Artificial Neural Network

9

Law of nature is The Way.

9.1 Introduction

When comes to learning and classifications, no other tool is more efficient and powerful than human brains. Therefore, there is sufficient motivation to design a machine learning tool which simulates human brains. This is further encouraged by the recent research findings on human brain from both cognitive science and biology. It is believed that a human brain is consisting of 10s of billions of neurons interconnected into a sophisticated network. The neurons in a brain are organized into functional units or regions, such as regions for visual, auditory, motion, reasoning, speech, etc. An individual neuron is shown in Fig. 9.1.

It is found that a neuron receives inputs from its dendrites, processes them in the cell body, and transmits the output signal to other neurons through its axon. The inputs the neuron received can be either *excitatory* or *inhibitory*. When there are more excitatory inputs than inhibitory inputs, the neuron is activated and a signal is transmitted out through the axon; otherwise, no signal is generated.

Then, neurons in many regions of a human brain are further organized into layers to create a layered network. Neurons from one layer usually receive inputs from neurons in an adjacent layer. Connections between layers are mostly in one direction, moving from low-level layer sensors like eyes or ears to higher coordination and reasoning layer [1].

With these understandings of human brains and neurons, it is possible to design artificial neurons and an artificial neural network or ANN.

D. Zhang, *Fundamentals of Image Data Mining*, Texts in Computer Science, https://doi.org/10.1007/978-3-030-17989-2_9

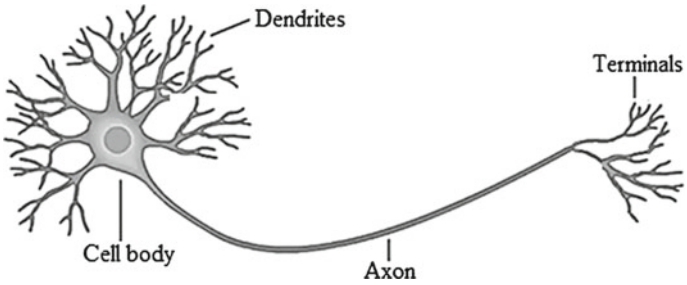


Fig. 9.1 A neuron of human brain

9.2 Artificial Neurons

The design of an artificial neural network starts with the modeling of artificial neurons. From the above understandings, a neuron is basically a unit which receives inputs and generates an output. Specifically, a biological neuron consists of three components: the inputs (dendrites), an activation or processing unit (cell body) and an output (axon). Electronically, these three components can be respectively represented as a set of inputs x_i, a weighted sum of the inputs Σ, and a threshold of the weighted sum. The alignment of an electronic neuron and a biological neuron is shown in Fig. 9.2.

The weighted sum and the thresholding are usually merged into a single activation unit and the axon is replaced with an output signal. Therefore, the simplified artificial neuron is shown in Fig. 9.3.

It turns out that an artificial neuron is just a binary linear classifier. Given an input $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, a weighted sum Σ is calculated and compared with a threshold T:

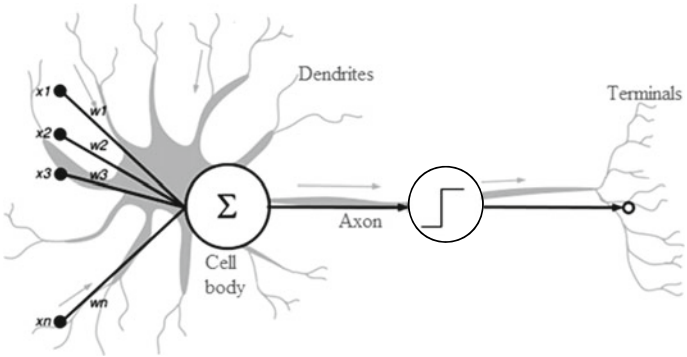


Fig. 9.2 The alignment of an artificial neuron with a biologic neuron

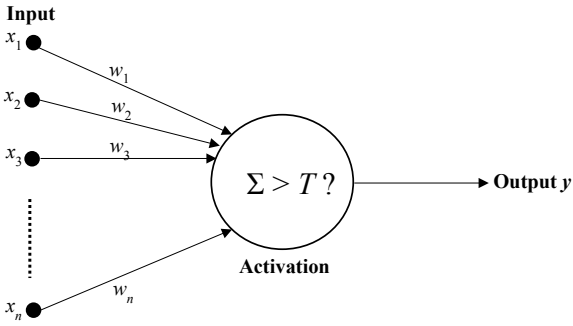


Fig. 9.3 The modeling of an artificial neuron

$$\Sigma = w_1x_1 + w_2x_2 + \cdots + w_nx_n > T \tag{9.1}$$

If $\Sigma > T$, the neuron is activated and an output signal y is sent out. In other words, the activation of the neuron or output y is based on the following rule:

$$y = \begin{cases} 1 & \Sigma > T \\ 0 & \Sigma < T \end{cases} \tag{9.2}$$

It is more convenient to combine both Σ and T and rewrite (9.1) as follows:

$$\begin{aligned} D = \Sigma - T &= w_1x_1 + w_2x_2 + \cdots + w_nx_n - T \\ &= -T + w_1x_1 + w_2x_2 + \cdots + w_nx_n \end{aligned} \tag{9.3}$$

For notation purpose, let $w_0\ x_0 = -T$ which represents a constant and $x_0 = 1$, then (9.3) becomes

$$D = w_0x_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n \tag{9.4}$$

And (9.2) becomes

$$y = \begin{cases} 1 & D > 0 \\ 0 & D < 0 \end{cases} \tag{9.5}$$

We can use this artificial neuron to do many simple linear classifications. One of them is to simulate the logic gates, such as the AND, OR, NAND, NOR, etc.

