section 9.1.1) based on image pyramids are normally used. Alternatively, Fourier transforms (Section 9.1.2) can be used to speed up the computation.

To get sub-pixel precision in the alignment, *incremental* methods (Section 9.1.3) based on a Taylor series expansion of the image function are often used. These can also be applied to *parametric motion models* (Section 9.2), which model global image transformations such as rotation or shearing. Motion estimation can be made more reliable by *learning* the typical dynamics or motion statistics of the scenes or objects being tracked, e.g., the natural gait of walking people (Section 9.2). For more complex motions, piecewise parametric *spline motion models* (Section 9.2.2) can be used.

In the presence of multiple independent (and perhaps non-rigid) motions, general-purpose *optical flow* (or *optic flow*) techniques need to be used, as described in Section 9.3. In recent years, the best-performing techniques have started using deep neural networks (Section 9.3.1). For even more complex motions that include a lot of occlusions, *layered motion models* (Section 9.4), which decompose the scene into coherently moving layers, can work well. Such representations can also be used to perform video object segmentation (Section 9.4.3) and object tracking (Section 9.4.4).

In this chapter, we describe each of these techniques in more detail. Additional details can be found in review and comparative evaluation papers on motion estimation (Barron,

Fleet, and Beauchemin 1994; Mitiche and Bouthemy 1996; Stiller and Konrad 1999; Szeliski 2006a; Baker, Scharstein *et al.* 2011; Sun, Yang *et al.* 2018; Janai, Güney *et al.* 2020; Hur and Roth 2020).

9.1 Translational alignment

The simplest way to establish an alignment between two images or image patches is to shift one image relative to the other. Given a *template* image $I_0(\mathbf{x})$ sampled at discrete pixel locations $\{\mathbf{x}_i = (x_i, y_i)\}$, we wish to find where it is located in image $I_1(\mathbf{x})$. A least squares solution to this problem is to find the minimum of the *sum of squared differences* (SSD) function

$$E_{\text{SSD}}(\mathbf{u}) = \sum_i [I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)]^2 = \sum_i e_i^2, \quad (9.1)$$

where $\mathbf{u} = (u, v)$ is the *displacement* and $e_i = I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)$ is called the *residual error* (or the *displaced frame difference* in the video coding literature).¹ (We ignore for the moment the possibility that parts of I_0 may lie outside the boundaries of I_1 or be otherwise not visible.) The assumption that corresponding pixel values remain the same in the two images is often called the *brightness constancy constraint*.²

In general, the displacement $\mathbf{u}$ can be fractional, so a suitable interpolation function must be applied to image $I_1(\mathbf{x})$. In practice, a bilinear interpolant is often used, but bicubic interpolation can yield slightly better results (Szeliski and Scharstein 2004). Color images can be processed by summing differences across all three color channels, although it is also possible to first transform the images into a different color space or to only use the luminance (which is often done in video encoders).

Robust error metrics. We can make the above error metric more robust to outliers by replacing the squared error terms with a robust function $\rho(e_i)$ (Huber 1981; Hampel, Ronchetti *et al.* 1986; Black and Anandan 1996; Stewart 1999) to obtain

$$E_{\text{SRD}}(\mathbf{u}) = \sum_i \rho(I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)) = \sum_i \rho(e_i). \quad (9.2)$$

The robust norm $\rho(e)$ is a function that grows less quickly than the quadratic penalty associated with least squares. One such function, sometimes used in motion estimation for video

¹The usual justification for using least squares is that it is the optimal estimate with respect to Gaussian noise. See the discussion below on robust error metrics as well as Appendix B.3.

²Brightness constancy (Horn 1974) is the tendency for objects to maintain their perceived brightness under varying illumination conditions.

coding because of its speed, is the *sum of absolute differences* (SAD) metric³ or L_1 norm, i.e.,

$$E_{\text{SAD}}(\mathbf{u}) = \sum_i |I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)| = \sum_i |e_i|. \quad (9.3)$$

However, because this function is not differentiable at the origin, it is not well suited to gradient-descent approaches such as the ones presented in Section 9.1.3.

Instead, a smoothly varying function that is quadratic for small values but grows more slowly away from the origin is often used. Black and Rangarajan (1996) discuss a variety of such functions, including the *Geman–McClure* function,

$$\rho_{\text{GM}}(x) = \frac{x^2}{1 + x^2/a^2}, \quad (9.4)$$

where a is a constant that can be thought of as an *outlier threshold*. An appropriate value for the threshold can itself be derived using robust statistics (Huber 1981; Hampel, Ronchetti *et al.* 1986; Rousseeuw and Leroy 1987), e.g., by computing the *median absolute deviation*, $MAD = \text{med}_i |e_i|$, and multiplying it by 1.4 to obtain a robust estimate of the standard deviation of the inlier noise process (Stewart 1999). Barron (2019) proposes a generalized robust loss function that can model various outlier distributions and thresholds, as discussed in more detail in Sections 4.1.3 and Appendix B.3, and also has a Bayesian method for estimating the loss function parameters.

Spatially varying weights. The error metrics above ignore that fact that for a given alignment, some of the pixels being compared may lie outside the original image boundaries. Furthermore, we may want to partially or completely downweight the contributions of certain pixels. For example, we may want to selectively “erase” some parts of an image from consideration when stitching a mosaic where unwanted foreground objects have been cut out. For applications such as background stabilization, we may want to downweight the middle part of the image, which often contains independently moving objects being tracked by the camera.

All of these tasks can be accomplished by associating a spatially varying per-pixel weight with each of the two images being matched. The error metric then becomes the weighted (or *windowed*) SSD function,

$$E_{\text{WSSD}}(\mathbf{u}) = \sum_i w_0(\mathbf{x}_i) w_1(\mathbf{x}_i + \mathbf{u}) [I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)]^2, \quad (9.5)$$

³In video compression, e.g., the H.264 standard (<https://www.itu.int/rec/T-REC-H.264>), the sum of absolute transformed differences (SATD), which measures the differences in a frequency transform space, e.g., using a Hadamard transform, is often used, as it more accurately predicts quality (Richardson 2003).

where the weighting functions w_0 and w_1 are zero outside the image boundaries.

If a large range of potential motions is allowed, the above metric can have a bias towards smaller overlap solutions. To counteract this bias, the windowed SSD score can be divided by the overlap area

$$A = \sum_i w_0(\mathbf{x}_i) w_1(\mathbf{x}_i + \mathbf{u}) \quad (9.6)$$

to compute a *per-pixel* (or mean) squared pixel error E_{WSSD}/A . The square root of this quantity is the *root mean square* intensity error

$$RMS = \sqrt{E_{\text{WSSD}}/A} \quad (9.7)$$

often reported in comparative studies.

Bias and gain (exposure differences). Often, the two images being aligned were not taken with the same exposure. A simple model of linear (affine) intensity variation between the two images is the *bias and gain* model,

$$I_1(\mathbf{x} + \mathbf{u}) = (1 + \alpha)I_0(\mathbf{x}) + \beta, \quad (9.8)$$

where β is the *bias* and α is the *gain* (Lucas and Kanade 1981; Gennert 1988; Fuh and Maragos 1991; Baker, Gross, and Matthews 2003; Evangelidis and Psarakis 2008). The least squares formulation then becomes

$$E_{\text{BG}}(\mathbf{u}) = \sum_i [I_1(\mathbf{x}_i + \mathbf{u}) - (1 + \alpha)I_0(\mathbf{x}_i) - \beta]^2 = \sum_i [\alpha I_0(\mathbf{x}_i) + \beta - e_i]^2. \quad (9.9)$$

Rather than taking a simple squared difference between corresponding patches, it becomes necessary to perform a *linear regression* (Appendix A.2), which is somewhat more costly. Note that for color images, it may be necessary to estimate a different bias and gain for each color channel to compensate for the automatic *color correction* performed by some digital cameras (Section 2.3.2). Bias and gain compensation are also used in video codecs, where they are known as *weighted prediction* (Richardson 2003).

A more general (spatially varying, non-parametric) model of intensity variation, which is computed as part of the registration process, is used in Negahdaripour (1998), Jia and Tang (2003), and Seitz and Baker (2009). This can be useful for dealing with local variations such as the *vignetting* caused by wide-angle lenses, wide apertures, or lens housings. It is also possible to pre-process the images before comparing their values, e.g., using band-pass filtered images (Anandan 1989; Bergen, Anandan *et al.* 1992), or gradients (Scharstein 1994; Papenberg, Bruhn *et al.* 2006), or using other local transformations such as histograms or rank

transforms (Cox, Roy, and Hingorani 1995; Zabih and Woodfill 1994), or to maximize *mutual information* (Viola and Wells III 1997; Kim, Kolmogorov, and Zabih 2003). Hirschmüller and Scharstein (2009) compare a number of these approaches and report on their relative performance in scenes with exposure differences.

Correlation. An alternative to taking intensity differences is to perform *correlation*, i.e., to maximize the *product* (or *cross-correlation*) of the two aligned images,

$$E_{CC}(\mathbf{u}) = \sum_i I_0(\mathbf{x}_i) I_1(\mathbf{x}_i + \mathbf{u}). \quad (9.10)$$

At first glance, this may appear to make bias and gain modeling unnecessary, since the images will prefer to line up regardless of their relative scales and offsets. However, this is actually not true. If a very bright patch exists in $I_1(\mathbf{x})$, the maximum product may actually lie in that area.

For this reason, *normalized cross-correlation* is more commonly used,

$$E_{NCC}(\mathbf{u}) = \frac{\sum_i [I_0(\mathbf{x}_i) - \bar{I}_0] [I_1(\mathbf{x}_i + \mathbf{u}) - \bar{I}_1]}{\sqrt{\sum_i [I_0(\mathbf{x}_i) - \bar{I}_0]^2} \sqrt{\sum_i [I_1(\mathbf{x}_i + \mathbf{u}) - \bar{I}_1]^2}}, \quad (9.11)$$

where

$$\bar{I}_0 = \frac{1}{N} \sum_i I_0(\mathbf{x}_i) \quad \text{and} \quad (9.12)$$

$$\bar{I}_1 = \frac{1}{N} \sum_i I_1(\mathbf{x}_i + \mathbf{u}) \quad (9.13)$$

are the *mean images* of the corresponding patches and N is the number of pixels in the patch. The normalized cross-correlation score is always guaranteed to be in the range $[-1, 1]$, which makes it easier to handle in some higher-level applications, such as deciding which patches truly match. Normalized correlation works well when matching images taken with different exposures, e.g., when creating high dynamic range images (Section 10.2). Note, however, that the NCC score is undefined if either of the two patches has zero variance (and, in fact, its performance degrades for noisy low-contrast regions).

A variant on NCC, which is related to the bias–gain regression implicit in the matching score (9.9), is the *normalized SSD* score

$$E_{NSSD}(\mathbf{u}) = \frac{1}{2} \frac{\sum_i [I_0(\mathbf{x}_i) - \bar{I}_0] - [I_1(\mathbf{x}_i + \mathbf{u}) - \bar{I}_1]^2}{\sqrt{\sum_i [I_0(\mathbf{x}_i) - \bar{I}_0]^2 + [I_1(\mathbf{x}_i + \mathbf{u}) - \bar{I}_1]^2}} \quad (9.14)$$

proposed by Criminisi, Sotton *et al.* (2007). In their experiments, they find that it produces comparable results to NCC, but is more efficient when applied to a large number of overlapping patches using a moving average technique (Section 3.2.2).