9.2.1 An AND Neuron

Let us start with the AND gate, the AND function is given in Table 9.1.

Now, if we set the activation or threshold value as $T = 1.5$ (or any value between 1 and 2), only input (1, 1) will be activated, this is exactly what we want. Therefore, the classifier for the AND function is given as follows:

Table 9.1 AND gate

Input		Output	Sum
x_1	x_2		
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	2

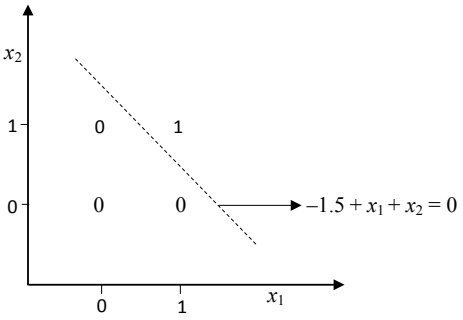


Fig. 9.4 The data of AND can be separated by a single line $-1.5 + x_1 + x_2 = 0$

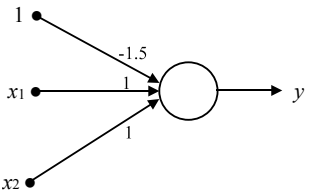


Fig. 9.5 The neuron which implements the AND function

$$y = -1.5 + x_1 + x_2 \tag{9.6}$$

In space, the classifier of (9.6) is represented by the following hyperplane which is a line in this case:

$$-1.5 + x_1 + x_2 = 0 \tag{9.7}$$

The AND data and the linear classifier is shown in Fig. 9.4. The neuron which implements the AND function is shown in Fig. 9.5 [1].

9.2.2 An OR Neuron

Similarly, the classifier of an OR function and the neuron that implements the OR function are shown in Figs. 9.6 and 9.7, respectively.

An artificial neuron is called a node in an artificial neural network and is usually represented by a circle.

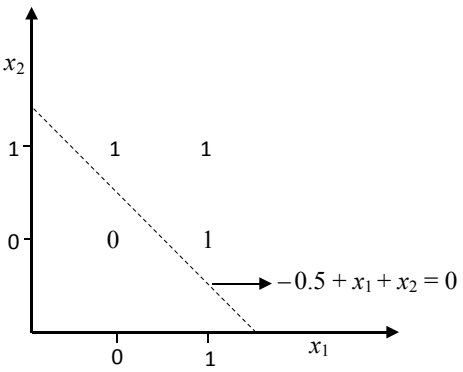


Fig. 9.6 The data of OR can be separated by a single line $-0.5 + x_1 + x_2 = 0$

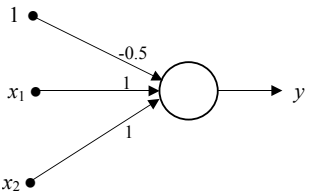


Fig. 9.7 The neuron which implements the OR function

9.3 Perceptron

The neuron we designed above can do binary and linear classification; however, both the weights and the threshold are predetermined by a human designer. A biological neuron of a human brain, on the other hand, can learn from new instances and memorize. It would be desirable that an artificial neuron can also learn and memorize. This is done by feeding the neuron with a set of known data or pre-labeled data and learn the weights by using certain criterion or algorithm such as minimizing the total error.

From (9.4), the decision function of a neuron is given as follows:

$$Y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n \tag{9.8}$$

To train the neuron:

- A training set $S = \{(\mathbf{x}_q, y_q), q = 1, 2, \ldots, N\}$ is collected,
- where y_q is the desired output of qth sample feature vector $\mathbf{x}_q = (x_{q1}, x_{q2}, \ldots, x_{qn})$.
- The training is then to minimize the squared error or MSE:

$$E = E(w_1, w_2, \ldots, w_n) = \frac{1}{2}\sum_{q=1}^{N} \left(Y_q - y_q\right)^2 \tag{9.9}$$

- where $Y_q = w_0 + w_1\, x_{q1} + w_2\, x_{q2} + \cdots + w_n\, x_{qn}$.

The minimization follows the *steepest descent* direction which is given by the gradient vector of

$$\left(-\frac{\partial E}{\partial w_0}, -\frac{\partial E}{\partial w_1}, -\frac{\partial E}{\partial w_2}, \ldots, -\frac{\partial E}{\partial w_n}\right) \tag{9.10}$$

If a sample is misclassified and the actual output Y is different from the correct output y, we would want to change the weights so that E is minimized. Therefore, the *steepest descent* algorithm to minimize E is given by the following algorithm:

1. Choose an initial weight set of w_0, w_1, w_2, …, w_n and a positive constant c.
2. For $i = 0, 1, 2, \ldots, n$, compute the partial derivatives of $\partial E/\partial w_i$ and let $w_i = w_i - c(\partial E/\partial w_i)$.
3. Repeat step 2 until w_0, w_1, w_2, …, w_n stop to change.

By combining (9.9) and (9.10), the partial derivatives $\partial E/\partial w_i$ in the above algorithm is given by

$$\frac{\partial E}{\partial w_i} = (Y_q - y_q)x_{qi}, \quad i = 0, 1, 2, \ldots, n \tag{9.11}$$

Therefore, the MSE learning algorithm of a *perceptron* is given as follows:

1. Choose an initial weight set of w_0, w_1, w_2, …, w_n and a positive constant c.
2. For each of samples $q = 1, 2, \ldots, N$, compute $Y_q = \sum_{i=0}^{n} w_i x_{qi}$.
3. Let $w_i = w_i - c(Y_q - y_q)x_{qi}$ for $i = 0, 1, 2, \ldots, n$.
4. Repeat steps 2 and 3 until w_0, w_1, w_2, …, w_n stop to change.

9.4 Nonlinear Neural Network

The perceptron is essentially a two-layer neural network with an input layer and an output layer, it can be trained to classify any linear data which can be separated by a hyperplane. However, for nonlinear data such as XOR (Fig. 9.8) and other data with convex data regions (Fig. 9.9), they cannot be separated by a single hyperplane in space, and consequently they cannot be classified by a perceptron.

Although a convex data region in space (a region is convex if any two data points can be connected by a line segment inside the region) cannot be separated by a single hyperplane, its boundary can be approximated by the intersection of a finite number of hyperplanes. For example, the XOR data in Fig. 9.8 can be separated by the following two half planes in 2D space:

$$-0.5 + x_1 + x_2 > 0 \quad \text{and} \quad -1.5 + x_1 + x_2 < 0 \tag{9.12}$$

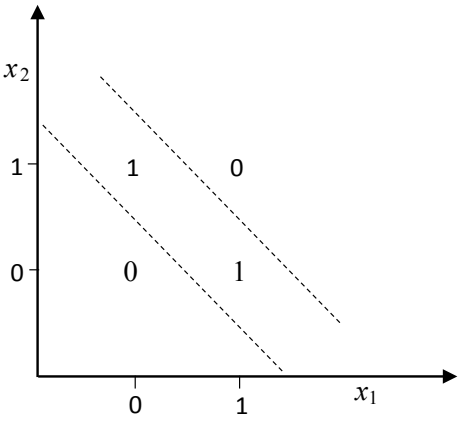


Fig. 9.8 Outputs of an XOR cannot be separated by a single line in 2D space

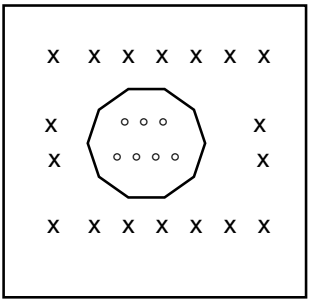


Fig. 9.9 The convex data region in the center cannot be separated by a single line in 2D space

The two neurons represent the two half planes in (9.12) which are given by

$$-0.5 + x_1 + x_2 > 0 \quad \text{and} \quad 1.5 - x_1 - x_2 > 0 \tag{9.13}$$

Now, by AND(ing) (Fig. 9.5) the two neurons of (9.13), a neural network with a middle layer or hidden layer is created which can separate the XOR outputs. The neural network with hidden layer is shown in Fig. 9.10 [1].

This idea of three-layer neural network can be easily extended to classify any generic convex nonlinear data like the one shown in Fig. 9.9. All we need now is more nodes in the second layer or middle layer (Fig. 9.11). Usually, a small number of nodes in the middle layer are sufficient to separate a convex region; however, more middle layer nodes produce a smoother region boundary.

By extending the above idea of classifying *convex* nonlinear data using neural network with a hidden layer, it is also possible to classify *non-convex* data using neural network. This is because a *non-convex* region can always be approximated by the union of a finite number of convex regions.

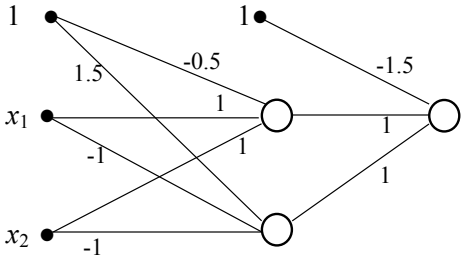


Fig. 9.10 A three-layer neural network to implement XOR. The two linear classifiers at the hidden layer are ANDed at the third layer

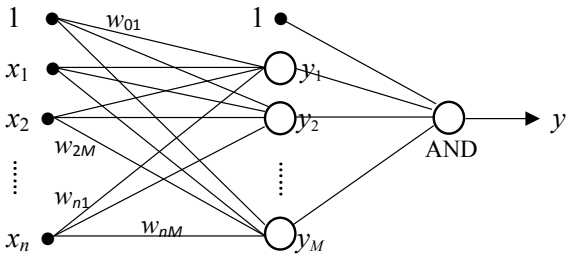


Fig. 9.11 A three-layer neural network with a hidden layer which can classify generic convex nonlinear data

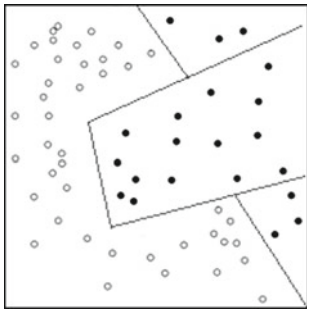


Fig. 9.12 A non-convex data region (black dots) is approximated by three convex regions

Therefore, it is possible to create a number of convex nodes in the third layer using the method discussed above and combine those convex nodes in the third layer using a logic OR in the fourth layer.

For example, the non-convex data region (black dots) in Fig. 9.12 can be approximated by three convex data regions, which can then be classified using a four-layer neural network shown in Fig. 9.13.

This indicates that any *nonlinear* data can be classified by a four-layer neural network.