9.1.1 Hierarchical motion estimation

Now that we have a well-defined alignment cost function to optimize, how can we find its minimum? The simplest solution is to do a *full search* over some range of shifts, using either integer or sub-pixel steps. This is often the approach used for *block matching* in *motion compensated video compression*, where a range of possible motions (say, ± 16 pixels) is explored.⁴

To accelerate this search process, *hierarchical motion estimation* is often used: an image pyramid (Section 3.5) is constructed and a search over a smaller number of discrete pixels (corresponding to the same range of motion) is first performed at coarser levels (Quam 1984; Anandan 1989; Bergen, Anandan *et al.* 1992). The motion estimate from one level of the pyramid is then used to initialize a smaller *local* search at the next finer level. Alternatively, several seeds (good solutions) from the coarse level can be used to initialize the fine-level search. While this is not guaranteed to produce the same result as a full search, it usually works almost as well and is much faster.

More formally, let

$$I_k^{(l)}(\mathbf{x}_j) \leftarrow \tilde{I}_k^{(l-1)}(2\mathbf{x}_j) \quad (9.15)$$

be the *decimated* image at level l obtained by subsampling (*downsampling*) a smoothed version of the image at level $l-1$. See Section 3.5 for how to perform the required downsampling (pyramid construction) without introducing too much aliasing.

At the coarsest level, we search for the best displacement $\mathbf{u}^{(l)}$ that minimizes the difference between images $I_0^{(l)}$ and $I_1^{(l)}$. This is usually done using a full search over some range of displacements $\mathbf{u}^{(l)} \in 2^{-l}[-S, S]^2$, where S is the desired *search range* at the finest (original) resolution level, optionally followed by the incremental refinement step described in Section 9.1.3.

Once a suitable motion vector has been estimated, it is used to *predict* a likely displacement

$$\hat{\mathbf{u}}^{(l-1)} \leftarrow 2\mathbf{u}^{(l)} \quad (9.16)$$

⁴In stereo matching (Section 12.1.2), an explicit search over all possible disparities (i.e., a *plane sweep*) is almost always performed, as the number of search hypotheses is much smaller due to the 1D nature of the potential displacements.

for the next finer level.⁵ The search over displacements is then repeated at the finer level over a much narrower range of displacements, say $\hat{\mathbf{u}}^{(l-1)} \pm 1$, again optionally combined with an incremental refinement step (Anandan 1989). Alternatively, one of the images can be *warped* (resampled) by the current motion estimate, in which case only small incremental motions need to be computed at the finer level. A nice description of the whole process, extended to parametric motion estimation (Section 9.2), is provided by Bergen, Anandan *et al.* (1992).

9.1.2 Fourier-based alignment

When the search range corresponds to a significant fraction of the larger image (as is the case in image stitching, see Section 8.2), the hierarchical approach may not work that well, as it is often not possible to coarsen the representation too much before significant features are blurred away. In this case, a Fourier-based approach may be preferable.

Fourier-based alignment relies on the fact that the Fourier transform of a shifted signal has the same magnitude as the original signal, but a linearly varying phase (Section 3.4), i.e.,

$$\mathcal{F}\{I_1(\mathbf{x} + \mathbf{u})\} = \mathcal{F}\{I_1(\mathbf{x})\} e^{-j\mathbf{u} \cdot \boldsymbol{\omega}} = \mathcal{I}_1(\boldsymbol{\omega}) e^{-j\mathbf{u} \cdot \boldsymbol{\omega}}, \quad (9.17)$$

where $\boldsymbol{\omega}$ is the vector-valued angular frequency of the Fourier transform and we use calligraphic notation $\mathcal{I}_1(\boldsymbol{\omega}) = \mathcal{F}\{I_1(\mathbf{x})\}$ to denote the Fourier transform of a signal (Section 3.4).

Another useful property of Fourier transforms is that convolution in the spatial domain corresponds to multiplication in the Fourier domain (Section 3.4).⁶ The Fourier transform of the cross-correlation function E_{CC} can thus be written as

$$\mathcal{F}\{E_{CC}(\mathbf{u})\} = \mathcal{F}\left\{\sum_i I_0(\mathbf{x}_i) I_1(\mathbf{x}_i + \mathbf{u})\right\} = \mathcal{F}\{I_0(\mathbf{u}) \bar{*} I_1(\mathbf{u})\} = \mathcal{I}_0(\boldsymbol{\omega}) \mathcal{I}_1^*(\boldsymbol{\omega}), \quad (9.18)$$

where

$$f(\mathbf{u}) \bar{*} g(\mathbf{u}) = \sum_i f(\mathbf{x}_i) g(\mathbf{x}_i + \mathbf{u}) \quad (9.19)$$

is the *correlation* function, i.e., the convolution of one signal with the reverse of the other, and $\mathcal{I}_1^*(\boldsymbol{\omega})$ is the *complex conjugate* of $\mathcal{I}_1(\boldsymbol{\omega})$. This is because convolution is defined as the summation of one signal with the reverse of the other (Section 3.4).

⁵This doubling of displacements is only necessary if displacements are defined in integer *pixel* coordinates, which is the usual case in the literature (Bergen, Anandan *et al.* 1992). If *normalized device coordinates* (Section 2.1.4) are used instead, the displacements (and search ranges) need not change from level to level, although the step sizes will need to be adjusted, to keep search steps of roughly one pixel or less.

⁶In fact, the Fourier shift property (9.17) derives from the convolution theorem by observing that shifting is equivalent to convolution with a displaced delta function $\delta(\mathbf{x} - \mathbf{u})$.

To efficiently evaluate E_{CC} over the range of all possible values of $\mathbf{u}$, we take the Fourier transforms of both images $I_0(\mathbf{x})$ and $I_1(\mathbf{x})$, multiply both transforms together (after conjugating the second one), and take the inverse transform of the result. The Fast Fourier Transform algorithm can compute the transform of an $N \times M$ image in $O(NM \log NM)$ operations (Bracewell 1986). This can be significantly faster than the $O(N^2 M^2)$ operations required to do a full search when the full range of image overlaps is considered.

While Fourier-based convolution is often used to accelerate the computation of image correlations, it can also be used to accelerate the sum of squared differences function (and its variants). Consider the SSD formula given in (9.1). Its Fourier transform can be written as

$$\begin{aligned} \mathcal{F}\{E_{SSD}(\mathbf{u})\} &= \mathcal{F}\left\{\sum_i [I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)]^2\right\} \\ &= \delta(\boldsymbol{\omega}) \sum_i [I_0^2(\mathbf{x}_i) + I_1^2(\mathbf{x}_i)] - 2\mathcal{I}_0(\boldsymbol{\omega})\mathcal{I}_1^*(\boldsymbol{\omega}). \end{aligned} \quad (9.20)$$

Thus, the SSD function can be computed by taking twice the correlation function and subtracting it from the sum of the energies in the two images (or patches).

Windowed correlation. Unfortunately, the Fourier convolution theorem only applies when the summation over $\mathbf{x}_i$ is performed over *all* the pixels in both images, using a circular shift of the image when accessing pixels outside the original boundaries. While this is acceptable for small shifts and comparably sized images, it makes no sense when the images overlap by a small amount or one image is a small subset of the other.

In that case, the cross-correlation function should be replaced with a *windowed* (weighted) cross-correlation function,

$$E_{WCC}(\mathbf{u}) = \sum_i w_0(\mathbf{x}_i)I_0(\mathbf{x}_i) w_1(\mathbf{x}_i + \mathbf{u})I_1(\mathbf{x}_i + \mathbf{u}), \quad (9.21)$$

$$= [w_0(\mathbf{x})I_0(\mathbf{x})] \star [w_1(\mathbf{x})I_1(\mathbf{x})] \quad (9.22)$$

where the weighting functions w_0 and w_1 are zero outside the valid ranges of the images and both images are padded so that circular shifts return 0 values outside the original image boundaries.

An even more interesting case is the computation of the *weighted* SSD function introduced in Equation (9.5),

$$E_{WSSD}(\mathbf{u}) = \sum_i w_0(\mathbf{x}_i)w_1(\mathbf{x}_i + \mathbf{u})[I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i)]^2. \quad (9.23)$$

Expanding this as a sum of correlations and deriving the appropriate set of Fourier transforms is left for Exercise 9.1.

The same kind of derivation can also be applied to the bias–gain corrected sum of squared difference function E_{BG} (9.9). Again, Fourier transforms can be used to efficiently compute all the correlations needed to perform the linear regression in the bias and gain parameters in order to estimate the exposure-compensated difference for each potential shift (Exercise 9.1). It is also possible to use Fourier transforms to estimate the rotation and scale between two patches that are centered on the same pixel, as described in De Castro and Morandi (1987) and Szeliski (2010, Section 8.1.2).

Phase correlation. A variant of regular correlation (9.18) that is sometimes used for motion estimation is *phase correlation* (Kuglin and Hines 1975; Brown 1992). Here, the spectrum of the two signals being matched is *whitened* by dividing each per-frequency product in (9.18) by the magnitudes of the Fourier transforms,

$$\mathcal{F}\{E_{\text{PC}}(\mathbf{u})\} = \frac{\mathcal{I}_0(\boldsymbol{\omega})\mathcal{I}_1^*(\boldsymbol{\omega})}{\|\mathcal{I}_0(\boldsymbol{\omega})\|\|\mathcal{I}_1(\boldsymbol{\omega})\|} \quad (9.24)$$

before taking the final inverse Fourier transform. In the case of noiseless signals with perfect (cyclic) shift, we have $I_1(\mathbf{x} + \mathbf{u}) = I_0(\mathbf{x})$ and hence, from Equation (9.17), we obtain

$$\begin{aligned} \mathcal{F}\{I_1(\mathbf{x} + \mathbf{u})\} &= \mathcal{I}_1(\boldsymbol{\omega})e^{-2\pi j\mathbf{u}\cdot\boldsymbol{\omega}} = \mathcal{I}_0(\boldsymbol{\omega}) \quad \text{and} \\ \mathcal{F}\{E_{\text{PC}}(\mathbf{u})\} &= e^{-2\pi j\mathbf{u}\cdot\boldsymbol{\omega}}. \end{aligned} \quad (9.25)$$

The output of phase correlation (under ideal conditions) is therefore a single spike (impulse) located at the correct value of $\mathbf{u}$, which (in principle) makes it easier to find the correct estimate.

Phase correlation has a reputation in some quarters of outperforming regular correlation, but this behavior depends on the characteristics of the signals and noise. If the original images are contaminated by noise in a narrow frequency band (e.g., low-frequency noise or peaked frequency “hum”), the whitening process effectively de-emphasizes the noise in these regions. However, if the original signals have very low signal-to-noise ratio at some frequencies (say, two blurry or low-textured images with lots of high-frequency noise), the whitening process can actually decrease performance (see Exercise 9.1).

Gradient cross-correlation has emerged as a promising alternative to phase correlation (Argyriou and Vlachos 2003), although further systematic studies are probably warranted. Phase correlation has also been studied by Fleet and Jepson (1990) as a method for estimating general optical flow and stereo disparity.