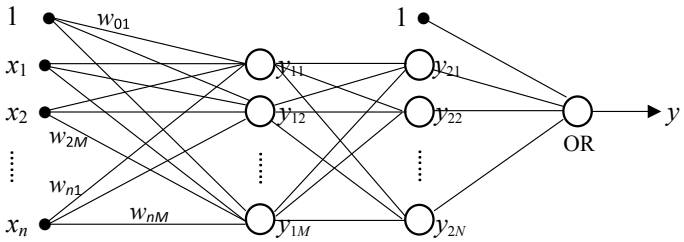


Fig. 9.13 A four-layer neural network which can classify any nonlinear data

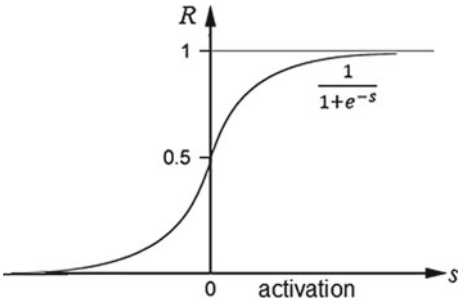
9.5 Activation and Inhibition

It should be noted that the activation or threshold function is crucial to a neural network, because it is shown that any neural network without thresholding is equivalent to a two-layer network which cannot separate nonlinear data. However, the binary threshold function is not desirable because it is not continuous. This non-continuity can cause the network to take a very long time to converge or even not converge. This is because the gradients from the MSE are differential values and are small, so the changes in the weights are also small. As the result, the small changes to the weights are usually not enough to pass the threshold or generate an output signal.

9.5.1 Sigmoid Activation

It is desirable to have a continuous activation function which changes continuously from 0 to 1 instead of jumping from 0 to 1. Among the many proposed continuous activation functions, the most widely used is the *sigmoid function*. The *S*-shaped *sigmoid function* is defined as (9.14) and the shape of the function is shown in Fig. 9.14.

Fig. 9.14 Sigmoid function



The $R(s)$ function guarantees an output signal while preserving the thresholding functionality of an activation function.

$$R(s) = \frac{1}{1 + e^{-s}} \tag{9.14}$$

The $R(s)$ function has the following important properties:

1. $\lim_{s \to -\infty} R(s) = 0$
2. $\lim_{s \to \infty} R(s) = 1$
3. $R(0) = ½$
4. $\dfrac{dR}{ds} = R(1 - R)$ (9.15)

This final property has a convenient use in the following backpropagation algorithm. $R(1 - R)$ is close to 0 at both ends of region (0, 1).

9.5.2 Shunting Inhibition

It is known that a biological neuron can be either *excitatory* or *inhibitory*. An inhibitory signal prevents impulse from arising in the receiving neuron. This inhibitory phenomenon can be represented mathematically as reducing the excitatory potential by division. This is called *shunting inhibitory* or SI. The idea is to learn a dual set of weights D_j and divided the original output of a neuron with the dual output adjusted by a decay factor. An SI neuron is illustrated in Fig. 9.15 [2].

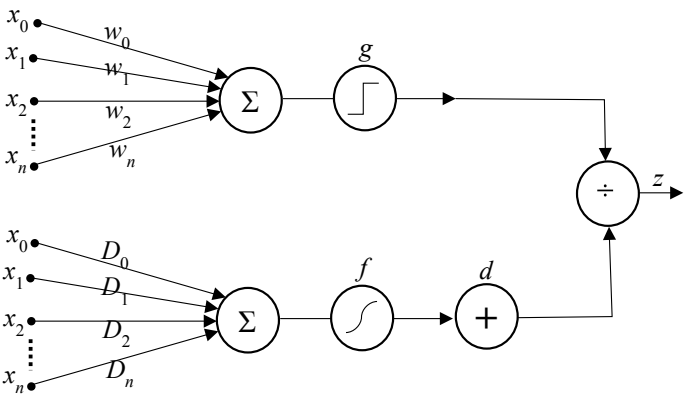


Fig. 9.15 A shunting inhibitory neuron

Mathematically, the output from an SI neuron is given by (9.16) [2].

$$z = \frac{g(\sum_{j=1}^{n} w_j x_j + b)}{d + f(\sum_{j=1}^{n} D_j x_j + a)} \tag{9.16}$$

where

- z is the output of the shunting neuron,
- x_j is the jth input,
- C_j and D_j are the connection weights of the jth input,
- $a = w_0 x_0$ and $b = D_0 x_0$ are biases,
- d is the passive decay rate,
- f and g are activation functions,
- n is the number of inputs from the previous layer,
- $d + f(\sum_{j=1}^{n} D_j x_j + a) > 0$.

An ANN designed with SI neurons is called an SIANN. The key to an SI neural network is to use different activation functions f and g in a layer so that only the strongest neurons are activated. Experiments show that when f and g are hyperbolic tangent function and exponential function, respectively, the network has better convergence.

9.6 The Backpropagation Neural Network

9.6.1 The BP Network and Error Function

One of humans' great learning skills is learning from mistakes or errors, this is because the mistakes/errors provide important feedback to improve the original learning process. Unfortunately, the conventional ANN described above does not provide this function. However, this function can be simulated in a neural network by using the backpropagation algorithm. The idea is to add another process so that the outputs in the final layer are used as feedback to the previous layer to update the weights, and repeat this feedback until the input layer.

Therefore, a *backpropagation neural network* or a BP-ANN consists of two major processes: (1) a conventional feedforward process which computes outputs at each layer starting from the input layer; (2) a backpropagation process where the weights are updated at each layer starting from the output layer, an attempt to improve the classification accuracy.

In order to formulate the backpropagation algorithm, the following notations are defined:

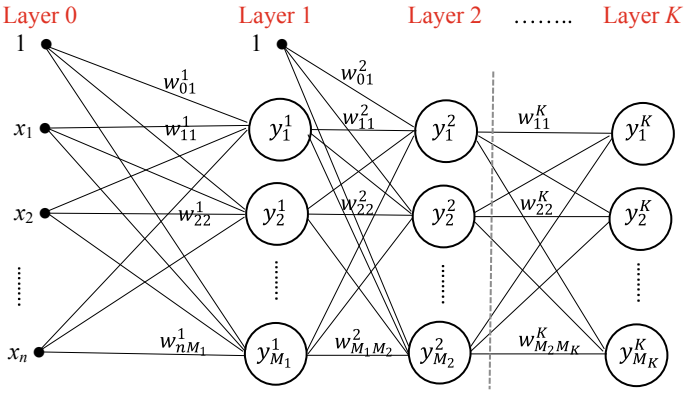


Fig. 9.16 A K-layer backpropagation neural network

- k denotes the kth layer of a network: $k = 0, 1, 2, \ldots, K$. Layer 0 is the input and layer K is the output.
- M_k denotes the number of nodes at layer k, $k = 1, 2, \ldots, K$.
- $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ stands for a training sample data.
- w_{ij}^k stands for the weight of connection between node i at layer $k-1$ and node j at layer k, $k = 1, 2, \ldots, K$.
- $net_j^k = \sum_{i=0}^{M_{k-1}} w_{ij}^k y_i^{k-1}$ stands for the output or the weighted sum of jth node of layer k, $j = 0, 1, 2, \ldots, M_k$, $k = 1, 2, \ldots, K$.
- $y_j^k = R(net_j^k)$ stands for the activated or thresholded output from the jth node of layer k, $j = 0, 1, 2, \ldots, M_k$, $k = 1, 2, \ldots, K$.

The notations are shown in the following backpropagation neural network.

The BP-ANN uses the same MSE and steepest gradient descent optimization as in the conventional ANN described earlier from (9.9) to (9.11). Assume t_j is the true output of node j at the final layer, and then the total squared error of the BP network in Fig. 9.16 is given by (9.17).

$$E = \frac{1}{2}\sum_{j=1}^{M_K}\left(y_j^K - t_j\right)^2 \tag{9.17}$$

The backpropagation algorithm starts from estimating and updating the weights of the final layer K, and then propagates the same estimating and updating procedure back to layer $K-1$, $K-2$, …, until layer 1.

9.6.2 Layer *K* Weight Estimation and Updating

- The weights of layer K are given by w_{ij}^K, $i = 1, 2, \ldots, M_{K-1}$; $j = 1, 2, \ldots, M_K$.
- To estimate w_{ij}^K, we compute the partial derivatives $\frac{\partial E}{\partial w_{ij}^K}$.
- We will make use of the fourth property of $R(s)$ in (9.15) for the following computations.
- Remember every y_j^k is a $R(s)$ function: $y_j^k = R(net_j^k)$.
- Figure 9.17 shows the connection with weight w_{ij}^K (red line) in a BP network.

Therefore, the computation of $\frac{\partial E}{\partial w_{ij}^K}$ is simple, because among the M_K terms in (9.17), only the jth term $\left(y_j^K - t_j\right)^2$ is related to w_{ij}^K, while all the other terms are irrelevant because they are not connected to node i in layer $K-1$. Therefore, we have the following:

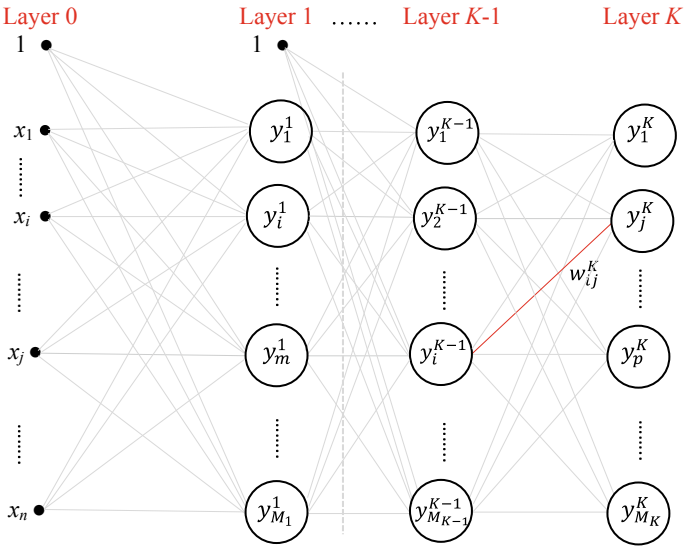


Fig. 9.17 Illustration of a connection between node i in layer $K-1$ and node j in layer K in a K-layer BP neural network