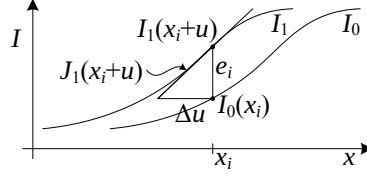


Figure 9.2 Taylor series approximation of a function and the incremental computation of the optical flow correction amount. $\mathbf{J}_1(\mathbf{x}_i + \mathbf{u})$ is the image gradient at $(\mathbf{x}_i + \mathbf{u})$ and e_i is the current intensity difference.

9.1.3 Incremental refinement

The techniques described up till now can estimate alignment to the nearest pixel (or potentially fractional pixel if smaller search steps are used). In general, image stabilization and stitching applications require much higher accuracies to obtain acceptable results.

To obtain better *sub-pixel* estimates, we can use one of several techniques described by Tian and Huhns (1986). One possibility is to evaluate several discrete (integer or fractional) values of (u, v) around the best value found so far and to *interpolate* the matching score to find an analytic minimum (Szeliski and Scharstein 2004).

A more commonly used approach, first proposed by Lucas and Kanade (1981), is to perform *gradient descent* on the SSD energy function (9.1), using a Taylor series expansion of the image function (Figure 9.2),

$$E_{\text{LK-SSD}}(\mathbf{u} + \Delta\mathbf{u}) = \sum_i [I_1(\mathbf{x}_i + \mathbf{u} + \Delta\mathbf{u}) - I_0(\mathbf{x}_i)]^2 \quad (9.26)$$

$$\approx \sum_i [I_1(\mathbf{x}_i + \mathbf{u}) + \mathbf{J}_1(\mathbf{x}_i + \mathbf{u})\Delta\mathbf{u} - I_0(\mathbf{x}_i)]^2 \quad (9.27)$$

$$= \sum_i [\mathbf{J}_1(\mathbf{x}_i + \mathbf{u})\Delta\mathbf{u} + e_i]^2, \quad (9.28)$$

where

$$\mathbf{J}_1(\mathbf{x}_i + \mathbf{u}) = \nabla I_1(\mathbf{x}_i + \mathbf{u}) = \left(\frac{\partial I_1}{\partial x}, \frac{\partial I_1}{\partial y} \right) (\mathbf{x}_i + \mathbf{u}) \quad (9.29)$$

is the *image gradient* or *Jacobian* at $(\mathbf{x}_i + \mathbf{u})$ and

$$e_i = I_1(\mathbf{x}_i + \mathbf{u}) - I_0(\mathbf{x}_i), \quad (9.30)$$

first introduced in (9.1), is the current intensity error.⁷ The gradient at a particular sub-pixel location $(\mathbf{x}_i + \mathbf{u})$ can be computed using a variety of techniques, the simplest of which is

⁷We follow the convention, commonly used in robotics and by Baker and Matthews (2004), that derivatives with respect to (column) vectors result in row vectors, so that fewer transposes are needed in the formulas.

simply to take the horizontal and vertical differences between pixels $\mathbf{x}$ and $\mathbf{x} + (1, 0)$ or $\mathbf{x} + (0, 1)$. More sophisticated derivatives can sometimes lead to noticeable performance improvements.

The linearized form of the incremental update to the SSD error (9.28) is often called the *optical flow constraint* or *brightness constancy constraint* equation (Horn and Schunck 1981)

$$I_x u + I_y v + I_t = 0, \quad (9.31)$$

where the subscripts in I_x and I_y denote spatial derivatives, and I_t is called the *temporal derivative*, which makes sense if we are computing instantaneous velocity in a video sequence. When squared and summed or integrated over a region, it can be used to compute optical flow (Horn and Schunck 1981).

The above least squares problem (9.28) can be minimized by solving the associated *normal equations* (Appendix A.2),

$$\mathbf{A} \Delta \mathbf{u} = \mathbf{b} \quad (9.32)$$

where

$$\mathbf{A} = \sum_i \mathbf{J}_1^T(\mathbf{x}_i + \mathbf{u}) \mathbf{J}_1(\mathbf{x}_i + \mathbf{u}) \quad (9.33)$$

and

$$\mathbf{b} = - \sum_i e_i \mathbf{J}_1^T(\mathbf{x}_i + \mathbf{u}) \quad (9.34)$$

are called the (Gauss–Newton approximation of the) *Hessian* and *gradient-weighted residual vector*, respectively.⁸ These matrices are also often written as

$$\mathbf{A} = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \quad \text{and} \quad \mathbf{b} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}. \quad (9.35)$$

The gradients required for $\mathbf{J}_1(\mathbf{x}_i + \mathbf{u})$ can be evaluated at the same time as the image warps required to estimate $I_1(\mathbf{x}_i + \mathbf{u})$ (Section 3.6.1 (3.75)) and, in fact, are often computed as a side-product of image interpolation. If efficiency is a concern, these gradients can be replaced by the gradients in the *template* image,

$$\mathbf{J}_1(\mathbf{x}_i + \mathbf{u}) \approx \mathbf{J}_0(\mathbf{x}_i), \quad (9.36)$$

because near the correct alignment, the template and displaced target images should look similar. This has the advantage of allowing the precomputation of the Hessian and Jacobian

⁸The true Hessian is the full second derivative of the error function E , which may not be positive definite—see Section 8.1.3 and Appendix A.3.

images, which can result in significant computational savings (Hager and Belhumeur 1998; Baker and Matthews 2004). A further reduction in computation can be obtained by writing the warped image $I_1(\mathbf{x}_i + \mathbf{u})$ used to compute e_i in (9.30) as a convolution of a sub-pixel interpolation filter with the discrete samples in I_1 (Peleg and Rav-Acha 2006). Precomputing the inner product between the gradient field and shifted version of I_1 allows the iterative re-computation of e_i to be performed in constant time (independent of the number of pixels).

The effectiveness of the above incremental update rule relies on the quality of the Taylor series approximation. When far away from the true displacement (say, 1–2 pixels), several iterations may be needed. It is possible, however, to estimate a value for $\mathbf{J}_1$ using a least squares fit to a series of larger displacements to increase the range of convergence (Jurie and Dhome 2002) or to “learn” a special-purpose recognizer for a given patch (Avidan 2001; Williams, Blake, and Cipolla 2003; Lepetit, Pilet, and Fua 2006; Hinterstoisser, Benhimane *et al.* 2008; Özuysal, Calonder *et al.* 2010) as discussed in Section 7.1.5.

A commonly used stopping criterion for incremental updating is to monitor the magnitude of the displacement correction $\|\mathbf{u}\|$ and to stop when it drops below a certain threshold (say, $1/10$ of a pixel). For larger motions, it is usual to combine the incremental update rule with a hierarchical coarse-to-fine search strategy, as described in Section 9.1.1.

Conditioning and aperture problems. Sometimes, the inversion of the linear system (9.32) can be poorly conditioned because of lack of two-dimensional texture in the patch being aligned. A commonly occurring example of this is the *aperture problem*, first identified in some of the early papers on optical flow (Horn and Schunck 1981) and then studied more extensively by Anandan (1989). Consider an image patch that consists of a slanted edge moving to the right (Figure 7.4). Only the *normal* component of the velocity (displacement) can be reliably recovered in this case. This manifests itself in (9.32) as a *rank-deficient* matrix $\mathbf{A}$, i.e., one whose smaller eigenvalue is very close to zero.⁹

When Equation (9.32) is solved, the component of the displacement along the edge is very poorly conditioned and can result in wild guesses under small noise perturbations. One way to mitigate this problem is to add a *prior* (soft constraint) on the expected range of motions (Simoncelli, Adelson, and Heeger 1991; Baker, Gross, and Matthews 2004; Govindu 2006). This can be accomplished by adding a small value to the diagonal of $\mathbf{A}$, which essentially biases the solution towards smaller $\Delta\mathbf{u}$ values that still (mostly) minimize the squared error.

However, the pure Gaussian model assumed when using a simple (fixed) quadratic prior, as in Simoncelli, Adelson, and Heeger (1991), does not always hold in practice, e.g., because

⁹The matrix $\mathbf{A}$ is by construction always guaranteed to be symmetric positive semi-definite, i.e., it has real non-negative eigenvalues.

of aliasing along strong edges (Triggs 2004). For this reason, it may be prudent to add some small fraction (say, 5%) of the larger eigenvalue to the smaller one before doing the matrix inversion.

Uncertainty modeling. The reliability of a particular patch-based motion estimate can be captured more formally with an *uncertainty model*. The simplest such model is a *covariance matrix*, which captures the expected variance in the motion estimate in all possible directions. As discussed in Section 8.1.4 and Appendix B.6, under small amounts of additive Gaussian noise, it can be shown that the covariance matrix $\Sigma_{\mathbf{u}}$ is proportional to the inverse of the Hessian $\mathbf{A}$,

$$\Sigma_{\mathbf{u}} = \sigma_n^2 \mathbf{A}^{-1}, \quad (9.37)$$

where σ_n^2 is the variance of the additive Gaussian noise (Anandan 1989; Matthies, Kanade, and Szeliski 1989; Szeliski 1989).

For larger amounts of noise, the linearization performed by the Lucas–Kanade algorithm in (9.28) is only approximate, so the above quantity becomes a *Cramer–Rao lower bound* on the true covariance. Thus, the minimum and maximum eigenvalues of the Hessian $\mathbf{A}$ can now be interpreted as the (scaled) inverse variances in the least-certain and most-certain directions of motion. (A more detailed analysis using a more realistic model of image noise is given by Steele and Jaynes (2005).) Figure 7.5 shows the local SSD surfaces for three different pixel locations in an image. As you can see, the surface has a clear minimum in the highly textured region and suffers from the aperture problem near the strong edge.

Bias and gain, weighting, and robust error metrics. The Lucas–Kanade update rule can also be applied to the bias–gain equation (9.9) to obtain

$$E_{\text{LK-BG}}(\mathbf{u} + \Delta\mathbf{u}) = \sum_i [\mathbf{J}_1(\mathbf{x}_i + \mathbf{u})\Delta\mathbf{u} + e_i - \alpha I_0(\mathbf{x}_i) - \beta]^2 \quad (9.38)$$

(Lucas and Kanade 1981; Gennert 1988; Fuh and Maragos 1991; Baker, Gross, and Matthews 2003). The resulting 4×4 system of equations can be solved to simultaneously estimate the translational displacement update $\Delta\mathbf{u}$ and the bias and gain parameters β and α .

A similar formulation can be derived for images (templates) that have a *linear appearance variation*,

$$I_1(\mathbf{x} + \mathbf{u}) \approx I_0(\mathbf{x}) + \sum_j \lambda_j B_j(\mathbf{x}), \quad (9.39)$$

where the $B_j(\mathbf{x})$ are the *basis images* and the λ_j are the unknown coefficients (Hager and Belhumeur 1998; Baker, Gross *et al.* 2003; Baker, Gross, and Matthews 2003). Potential

linear appearance variations include illumination changes (Hager and Belhumeur 1998) and small non-rigid deformations (Black and Jepson 1998; Kambhamettu, Goldgof *et al.* 2003).

A weighted (windowed) version of the Lucas–Kanade algorithm is also possible:

$$E_{\text{LK-WSSD}}(\mathbf{u} + \Delta\mathbf{u}) = \sum_i w_0(\mathbf{x}_i) w_1(\mathbf{x}_i + \mathbf{u}) [\mathbf{J}_1(\mathbf{x}_i + \mathbf{u}) \Delta\mathbf{u} + e_i]^2. \quad (9.40)$$

Note that here, in deriving the Lucas–Kanade update from the original weighted SSD function (9.5), we have neglected taking the derivative of the $w_1(\mathbf{x}_i + \mathbf{u})$ weighting function with respect to $\mathbf{u}$, which is usually acceptable in practice, especially if the weighting function is a binary mask with relatively few transitions.