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^K} &= \frac{\partial E}{\partial y_j^K}\frac{\partial y_j^K}{\partial w_{ij}^K} \\ &= \left(y_j^K - t_j\right)\frac{\partial y_j^K}{\partial w_{ij}^K} \\ &= \left(y_j^K - t_j\right)\frac{\partial y_j^K}{\partial net_j^K}\frac{\partial net_j^K}{\partial w_{ij}^K} \\ &= \left(y_j^K - t_j\right)y_j^K\left(1 - y_j^K\right)y_i^{K-1} \\ &= \delta_j^K y_i^{K-1}\end{aligned} \tag{9.18}$$

where

$$\delta_j^K = y_j^K\left(1 - y_j^K\right)\left(y_j^K - t_j\right) \tag{9.19}$$

Therefore, the weights of layer K are updated according to (9.19) as given below:

$$w_{ij}^K \leftarrow w_{ij}^K - c\frac{\partial E}{\partial w_{ij}^K} = w_{ij}^K - c\delta_j^K y_i^{K-1} \tag{9.20}$$

where c is a positive constant. δ_j^K is equivalent to a *network sensor*, it measures if the signal of a connection should be raised or suppressed. This can be explained by (9.19) and (9.20):

- w_{ij}^K is increased (raised) if $\delta_j^K < 0$ (when $y_j^K < t_j$);
- w_{ij}^K is decreased (suppressed) if $\delta_j^K > 0$ (when $y_j^K > t_j$);
- w_{ij}^K changes little if δ_j^K is close to 0 (when y_j^K is close to t_j, 0 or 1), indicating converging;
- w_{ij}^K changes most significantly if y_j^K is very different from t_j.

9.6.3 Layer $K - 1$ Weight Estimation and Updating

- The weights of layer $K - 1$ are given by w_{mi}^{K-1}, $m = 1, 2, \ldots, M_{K\text{-}2}$; $i = 1, 2, \ldots, M_{K-1}$.
- To estimate w_{mi}^{K-1}, we compute the partial derivatives $\frac{\partial E}{\partial w_{mi}^{K-1}}$.
- The computation of partial derivative $\frac{\partial E}{\partial w_{mi}^{K-1}}$ at layer $K - 1$ is more complicated than layer K.

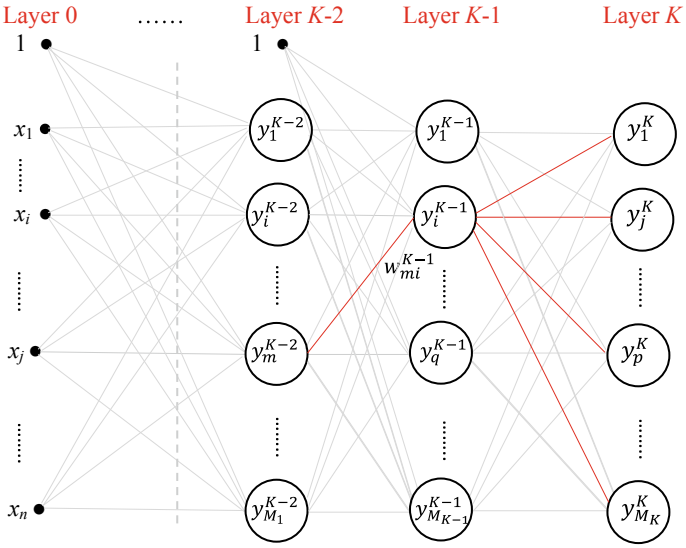


Fig. 9.18 Illustration of all connections relevant to w_{mi}^{K-1} in a K-layer BP neural network. Relevant connections are shown in red lines

- This is because w_{mi}^{K-1} connects to node i (at layer $K-1$), which is then connected to all nodes at layer K.
- The connections which are relevant to w_{mi}^{K-1} are shown in Fig. 9.18 (red lines).
- Therefore, each of the M_K terms in (9.17) is now related to the computation of $\frac{\partial E}{\partial w_{mi}^{K-1}}$.

Therefore, by using $y_j^k = R(net_j^k)$ and $dR(s)/ds = R(s)[1 - R(s)]$, the computation of $\frac{\partial E}{\partial w_{mi}^{K-1}}$ is given below by the sum of chaining derivatives:

$$\begin{aligned}
\frac{\partial E}{\partial w_{mi}^{K-1}} &= \frac{1}{2}\sum_{j=1}^{M_K}\left(\frac{\partial\left(y_j^K - t_j\right)^2}{y_j^K}\frac{\partial y_j^K}{\partial net_j^K}\frac{\partial net_j^K}{\partial y_i^{K-1}}\frac{\partial y_i^{K-1}}{\partial net_i^{K-1}}\frac{\partial net_i^{K-1}}{\partial w_{mi}^{K-1}}\right) \\
&= \left[\sum_{j=1}^{M_K}\left(y_j^K - t_j\right)y_j^K\left(1 - y_j^K\right)w_{ij}^K\right]y_i^{K-1}\left(1 - y_i^{K-1}\right)y_i^{K-2} \qquad (9.21) \\
&\overset{(9.19)}{\Rightarrow} \left[y_i^{K-1}\left(1 - y_i^{K-1}\right)\sum_{j=1}^{M_K}\delta_j^K w_{ij}^K\right]y_i^{K-2} \\
&= \delta_i^{K-1} y_i^{K-2}
\end{aligned}$$

where

$$\delta_i^{K-1} = y_i^{K-1}\left(1 - y_i^{K-1}\right)\sum_{j=1}^{M_K}\delta_j^K w_{ij}^K \tag{9.22}$$

By repeating (9.21), it can be shown that for *any hidden layer k*, δ_i^k is given by (9.23).

$$\delta_i^k = y_i^k\left(1 - y_i^k\right)\sum_{j=1}^{M_{k+1}}\delta_j^{k+1} w_{ij}^{k+1} \tag{9.23}$$

Equation (9.23) indicates that a network sensor at layer k depends on the combined network sensors at next layer $k + 1$. In other words, during the back propagation, the weighted sum of network sensors at layer $k + 1$ has been propagated (through $dR(s)/ds$) to each network sensor at previous layer k. This kind of propagation is similar to the feedforward process where the weighted sum of network values (or signals) at layer k is propagated (through $R(s)$) to each connection at next layer $k + 1$.

The difference between the two rounds of propagation is that different propagation functions are used. In the feedforward process, the propagation function is just the activation function $R(s)$ itself, while in the backpropagation process, the propagation function is the gradient of the activation function: $dR(s)/ds$.

9.6.4 The BP Algorithm

Now that we have computed the gradients or partial derivatives of the error function E, the BP algorithm is designed as following [1]:

1. Initialize all the weights w_{ij}^k (for all i, j, k) on the network and the constant c with some small random values.
2. Input a new training data: $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ from a set of N training data.
3. *Feedforward step*. Compute the outputs at each layer starting from the input layer:

$$y_j^k = R\left(\sum_{i=0}^{M_{k-1}} w_{ij}^k y_i^{k-1}\right) \quad j = 1, 2, \ldots, M_k; \;\; k = 1, 2, \ldots, K.$$

4. *Backpropagation step*. Compute the network sensors at each layer starting from the output layer:

$$\delta_j^K = y_j^K\left(1 - y_j^K\right)\left(y_j^K - t_j\right) \quad (\text{for layer } K)$$

$$\text{and} \quad \delta_i^k = y_i^k\left(1 - y_i^k\right)\sum_{j=1}^{M_{k+1}} \delta_j^{k+1} w_{ij}^{k+1} \quad (\text{for } k = K-1, K-2, \ldots, 2, 1).$$

5. Update weights on the network by $w_{ij}^k \leftarrow w_{ij}^k - c\delta_j^k y_i^{k-1}$ for all i, j, k.
6. Repeat steps 2 and 5 until all the weights w_{ij}^k stop to change or stop to change significantly.

Because the BP algorithm is a steepest gradient descent algorithm, choosing the initial values for w_{ij}^k and the constant c is crucial to the performance of a network. If the initial values of w_{ij}^k are too far from the global minima, the algorithm may converge to a local minimum. Consequently, the result of class boundaries is not accurate. If the initial value of the constant c is either too small or too big, the converging progress can be very slow. In practice, several rounds of guessing the initial values of both the weights and c may be required to achieve a desirable performance.

9.7 Convolutional Neural Network

An ordinary ANN does not take raw data as input; instead, it takes features as input and classifies the data into classes based on their features. The features are computed through a separated feature extraction process (handcrafted) and are given as an n-dimensional feature vector. The reason behind this separation of feature extraction and classification is that the data dimension is usually very larger, typically from tens of thousands to millions. Direct connection of raw data to an ANN would make the network too complex and too expensive to compute with traditional computing power. Besides, the various data dimension is also a design issue for such a combined ANN.

Nowadays, with the rapid increase of computation power, it is possible to combine both the feature extraction and classification processes into a single neural network. The idea is to integrate a feature extraction network in front of an ordinary ANN. Because the feature extraction is typically done through the convolution of local filters upon an image, an ANN with feature extraction functionality is called a *convolutional neural network* or CNN for short.

9.7.1 CNN Architecture

The architecture of a CNN can be best demonstrated using the LeNet [3] in Fig. 9.19. Basically, a CNN consists of a *convolution network* in front and a *fully connected MLP* (multilayer perceptron, an ordinary ANN) at the backend.

Because each hidden unit in the convolutional network is only connected to a local neighborhood (e.g., a clock) in the input image instead of every pixel, it is also called a *locally connected* network. In contrast, in an ordinary ANN, each element

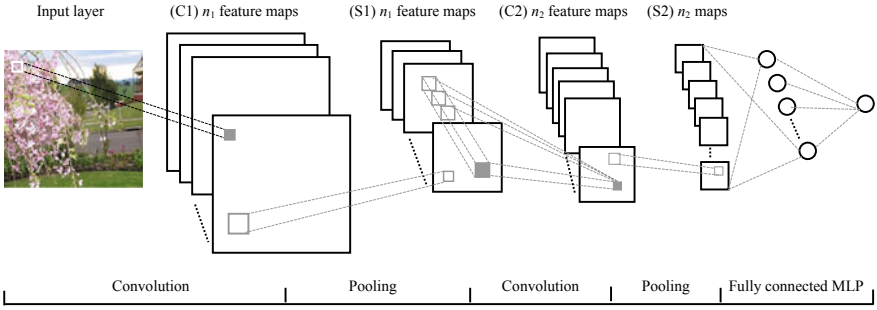


Fig. 9.19 Architecture of a CNN

of an input data is connected to each hidden unit in the network, so it is called *fully connected* network.

The convolutional network is a repeat process of *convolution* and *pooling* as shown in Fig. 9.19 [3]. Depending on the dimension of the input data, the repetitions can occur for a number of rounds. In the following, we describe the CNN architecture in detail.

9.7.2 Input Layer

- The input data are a set of training images $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$.
- Each image $\mathbf{x}$ is a C-dimensional volume $M \times M \times C$, where M is the height and width of the image and C is the number of channels.
- For the convenience of formulation, the height and width of the images are assumed to be the same.
- Typically, the input is a RGB color image $\mathbf{x} = \mathbf{x}\,[i, j, k]$ and $C = 3$.
- For a gray-level image, $C = 1$.