Baker, Gross *et al.* (2003) only use the $w_0(\mathbf{x})$ term, which is reasonable if the two images have the same extent and no (independent) cutouts in the overlap region. They also discuss the idea of making the weighting proportional to $\nabla I(\mathbf{x})$, which helps for very noisy images, where the gradient itself is noisy. Similar observations, formulated in terms of *total least squares* (Van Huffel and Vandewalle 1991; Van Huffel and Lemmerling 2002), have been made by other researchers studying optical flow (Weber and Malik 1995; Bab-Hadiashar and Suter 1998b; Mühlich and Mester 1998). Baker, Gross *et al.* (2003) show how evaluating Equation (9.40) at just the *most reliable* (highest gradient) pixels does not significantly reduce performance for large enough images, even if only 5–10% of the pixels are used. (This idea was originally proposed by Dellaert and Collins (1999), who used a more sophisticated selection criterion.)

The Lucas–Kanade incremental refinement step can also be applied to the robust error metric introduced in Section 9.1,

$$E_{\text{LK-SRD}}(\mathbf{u} + \Delta\mathbf{u}) = \sum_i \rho(\mathbf{J}_1(\mathbf{x}_i + \mathbf{u}) \Delta\mathbf{u} + e_i), \quad (9.41)$$

which can be solved using the *iteratively reweighted least squares* technique described in Section 8.1.4.

9.2 Parametric motion

Many image alignment tasks, for example image stitching with handheld cameras, require the use of more sophisticated motion models, as described in Section 2.1.1. As these models, e.g., affine deformations, typically have more parameters than pure translation, a full search over the possible range of values is impractical. Instead, the incremental Lucas–Kanade algorithm can be generalized to parametric motion models and used in conjunction with a hierarchical search algorithm (Lucas and Kanade 1981; Rehg and Witkin 1991; Fuh and Maragos 1991;

Bergen, Anandan *et al.* 1992; Shashua and Toelg 1997; Shashua and Wexler 2001; Baker and Matthews 2004).

For parametric motion, instead of using a single constant translation vector $\mathbf{u}$, we use a spatially varying *motion field* or *correspondence map*, $\mathbf{x}'(\mathbf{x}; \mathbf{p})$, parameterized by a low-dimensional vector $\mathbf{p}$, where $\mathbf{x}'$ can be any of the motion models presented in Section 2.1.1. The parametric incremental motion update rule now becomes

$$E_{\text{LK-PM}}(\mathbf{p} + \Delta\mathbf{p}) = \sum_i [I_1(\mathbf{x}'(\mathbf{x}_i; \mathbf{p} + \Delta\mathbf{p})) - I_0(\mathbf{x}_i)]^2 \quad (9.42)$$

$$\approx \sum_i [I_1(\mathbf{x}'_i) + \mathbf{J}_1(\mathbf{x}'_i)\Delta\mathbf{p} - I_0(\mathbf{x}_i)]^2 \quad (9.43)$$

$$= \sum_i [\mathbf{J}_1(\mathbf{x}'_i)\Delta\mathbf{p} + e_i]^2, \quad (9.44)$$

where the Jacobian is now

$$\mathbf{J}_1(\mathbf{x}'_i) = \frac{\partial I_1}{\partial \mathbf{p}} = \nabla I_1(\mathbf{x}'_i) \frac{\partial \mathbf{x}'}{\partial \mathbf{p}}(\mathbf{x}_i), \quad (9.45)$$

i.e., the product of the image gradient ∇I_1 with the Jacobian of the correspondence field, $\mathbf{J}_{x'} = \partial \mathbf{x}' / \partial \mathbf{p}$.

The motion Jacobians $\mathbf{J}_{x'}$ for the 2D planar transformations introduced in Section 2.1.1 and Table 2.1 are given in Table 8.1. Note how we have re-parameterized the motion matrices so that they are always the identity at the origin $\mathbf{p} = 0$. This becomes useful later, when we talk about the compositional and inverse compositional algorithms. (It also makes it easier to impose priors on the motions.)

For parametric motion, the (Gauss–Newton) *Hessian* and *gradient-weighted residual vector* become

$$\mathbf{A} = \sum_i \mathbf{J}_{x'}^T(\mathbf{x}_i) [\nabla I_1^T(\mathbf{x}'_i) \nabla I_1(\mathbf{x}'_i)] \mathbf{J}_{x'}(\mathbf{x}_i) \quad (9.46)$$

and

$$\mathbf{b} = - \sum_i \mathbf{J}_{x'}^T(\mathbf{x}_i) [e_i \nabla I_1^T(\mathbf{x}'_i)]. \quad (9.47)$$

Note how the expressions inside the square brackets are the same ones evaluated for the simpler translational motion case (9.33–9.34).

Patch-based approximation. The computation of the Hessian and residual vectors for parametric motion can be significantly more expensive than for the translational case. For parametric motion with n parameters and N pixels, the accumulation of $\mathbf{A}$ and $\mathbf{b}$ takes $O(n^2 N)$ operations (Baker and Matthews 2004). One way to reduce this by a significant

amount is to divide the image up into smaller sub-blocks (patches) P_j and to only accumulate the simpler 2×2 quantities inside the square brackets at the pixel level (Shum and Szeliski 2000),

$$\mathbf{A}_j = \sum_{i \in P_j} \nabla I_1^T(\mathbf{x}'_i) \nabla I_1(\mathbf{x}'_i) \quad (9.48)$$

$$\mathbf{b}_j = \sum_{i \in P_j} e_i \nabla I_1^T(\mathbf{x}'_i). \quad (9.49)$$

The full Hessian and residual can then be approximated as

$$\mathbf{A} \approx \sum_j \mathbf{J}_{\mathbf{x}'}^T(\hat{\mathbf{x}}_j) \left[\sum_{i \in P_j} \nabla I_1^T(\mathbf{x}'_i) \nabla I_1(\mathbf{x}'_i) \right] \mathbf{J}_{\mathbf{x}'}(\hat{\mathbf{x}}_j) = \sum_j \mathbf{J}_{\mathbf{x}'}^T(\hat{\mathbf{x}}_j) \mathbf{A}_j \mathbf{J}_{\mathbf{x}'}(\hat{\mathbf{x}}_j) \quad (9.50)$$

and

$$\mathbf{b} \approx - \sum_j \mathbf{J}_{\mathbf{x}'}^T(\hat{\mathbf{x}}_j) \left[\sum_{i \in P_j} e_i \nabla I_1^T(\mathbf{x}'_i) \right] = - \sum_j \mathbf{J}_{\mathbf{x}'}^T(\hat{\mathbf{x}}_j) \mathbf{b}_j, \quad (9.51)$$

where $\hat{\mathbf{x}}_j$ is the *center* of each patch P_j (Shum and Szeliski 2000). This is equivalent to replacing the true motion Jacobian with a piecewise-constant approximation. In practice, this works quite well.

Compositional approach. For a complex parametric motion such as a homography, the computation of the motion Jacobian becomes complicated and may involve a per-pixel division. Szeliski and Shum (1997) observed that this can be simplified by first warping the target image I_1 according to the current motion estimate $\mathbf{x}'(\mathbf{x}; \mathbf{p})$,

$$\tilde{I}_1(\mathbf{x}) = I_1(\mathbf{x}'(\mathbf{x}; \mathbf{p})), \quad (9.52)$$

and then comparing this *warped* image against the template $I_0(\mathbf{x})$. Subsequently Hager and Belhumeur (1998) suggested replacing the gradient of $\tilde{I}_1(\mathbf{x})$ with the gradient of $I_0(\mathbf{x})$, as described previously in (9.36), which allows the precomputation (and inversion) of the Hessian matrix $\mathbf{A}$ given in (9.46). The residual vector $\mathbf{b}$ (9.47) can also be partially pre-computed, i.e., the *steepest descent* images $\nabla I_0(\mathbf{x}) \mathbf{J}_{\tilde{\mathbf{x}}}(\mathbf{x})$ can be precomputed and stored for later multiplication with the $e(\mathbf{x}) = \tilde{I}_1(\mathbf{x}) - I_0(\mathbf{x})$ error images, as described in (Szeliski 2010, Section 8.2) and (Baker and Matthews 2004), where this is called the *inverse additive* scheme. Baker and Matthews (2004) also introduce one more variant they call the *inverse compositional* algorithm where they warp the template image $I_0(\mathbf{x})$ and precompute the inverse Hessian and the steepest descent images, which makes it the preferred approach. They also discuss the advantage of using Gauss–Newton iteration (i.e., the first-order expansion

of the least squares, as above) compared to other approaches such as steepest descent and Levenberg–Marquardt.

Subsequent parts of the series (Baker, Gross *et al.* 2003; Baker, Gross, and Matthews 2003, 2004) discuss more advanced topics such as per-pixel weighting, pixel selection for efficiency, a more in-depth discussion of robust metrics and algorithms, linear appearance variations, and priors on parameters. They make for invaluable reading for anyone interested in implementing a highly tuned implementation of incremental image registration and have been widely used as components of subsequent object trackers, which are discussed in Section 9.4.4. Evangelidis and Psarakis (2008) provide some detailed experimental evaluations of these and other related approaches.

Learned motion models

An alternative to parameterizing the motion field with a geometric deformation such as an affine transform is to learn a set of basis functions tailored to a particular application (Black, Yacoob *et al.* 1997). First, a set of dense motion fields (Section 9.3) is computed from a set of training videos. Next, singular value decomposition (SVD) is applied to the stack of motion fields $\mathbf{u}_t(\mathbf{x})$ to compute the first few singular vectors $\mathbf{v}_k(\mathbf{x})$. Finally, for a new test sequence, a novel low field is computed using a coarse-to-fine algorithm that estimates the unknown coefficient a_k in the parameterized low field

$$\mathbf{u}(\mathbf{x}) = \sum_k a_k \mathbf{v}_k(\mathbf{x}). \quad (9.53)$$

Figure 9.3a shows a set of basis fields learned by observing videos of walking motions. Figure 9.3b shows the temporal evolution of the basis coefficients as well as a few of the recovered parametric motion fields. Note that similar ideas can also be applied to feature tracks (Torresani, Hertzmann, and Bregler 2008), which is a topic we discuss in more detail in Sections 7.1.5 and 13.6.4, as well as video stabilization (Yu and Ramamoorthi 2020).

9.2.1 Application: Video stabilization

Video stabilization is one of the most widely used applications of parametric motion estimation (Hansen, Anandan *et al.* 1994; Irani, Rousso, and Peleg 1997; Morimoto and Chellappa 1997; Srinivasan, Chellappa *et al.* 2005; Grundmann, Kwatra, and Essa 2011). Algorithms for stabilization run inside both hardware devices, such as camcorders and still cameras, and software packages for improving the visual quality of shaky videos.

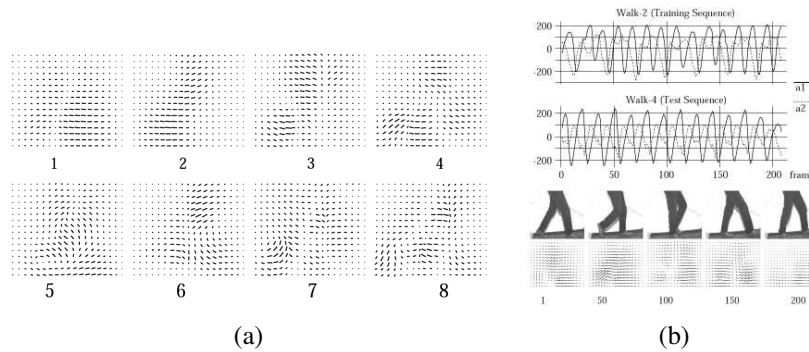


Figure 9.3 *Learned parameterized motion fields for a walking sequence (Black, Yacoob et al. 1997) © 1997 IEEE: (a) learned basis flow fields; (b) plots of motion coefficients over time and corresponding estimated motion fields.*

In their paper on full-frame video stabilization, Matsushita, Ofek *et al.* (2006) give a nice overview of the three major stages of stabilization, namely motion estimation, motion smoothing, and image warping. Motion estimation algorithms often use a similarity transform to handle camera translations, rotations, and zooming. The tricky part is getting these algorithms to lock onto the background motion, which is a result of the camera movement, without getting distracted by independently moving foreground objects (Yu and Ramamoorthi 2018, 2020; Yu, Ramamoorthi *et al.* 2021). Motion smoothing algorithms recover the low-frequency (slowly varying) part of the motion and then estimate the high-frequency shake component that needs to be removed. While quadratic penalties on motion derivatives are commonly used, more realistic virtual camera motions (locked and linear) can be obtained using L_1 minimization of derivatives (Grundmann, Kwatra, and Essa 2011). Finally, image warping algorithms apply the high-frequency correction to render the original frames as if the camera had undergone only the smooth motion.

The resulting stabilization algorithms can greatly improve the appearance of shaky videos but they often still contain visual artifacts. For example, image warping can result in missing borders around the image, which must be cropped, filled using information from other frames, or hallucinated using inpainting techniques (Section 10.5.1). Furthermore, video frames captured during fast motion are often blurry. Their appearance can be improved either by using deblurring techniques (Section 10.3) or by stealing sharper pixels from other frames with less motion or better focus (Matsushita, Ofek *et al.* 2006). Exercise 9.3 has you implement and test some of these ideas.

In situations where the camera is translating a lot in 3D, e.g., when the videographer is

walking, an even better approach is to compute a full structure from motion reconstruction of the camera motion and 3D scene. One or more smooth 3D camera paths can then be computed and the original video re-rendered using view interpolation with the interpolated 3D point cloud serving as the proxy geometry while preserving salient features in what is sometimes called *content preserving warps* (Liu, Gleicher *et al.* 2009, 2011; Liu, Yuan *et al.* 2013; Kopf, Cohen, and Szeliski 2014). If you have access to a camera array instead of a single video camera, you can do even better using a light field rendering approach (Section 14.3) (Smith, Zhang *et al.* 2009).

9.2.2 Spline-based motion

While parametric motion models are useful in a wide variety of applications (such as video stabilization and mapping onto planar surfaces), most image motion is too complicated to be captured by such low-dimensional models.

Traditionally, optical flow algorithms (Section 9.3) compute an independent motion estimate for each pixel, i.e., the number of flow vectors computed is equal to the number of input pixels. The general optical flow analog to Equation (9.1) can thus be written as

$$E_{\text{SSD-OF}}(\{\mathbf{u}_i\}) = \sum_i [I_1(\mathbf{x}_i + \mathbf{u}_i) - I_0(\mathbf{x}_i)]^2. \quad (9.54)$$

Notice how in the above equation, the number of variables $\{\mathbf{u}_i\}$ is twice the number of measurements, so the problem is underconstrained.

The two classic approaches to this problem, which we study in Section 9.3, are to perform the summation over overlapping regions (the *patch-based* or *window-based* approach) or to add smoothness terms on the $\{\mathbf{u}_i\}$ field using *regularization* or *Markov random fields* (Chapter 4). In this section, we describe an alternative approach that lies somewhere between general optical flow (independent flow at each pixel) and parametric flow (a small number of global parameters). The approach is to represent the motion field as a two-dimensional *spline* controlled by a smaller number of *control vertices* $\{\hat{\mathbf{u}}_j\}$ (Figure 9.4),

$$\mathbf{u}_i = \sum_j \hat{\mathbf{u}}_j B_j(\mathbf{x}_i) = \sum_j \hat{\mathbf{u}}_j w_{i,j}, \quad (9.55)$$

where the $B_j(\mathbf{x}_i)$ are called the *basis functions* and are only non-zero over a small *finite support* interval (Szeliski and Coughlan 1997). We call the $w_{i,j} = B_j(\mathbf{x}_i)$ *weights* to emphasize that the $\{\mathbf{u}_i\}$ are known linear combinations of the $\{\hat{\mathbf{u}}_j\}$.

Substituting the formula for the individual per-pixel flow vectors $\mathbf{u}_i$ (9.55) into the SSD error metric (9.54) yields a parametric motion formula similar to Equation (9.43). The biggest

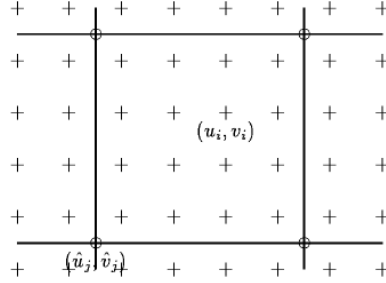


Figure 9.4 *Spline motion field: the displacement vectors $\mathbf{u}_i = (u_i, v_i)$ are shown as pluses (+) and are controlled by the smaller number of control vertices $\hat{\mathbf{u}}_j = (\hat{u}_j, \hat{v}_j)$, which are shown as circles (o).*

difference is that the Jacobian $\mathbf{J}_1(\mathbf{x}'_i)$ (9.45) now consists of the sparse entries in the weight matrix $\mathbf{W} = [w_{ij}]$.

In situations where we know something more about the motion field, e.g., when the motion is due to a camera moving in a static scene, we can use more specialized motion models. For example, the *plane plus parallax* model (Section 2.1.4) can be naturally combined with a spline-based motion representation, where the in-plane motion is represented by a homography (8.19) and the out-of-plane parallax d is represented by a scalar variable at each spline control point (Szeliski and Kang 1995; Szeliski and Coughlan 1997).

In many cases, the small number of spline vertices results in a motion estimation problem that is well conditioned. However, if large textureless regions (or elongated edges subject to the aperture problem) persist across several spline patches, it may be necessary to add a *regularization* term to make the problem well posed (Section 4.2). The simplest way to do this is to directly add squared difference penalties between adjacent vertices in the spline control mesh $\{\hat{\mathbf{u}}_j\}$, as in (4.24). If a multi-resolution (coarse-to-fine) strategy is being used, it is important to re-scale these smoothness terms while going from level to level.

The linear system corresponding to the spline-based motion estimator is sparse and regular. Because it is usually of moderate size, it can often be solved using direct techniques such as Cholesky decomposition (Appendix A.4). Alternatively, if the problem becomes too large and subject to excessive fill-in, iterative techniques such as hierarchically preconditioned conjugate gradient (Szeliski 1990b, 2006b; Krishnan and Szeliski 2011; Krishnan, Fattal, and Szeliski 2013) can be used instead (Appendix A.5).

Because of its robustness, spline-based motion estimation has been used for a number of applications, including visual effects (Roble 1999) and medical image registration (Sec-

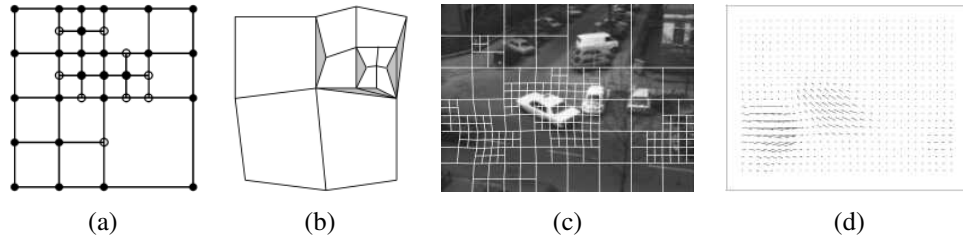


Figure 9.5 Quadtree spline-based motion estimation (Szeliski and Shum 1996) © 1996 IEEE: (a) quadtree spline representation, (b) which can lead to cracks, unless the white nodes are constrained to depend on their parents; (c) deformed quadtree spline mesh overlaid on grayscale image; (d) flow field visualized as a needle diagram.

tion 9.2.3) (Szeliski and Lavallée 1996; Kybic and Unser 2003).

One disadvantage of the basic technique, however, is that the model does a poor job near motion discontinuities, unless an excessive number of nodes are used. To remedy this situation, Szeliski and Shum (1996) propose using a *quadtree* representation embedded in the spline control grid (Figure 9.5a). Large cells are used to present regions of smooth motion, while smaller cells are added in regions of motion discontinuities (Figure 9.5c).

To estimate the motion, a coarse-to-fine strategy is used. Starting with a regular spline imposed over a lower-resolution image, an initial motion estimate is obtained. Spline patches where the motion is inconsistent, i.e., the squared residual (9.54) is above a threshold, are subdivided into smaller patches. To avoid *cracks* in the resulting motion field (Figure 9.5b), the values of certain nodes in the refined mesh, i.e., those adjacent to larger cells, need to be *restricted* so that they depend on their parent values. This is most easily accomplished using a hierarchical basis representation for the quadtree spline (Szeliski 1990b) and selectively setting some of the hierarchical basis functions to 0, as described in (Szeliski and Shum 1996).

9.2.3 Application: Medical image registration

Because they excel at representing smooth *elastic* deformation fields, spline-based motion models have found widespread use in medical image registration (Bajcsy and Kovacic 1989; Szeliski and Lavallée 1996; Christensen, Joshi, and Miller 1997).¹⁰ Registration techniques can be used both to track an individual patient's development or progress over time (a *longitudinal* study) or to match different patient images together to find commonalities and de-

¹⁰In computer graphics, such elastic volumetric deformations are known as *free-form deformations* (Sederberg and Parry 1986; Coquillart 1990; Celniker and Gossard 1991).

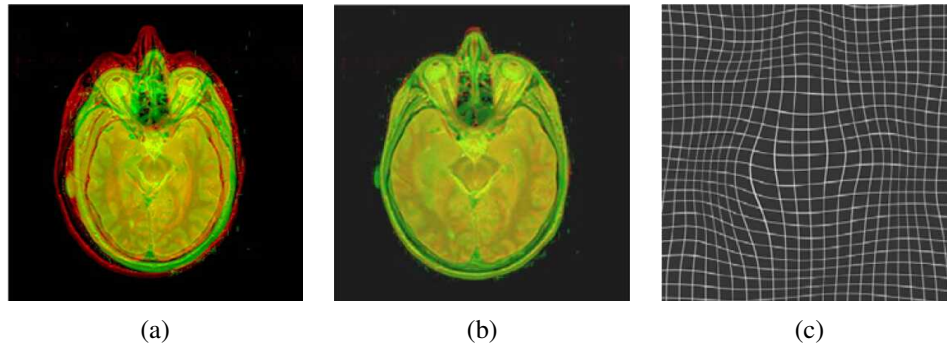


Figure 9.6 Elastic brain registration (Kybic and Unser 2003) © 2003 IEEE: (a) original brain atlas and patient MRI images overlaid in red–green; (b) after elastic registration with eight user-specified landmarks (not shown); (c) a cubic B-spline deformation field, shown as a deformed grid.

tect variations or pathologies (*cross-sectional* studies). When different imaging *modalities* are being registered, e.g., computed tomography (CT) scans and magnetic resonance images (MRI), *mutual information* measures of similarity are often necessary (Viola and Wells III 1997; Maes, Collignon *et al.* 1997).

Kybic and Unser (2003) provide a nice literature review and describe a complete working system based on representing both the images and the deformation fields as multi-resolution splines. Figure 9.6 shows an example of the Kybic and Unser system being used to register a patient’s brain MRI with a labeled brain atlas image. The system can be run in a fully automatic mode but more accurate results can be obtained by locating a few key *landmarks*. More recent papers on deformable medical image registration, including performance evaluations, include Klein, Staring, and Pluim (2007), Glocker, Komodakis *et al.* (2008), and the survey by Sotiras, Davatzikos, and Paragios (2013).

As with other applications, regular volumetric splines can be enhanced using selective refinement. In the case of 3D volumetric image or surface registration, these are known as *octree splines* (Szeliski and Lavallée 1996) and have been used to register medical surface models such as vertebrae and faces from different patients (Figure 9.7).

9.3 Optical flow

The most general (and challenging) version of motion estimation is to compute an independent estimate of motion at *each* pixel, which is generally known as *optical* (or *optic*) *flow*. As

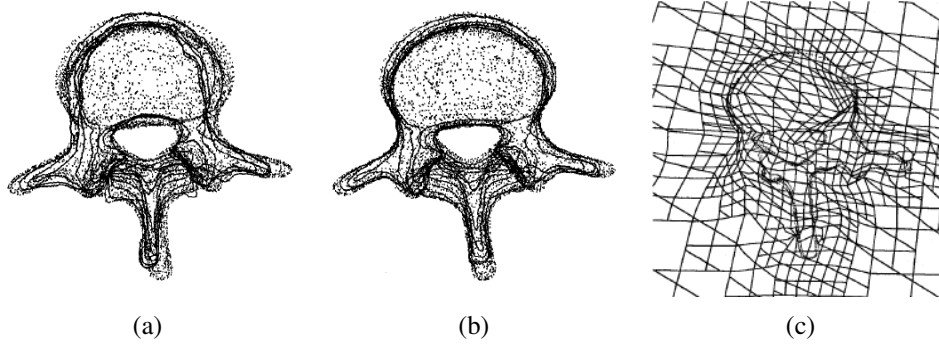


Figure 9.7 Octree spline-based image registration of two vertebral surface models (Szeliski and Lavallée 1996) © 1996 Springer: (a) after initial rigid alignment; (b) after elastic alignment; (c) a cross-section through the adapted octree spline deformation field.

we mentioned in the previous section, this generally involves minimizing the brightness or color difference between corresponding pixels summed over the image,

$$E_{\text{SSD-OF}}(\{\mathbf{u}_i\}) = \sum_i [I_1(\mathbf{x}_i + \mathbf{u}_i) - I_0(\mathbf{x}_i)]^2. \quad (9.56)$$

Because the number of variables $\{\mathbf{u}_i\}$ is twice the number of measurements, the problem is underconstrained. The two classic approaches to this problem are to perform the summation *locally* over overlapping regions (the *patch-based* or *window-based* approach) or to add smoothness terms on the $\{\mathbf{u}_i\}$ field using regularization or Markov random fields (Chapter 4) and to search for a global minimum. Good overviews of recent optical flow algorithms can be found in Baker, Scharstein *et al.* (2011), Sun, Yang *et al.* (2018), Janai, Güney *et al.* (2020), and Hur and Roth (2020).

The patch-based approach usually involves using a Taylor series expansion of the displaced image function (9.28) to obtain sub-pixel estimates (Lucas and Kanade 1981). Anandan (1989) shows how a series of local discrete search steps can be interleaved with Lucas–Kanade incremental refinement steps in a coarse-to-fine pyramid scheme, which allows the estimation of large motions, as described in Section 9.1.1. He also analyzes how the *uncertainty* in local motion estimates is related to the eigenvalues of the local Hessian matrix $\mathbf{A}_i$ (9.37), as shown in Figures 7.4 and 7.5.

Bergen, Anandan *et al.* (1992) develop a unified framework for describing both parametric (Section 9.2) and patch-based optical flow algorithms and provide a nice introduction to this topic. After each iteration of optical flow estimation in a coarse-to-fine pyramid, they re-wrap one of the images so that only incremental flow estimates are computed (Section 9.1.1).

When overlapping patches are used, an efficient implementation is to first compute the outer products of the gradients and intensity errors (9.33–9.34) at every pixel and then perform the overlapping window sums using a moving average filter.¹¹

Instead of solving for each motion (or motion update) independently, Horn and Schunck (1981) develop a regularization-based framework where (9.56) is simultaneously minimized over all flow vectors $\{\mathbf{u}_i\}$. To constrain the problem, smoothness constraints, i.e., squared penalties on flow derivatives, are added to the basic per-pixel error metric. Because the technique was originally developed for small motions in a variational (continuous function) framework, the linearized *brightness constancy constraint* corresponding to (9.28), i.e., (9.31), is more commonly written as an analytic integral

$$E_{\text{HS}} = \int (I_x u + I_y v + I_t)^2 dx dy, \quad (9.57)$$

where $(I_x, I_y) = \nabla I_1 = \mathbf{J}_1$, $I_t = e_i$ is the *temporal derivative*, i.e., the brightness change between images, and $u(x, y)$ and $v(x, y)$ are the 2D optical flow functions. The Horn and Schunck model can also be viewed as the limiting case of spline-based motion estimation as the splines become 1×1 pixel patches.

It is also possible to combine ideas from local and global flow estimation into a single framework by using a locally aggregated (as opposed to single-pixel) Hessian as the brightness constancy term (Bruhn, Weickert, and Schnörr 2005). Consider the discrete analog (9.28) to the analytic global energy (9.57),

$$E_{\text{HSD}} = \sum_i \mathbf{u}_i^T [\mathbf{J}_i \mathbf{J}_i^T] \mathbf{u}_i + 2e_i \mathbf{J}_i^T \mathbf{u}_i + e_i^2. \quad (9.58)$$

If we replace the per-pixel (rank 1) Hessians $\mathbf{A}_i = [\mathbf{J}_i \mathbf{J}_i^T]$ and residuals $\mathbf{b}_i = \mathbf{J}_i e_i$ with area-aggregated versions (9.33–9.34), we obtain a global minimization algorithm where region-based brightness constraints are used.

Another extension to the basic optical flow model is to use a combination of global (parametric) and local motion models. For example, if we know that the motion is due to a camera moving in a static scene (rigid motion), we can re-formulate the problem as the estimation of a per-pixel depth along with the parameters of the global camera motion (Adiv 1989; Hanna 1991; Bergen, Anandan *et al.* 1992; Szeliski and Coughlan 1997; Nir, Bruckstein, and Kimmel 2008; Wedel, Cremers *et al.* 2009). Such techniques are closely related to stereo matching (Chapter 12). Alternatively, we can estimate either per-image or per-segment affine motion models combined with per-pixel *residual* corrections (Black and Jepson 1996; Ju, Black, and

¹¹Other smoothing or aggregation filters can also be used at this stage (Bruhn, Weickert, and Schnörr 2005).

Jepson 1996; Chang, Tekalp, and Sezan 1997; Mémin and Pérez 2002). We revisit this topic in Section 9.4.

Of course, image brightness may not always be an appropriate metric for measuring appearance consistency, e.g., when the lighting in an image is varying. As discussed in Section 9.1, matching gradients, filtered images, or other metrics such as image Hessians (second derivative measures) may be more appropriate. It is also possible to locally compute the *phase* of steerable filters in the image, which is insensitive to both bias and gain transformations (Fleet and Jepson 1990). Papenberg, Bruhn *et al.* (2006) review and explore such constraints and also provide a detailed analysis and justification for iteratively re-warping images during incremental flow computation.

Because the brightness constancy constraint is evaluated at each pixel independently, rather than being summed over patches where the constant flow assumption may be violated, global optimization approaches tend to perform better near motion discontinuities. This is especially true if robust metrics are used in the smoothness constraint (Black and Anandan 1996; Bab-Hadiashar and Suter 1998a).¹² One popular choice for robust metrics is the L_1 norm, also known as *total variation* (TV), which results in a convex energy whose global minimum can be found (Bruhn, Weickert, and Schnörr 2005; Papenberg, Bruhn *et al.* 2006; Zach, Pock, and Bischof 2007b; Zimmer, Bruhn, and Weickert 2011). Anisotropic smoothness priors, which apply a different smoothness in the directions parallel and perpendicular to the image gradient, are another popular choice (Nagel and Enkelmann 1986; Sun, Roth *et al.* 2008; Werlberger, Trobin *et al.* 2009; Werlberger, Pock, and Bischof 2010). It is also possible to learn a set of better smoothness constraints (derivative filters and robust functions) from a set of paired flow and intensity images (Sun, Roth *et al.* 2008). Many of these techniques are discussed in more detail by Baker, Scharstein *et al.* (2011) and Sun, Roth, and Black (2014).

Because of the large, two-dimensional search space in estimating flow, most algorithms use variations of gradient descent and coarse-to-fine continuation methods to minimize the global energy function. This contrasts starkly with stereo matching, which is an “easier” one-dimensional disparity estimation problem, where combinatorial optimization techniques were the method of choice until the advent of deep neural networks.¹³ One way to deal with this complexity is to start with efficient patch-based correspondences (Kroeger, Timofte *et al.* 2016). Another way to deal with the large two-dimensional search space is to integrate sparse feature matches into a variational formulation, as was initially proposed by Brox and Malik (2010a). This approach was later extended by several authors, including

¹²Robust brightness metrics (Section 9.1, (9.2)) can also help improve the performance of window-based approaches (Black and Anandan 1996).

¹³Some exceptions to this trend of not exploring the full 4D cost volume can be found in Xu, Ranftl, and Koltun (2017) and Teed and Deng (2020b).

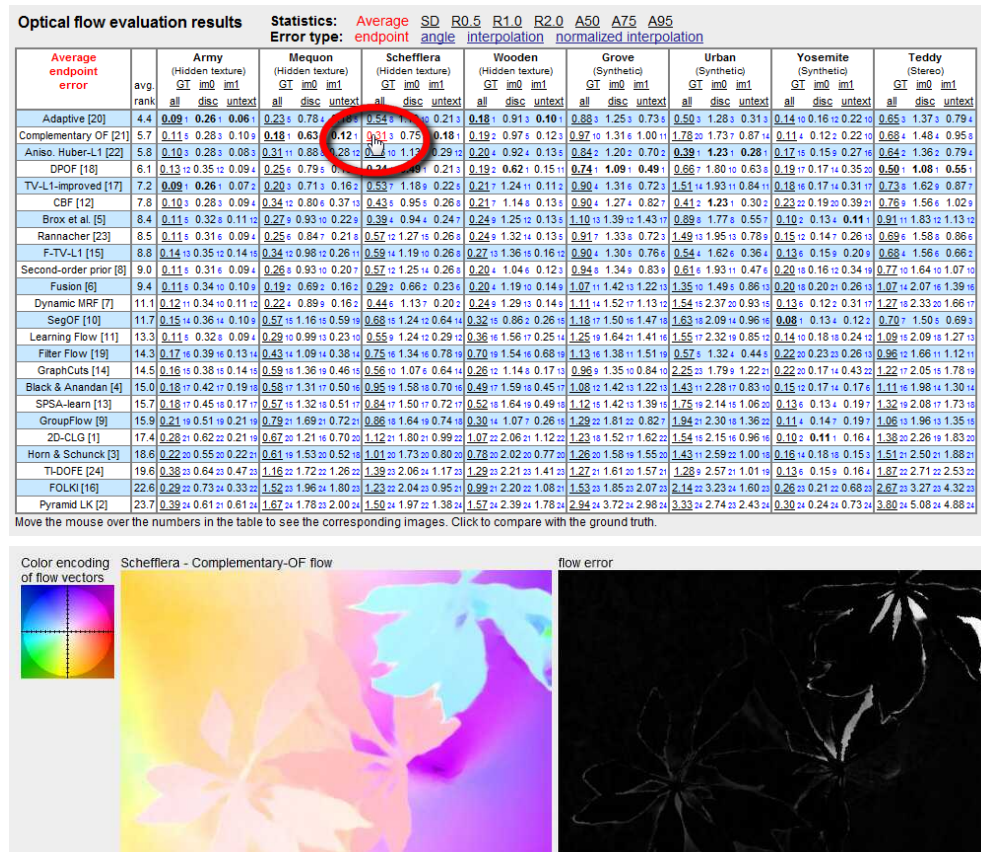


Figure 9.8 Evaluation of the results of 24 optical flow algorithms, October 2009, <https://vision.middlebury.edu/flow>, (Baker, Scharstein et al. 2009). By moving the mouse pointer over an underlined performance score, the user can interactively view the corresponding flow and error maps. Clicking on a score toggles between the computed and ground truth flows. Next to each score, the corresponding rank in the current column is indicated by a smaller blue number. The minimum (best) score in each column is shown in boldface. The table is sorted by the average rank (computed over all 24 columns, three region masks for each of the eight sequences). The average rank serves as an approximate measure of performance under the selected metric/statistic.

Weinzaepfel, Revaud *et al.* (2013), whose DeepFlow system use a hand-crafted (non-learned) convolutional network to compute initial quasi-dense correspondences, and Revaud, Weinzaepfel *et al.* (2015), whose EpicFlow system added an edge and occlusion-aware interpolation step before the variational optimization.

Combinatorial optimization methods based on Markov random fields were among the better-performing methods on the optical flow database of Baker, Scharstein *et al.* (2011)¹⁴ when it was originally released, but have now been overtaken by deep neural networks. Examples of such techniques include the one developed by Glocker, Paragios *et al.* (2008), who use a coarse-to-fine strategy with per-pixel 2D uncertainty estimates, which are then used to guide the refinement and search at the next finer level. Lempitsky, Roth, and Rother (2008) use fusion moves (Lempitsky, Rother, and Blake 2007) over proposals generated from basic flow algorithms (Horn and Schunck 1981; Lucas and Kanade 1981) to find good solutions.

A careful empirical analysis of these kinds of “classic” coarse-to-fine energy-minimization approaches is provided in the meticulously executed paper by Sun, Roth, and Black (2014).¹⁵ Figure 9.9a shows the main components of the framework they examine, including an initial warping based on the previous level’s flow (or a grid search at the coarsest level), followed by energy minimizing flow updates, and then an optional post-processing step. In their paper, the authors not only review dozens of variational (energy-minimization) approaches developed from the 1980s (Horn and Schunck 1981) through to 2013, but also show that algorithmic details such as median filtering post-processing, often glossed over by previous authors, have a strong influence on the results. In addition to performing their analysis on the Middlebury Flow dataset (Baker, Scharstein *et al.* 2011), they also evaluate on the newer Sintel dataset (Butler, Wulff *et al.* 2012).¹⁶

The field of accurate motion estimation continues to evolve at a rapid pace, with significant advances in performance occurring every year. While the Middlebury optical flow website (Figure 9.8) continues to be a good source of pointers to high-performing algorithms, more recent publications tend to focus (both training and evaluation) on the MPI Sintel dataset developed by Butler, Wulff *et al.* (2012), some samples of which are shown in Figure 9.1e–f. Some algorithms also train and test on the KITTI flow benchmark (Geiger, Lenz, and Urtasun 2012), although that dataset focuses on video acquired from a driving vehicle. In general, it appears that learning-based algorithms trained on one dataset still have trouble when applied to a different dataset.¹⁷

¹⁴<https://vision.middlebury.edu/flow>

¹⁵The earlier conference version of this paper had the eye-catching title of “Secrets of optical flow estimation and their principles” (Sun, Roth, and Black 2010).

¹⁶<http://sintel.is.tue.mpg.de>

¹⁷<http://www.robustvision.net>

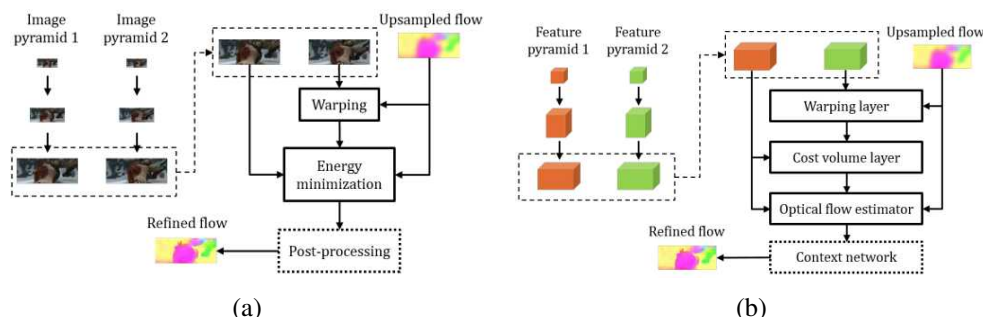


Figure 9.9 Iterative coarse-to-fine optical flow estimation (Sun, Yang et al. 2018) © 2018 IEEE: (a) “classic” variational (energy minimization) approach (Sun, Roth, and Black 2014); (b) newer neural network approach trained with end-to-end deep learning (Sun, Yang et al. 2018). Both figures show the processing at a single level of the coarse-to-fine pyramid, taking as input the flow computed by the previous (coarser) level and passing the refined flow onto the finer level below.

9.3.1 Deep learning approaches

Over the last decade, deep neural networks have become an essential component of all highly-performant optical flow algorithms, as described in the survey articles by Janai, Güney *et al.* (2020, Chapter 11) and Hur and Roth (2020). An early approach to use non-linear aggregation inspired by deep convolutional networks is the DeepFlow system of Weinzaepfel, Revaud *et al.* (2013), which uses a hand-crafted (non-learned) convolutions and pooling to compute multi-level response maps (matching costs), which are then optimized using a classic energy-minimizing variational framework.

The first system to use full deep end-to-end learning in an encoder-decoder network was FlowNetS (Dosovitskiy, Fischer *et al.* 2015), which was trained on the authors’ synthetic FlyingChairs dataset. The paper also introduced FlowNetC, which uses a correlation network (local cost volume). The follow-on FlowNet 2.0 system uses the initial flow estimates to warp the images and then refines the flow estimates using cascaded encoder-decoder networks (Ilg, Mayer *et al.* 2017), while subsequent papers also deal with occlusions and uncertainty modeling (Ilg, Saikia *et al.* 2018; Ilg, Çiçek *et al.* 2018).

An alternative to stacking full-resolution networks in series is to use image and flow pyramids together with coarse-to-fine warping and refinement, as first explored in the SPyNet paper by Ranjan and Black (2017). The more recent PWC-Net of Sun, Yang *et al.* (2018, 2019) shown in Figure 9.9b extends this idea by first computing a feature pyramid from each frame, warping the second set of features by the flow interpolated from the previous resolution

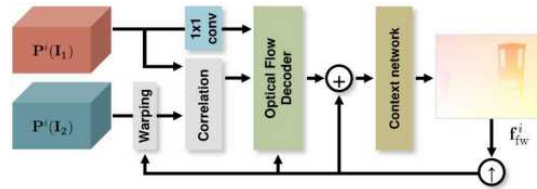


Figure 9.10 Iterative residual refinement optical flow estimation (Hur and Roth 2019) © 2019 IEEE. The coarse-to-fine cascade of Sun, Yang et al. (2018) in Figure 9.9b is replaced with a recurrent neural network (RNN) that cycles interpolated coarser level flow estimates as warping inputs to the next finer level but uses the same convolutional weights at each level.

level, and then computing a cost volume by correlating these features using a dot product between feature maps shifted by up to $d = \pm 4$ pixels. The refined optical flow estimates at the current level are produced using a multi-layer CNN whose inputs are the cost volume, the image features, and the interpolated flow from the previous level. A final context network takes as input the flow estimate and features from the second to last level and uses dilated convolutions to endow the network with a broader context. If you compare Figures 9.9a–b, you will see a pleasing correspondence between the various processing stages of classic and deep coarse-to-fine flow estimation algorithms.¹⁸

A variant on the coarse-to-fine PWC-Net developed by Hur and Roth (2019) is the Iterative Residual Refinement network shown in Figure 9.10. Instead of cascading a set of different deep networks as in FlowNet 2.0 and PWC-Net, IRRs re-use the same structure and convolution weights at each layer, which allows the network to be re-drawn in the “rolled up” version, as shown in this figure. The network can thus be thought of as a simple recurrent neural network (RNN) that upsamples the output flow estimates after each stage. In addition to having fewer parameters, this weight sharing also improves accuracy. In their paper, the authors also show how this network can be extended (doubled) to simultaneously compute forward and backward flows as well as occlusions.

In more recent work, Jonschkowski, Stone et al. (2020) take PWC-Net as their basic architecture and systematically study all of the components involved in training the flow estimator in an *unsupervised* manner, i.e., using regular real-world videos with no ground truth flow, which can enable much larger training sets to be used (Ahmadi and Patras 2016; Meister, Hur, and Roth 2018). In their paper, Jonschkowski et al. systematically compare photometric losses, occlusion estimation, self-supervision, and smoothness constraints, and analyze the

¹⁸Note that as with other coarse-to-fine warping approaches, these algorithms struggle with fast-moving fine structures that may not be visible at coarser levels (Brox and Malik 2010a).

effect of other choices, such as pre-training, image resolution, data augmentation, and batch size. They also propose four improvements to these key components, including cost volume normalization, gradient stopping for occlusion estimation, applying smoothness at the native low resolution, and image resizing for self-supervision. Another recent paper that explicitly deals with occlusions is Jiang, Campbell *et al.* (2021).

Another recent trend has been to model the uncertainty that arises in low field estimation due to homogeneous and occluded regions (Ilg, Çiçek *et al.* 2018). The HD³ network developed by Yin, Darrell, and Yu (2019) models correspondence distributions across multiple resolution levels, while the LiteFlowNet3 network of Hui and Loy (2020) extends their small and fast LiteFlowNet2 network (Hui, Tang, and Loy 2021) with cost volume modulation and low field deformation modules to significantly improve accuracy at minimal cost. In concurrent work, Hoflinger, Rota Bulò *et al.* (2020) introduce novel components such as replacing warping by sampling, smart gradient blocking, and knowledge distillation, which not only improve the quality of their low estimates but can also be used in other applications such as stereo matching. Teed and Deng (2020b) build on the idea of a recurrent network (Hur and Roth 2019), but instead of warping feature maps, they precompute a full $(W \times H)^2$ multi-resolution correlation volume (Recurrent All-Pairs Field Transforms or RAFT), which is accessed at each iteration based on the current low estimates. Computing a sparse correlation volume storing only the k closest matches for each reference image feature can further accelerate the computation (Jiang, Lu *et al.* 2021).

Given the rapid evolution in optical flow techniques, which is the best one to use? The answer is highly problem-dependent. One way to assess this is to look across a number of datasets, as is done in the Robust Vision Challenge.¹⁹ On this aggregated benchmark, variants of RAFT, IRR, and PWC all perform well. Another is to specifically evaluate a flow algorithm based on its intended use, and, if possible, to fine-tune the network on problem-specific data. Xue, Chen *et al.* (2019) describe how they fine-tune a SPyNet coarse-to-fine network on their synthetically degraded Vimeo-90K dataset to estimate *task-oriented flow* (TOFlow), which outperforms “higher accuracy” networks (and even ground truth flow) on three different video processing tasks, namely frame interpolation (Section 9.4.1), video denoising (Section 9.3.4), and video super-resolution. It is also possible to significantly improve the performance of learning-based flow algorithms by tailoring the synthetic training data to a target dataset (Sun, Vlasic *et al.* 2021).

¹⁹<http://www.robustvision.net/leaderboard.php?benchmark=low>

9.3.2 Application: Rolling shutter wobble removal

To save on silicon circuitry and enable greater photo sensitivity or fill factors, many CMOS imaging sensors such as those found in mobile phones use a *rolling shutter*, where different rows or columns are exposed in sequence. When photographing or filming a scene with fast scene or camera motions, this can result in straight lines becoming slanted or curved (e.g., the propeller blades on a plane or helicopter) or rigid parts of the scene wobbling (also known as the *jello effect*), e.g., when the camera is rapidly vibrating during action photography.

To compensate for these distortion, which are caused by different exposure times for different scanlines, accurate per-pixel optical flow must be estimated, as opposed to the whole-frame parametric motion that can sometimes be used for slower-motion video stabilization (Section 9.2.1). Baker, Bennett *et al.* (2010) and Forssén and Ringaby (2010) were among the first computer vision researchers to study this problem. In their paper, Baker, Bennett *et al.* (2010) recover a high-frequency motion field from the lower-frequency inter-frame motions and use this to resample each output scanline. Forssén and Ringaby (2010) perform similar computations using models of camera rotation, which require intrinsic lens calibration. Grundmann, Kwatra *et al.* (2012) remove the need for such calibration using mixtures of homographies to model the camera and scene motions, while Liu, Gleicher *et al.* (2011) use subspace constraints. Accurate rolling shutter correction is also required to produce high-quality image stitching results (Zhuang and Tran 2020).

While in some modern imaging systems such as action cameras, inertial measurements units (IMUs) can provide high-frequency estimates of camera motion, they cannot directly provide estimates of depth-dependent parallax and independent object motions. For this reason, the best in-camera image stabilizers use a combination of IMU data and sophisticated image processing.²⁰ Modeling rolling shutter is also important to obtain accurate pose estimates in structure from motion (Hedborg, Forssén *et al.* 2012; Kukeleva, Albl *et al.* 2018; Albl, Kukeleva *et al.* 2020; Kukeleva, Albl *et al.* 2020) and visual-inertial fusion in SLAM (Patron-Perez, Lovegrove, and Sibley 2015; Schubert, Demmel *et al.* 2018), which are discussed in Sections 11.4.2 and 11.5.

9.3.3 Multi-frame motion estimation

So far, we have looked at motion estimation as a two-frame problem, where the goal is to compute a motion field that aligns pixels from one image with those in another. In practice, motion estimation is usually applied to video, where a whole sequence of frames is available to perform this task.

²⁰<https://gopro.com/en/us/news/hero7-black-hypersmooth-technology>

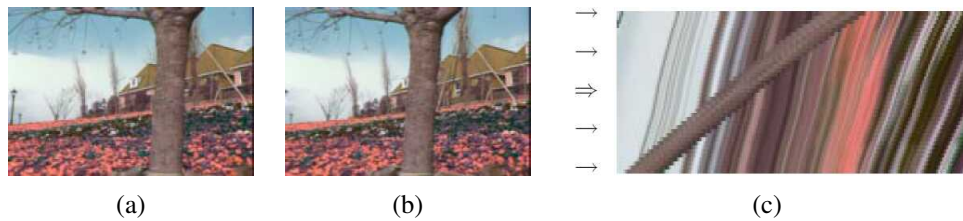


Figure 9.11 *Slice through a spatio-temporal volume (Szeliski 1999a) © 1999 IEEE: (a–b) two frames from the lower garden sequence; (c) a horizontal slice through the complete spatio-temporal volume, with the arrows indicating locations of potential key frames where flow is estimated. Note that the colors for the flower garden sequence are incorrect; the correct colors (yellow flowers) are shown in Figure 9.13.*

One classic approach to multi-frame motion is to *filter* the spatio-temporal volume using oriented or steerable filters (Heeger 1988), in a manner analogous to oriented edge detection (Section 3.2.3). Figure 9.11 shows two frames from the commonly used *flower garden* sequence, as well as a horizontal slice through the spatio-temporal volume, i.e., the 3D volume created by stacking all of the video frames together. Because the pixel motion is mostly horizontal, the slopes of individual (textured) pixel tracks, which correspond to their horizontal velocities, can clearly be seen. Spatio-temporal filtering uses a 3D volume around each pixel to determine the best orientation in space–time, which corresponds directly to a pixel’s velocity.

Unfortunately, to obtain reasonably accurate velocity estimates everywhere in an image, spatio-temporal filters have moderately large extents, which severely degrades the quality of their estimates near motion discontinuities. (This same problem is endemic in 2D window-based motion estimators.) An alternative to full spatio-temporal filtering is to estimate more local spatio-temporal derivatives and use them inside a global optimization framework to fill in textureless regions (Bruhn, Weickert, and Schnörr 2005; Govindu 2006).

Another alternative is to simultaneously estimate multiple motion estimates, while also optionally reasoning about occlusion relationships (Szeliski 1999a). Figure 9.11c shows schematically one potential approach to this problem. The horizontal arrows show the locations of keyframes s where motion is estimated, while other slices indicate video frames t whose colors are matched with those predicted by interpolating between the keyframes. Motion estimation can be cast as a global energy minimization problem that simultaneously minimizes brightness compatibility and flow compatibility terms between keyframes and other frames, in addition to using robust smoothness terms.

The multi-view framework is potentially even more appropriate for rigid scene motion

(multi-view stereo) (Section 12.7), where the unknowns at each pixel are disparities and occlusion relationships can be determined directly from pixel depths (Szeliski 1999a; Kolmogorov and Zabih 2002). However, it is also applicable to general motion, with the addition of models for occlusion relationships, as in the MirrorFlow system of Hur and Roth (2017) as well as multi-frame versions (Janai, Guney *et al.* 2018; Neoral, Sochman, and Matas 2018; Ren, Gallo *et al.* 2019).

9.3.4 Application: Video denoising

Video denoising is the process of removing noise and other artifacts such as scratches from film and video (Kokaram 2004; Gai and Kang 2009; Liu and Freeman 2010). Unlike single image denoising, where the only information available is in the current picture, video denoisers can average or borrow information from adjacent frames. However, to do this without introducing blur or jitter (irregular motion), they need accurate per-pixel motion estimates. One way to do this is to use task-oriented low, where the low network is specifically tuned end-to-end to provide the best denoising performance (Xue, Chen *et al.* 2019).

Exercise 9.6 lists some of the steps required, which include the ability to determine if the current motion estimate is accurate enough to permit averaging with other frames. And while some recent papers continue to estimate low as part of the multi-frame denoising pipeline (Tassano, Delon, and Veit 2019; Xue, Chen *et al.* 2019), others either concatenate similar patches from different frames (Maggioni, Boracchi *et al.* 2012) or concatenate small subsets of frames into a deep network that never explicitly estimates a motion representation (Claus and van Gemert 2019; Tassano, Delon, and Veit 2020). A more general form of video enhancement and restoration called *video quality mapping* has also recently started being investigated (Fuoli, Huang *et al.* 2020).

9.4 Layered motion

In many situations, visual motion is caused by the movement of a small number of objects at different depths in the scene. In such situations, the pixel motions can be described more succinctly (and estimated more reliably) if pixels are grouped into appropriate objects or *layers* (Wang and Adelson 1994).

Figure 9.12 shows this approach schematically. The motion in this sequence is caused by the translational motion of the checkered background and the rotation of the foreground hand. The complete motion sequence can be reconstructed from the appearance of the foreground and background elements, which can be represented as alpha-matted images (*sprites* or *video*

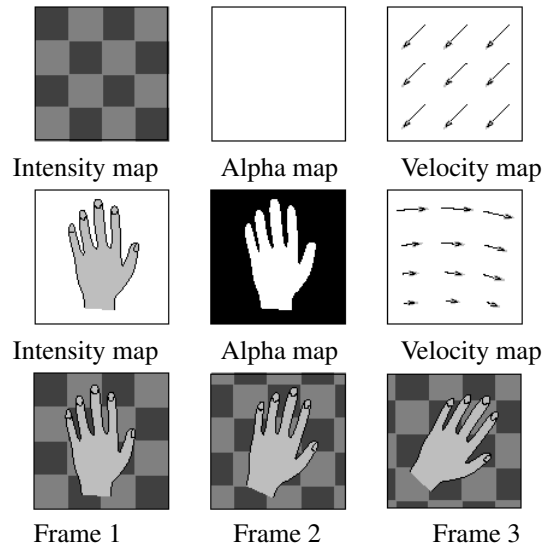


Figure 9.12 *Layered motion estimation framework (Wang and Adelson 1994) © 1994 IEEE: The top two rows describe the two layers, each of which consists of an intensity (color) image, an alpha mask (black=transparent), and a parametric motion field. The layers are composited with different amounts of motion to recreate the video sequence.*

objects) and the parametric motion corresponding to each layer. Displacing and compositing these layers in back to front order (Section 3.1.3) recreates the original video sequence.

Layered motion representations not only lead to compact representations (Wang and Adelson 1994; Lee, Chen *et al.* 1997), but they also exploit the information available in multiple video frames, as well as accurately modeling the appearance of pixels near motion discontinuities. This makes them particularly suited as a representation for image-based rendering (Section 14.2.1) (Shade, Gortler *et al.* 1998; Zitnick, Kang *et al.* 2004) as well as object-level video editing.

To compute a layered representation of a video sequence, Wang and Adelson (1994) first estimate affine motion models over a collection of non-overlapping patches and then cluster these estimates using k-means. They then alternate between assigning pixels to layers and recomputing motion estimates for each layer using the assigned pixels, using a technique first proposed by Darrell and Pentland (1991). Once the parametric motions and pixel-wise layer assignments have been computed for each frame independently, layers are constructed by warping and merging the various layer pieces from all of the frames together. Median filtering is used to produce sharp composite layers that are robust to small intensity variations,