9.7.3 Convolution Layer 1 (C1)

In a CNN, the convolution is a high-dimensional volume convolution.

- Specifically, the convolution is done by shifting a high-dimensional *volume filter* W: $N \times N \times S$ across the image as shown in Fig. 9.20a, where N is the height and width of the windowed filter and S is the number of channels of the filter.
- S can be either the same as the number of image channels C or different.
- It can be shown that a high-dimensional volume filter consists of S number of 2D filters w with size of $N \times N$.
- In practice, the convolution is done by convoluting each channel of the high-dimensional volume filter with each channel of the input image.
- Each of these 2D filters w is meant to capture a particular type of edges, shapes, or textures from the input image.

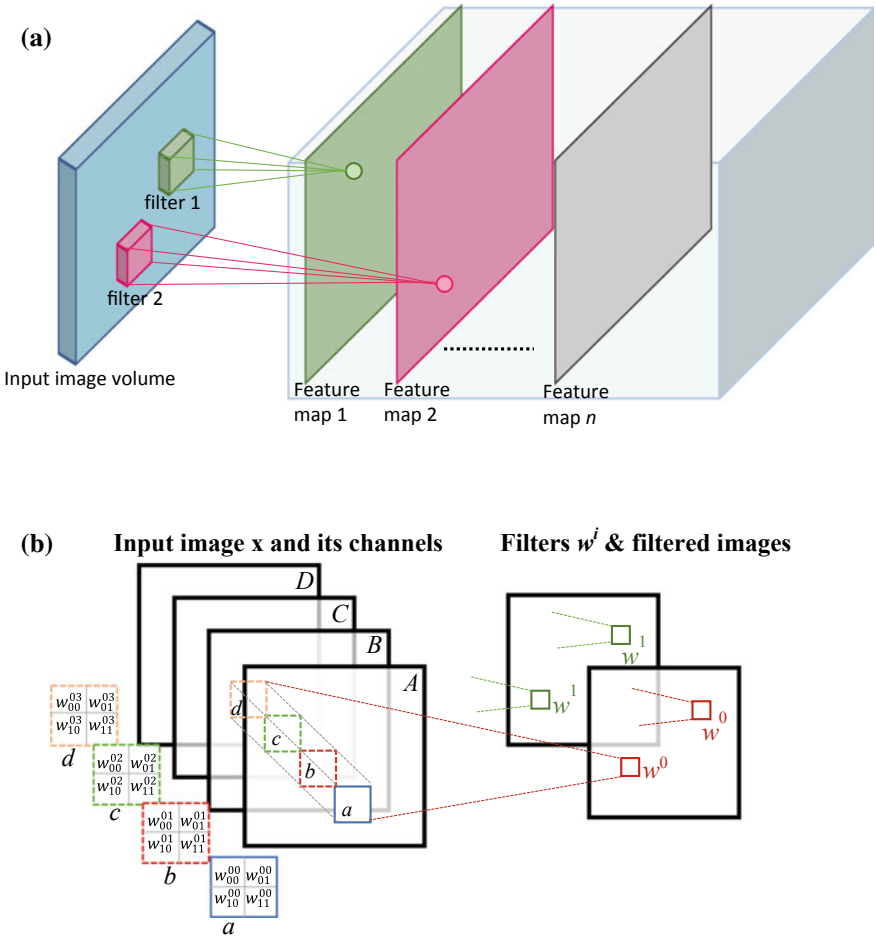


Fig. 9.20 Volume convolution. **a** Demonstration of volume filter and high-dimensional convolution; **b** demonstration of high-dimensional convolution $\mathbf{x} * w^0$ which can be done using a series of 2D convolutions. Each of the filter channels *a*, *b*, *c*, and *d* is convoluted with each of the corresponding image channels *A*, *B*, *C*, and *D*

- Figure 9.20b demonstrates how a volume convolution is done by a series of *2D convolution*.
- In Fig. 9.20b, the input data is an image **x** with four channels *A*, *B*, *C*, and *D*.
- There are two high-dimensional volume filters w^0 and w^1 on the right-hand side, each of the volume filters consists of four channels *a*, *b*, *c*, and *d*, which are shown at the bottom left of Fig. 9.20 [3].
- The convolution between **x** and w^0 ($\mathbf{x} * w^0$) is done by convoluting each of the filter channels *a*, *b*, *c*, and *d* across each of the corresponding image channels *A*, *B*, *C*, and *D*.
- The convolution of image **x** with filter w^1: $\mathbf{x} * w^1$ is done the same way.

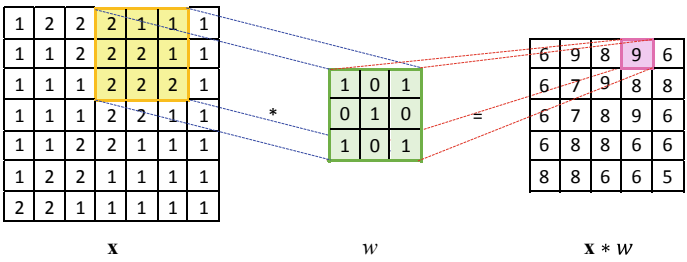


Fig. 9.21 2D convolution. An image **x** is convoluted with a filter window w and the result of the convolution is given by **x** * w at the right-hand side

- Therefore, understanding 2D convolution is the key to understand high-dimensional volume convolution.

9.7.3.1 2D Convolution

- A 2D convolution is done by sliding an $N \times N$ window w across the image **x** row by row and column by column, assuming the window slides one pixel per time and there is no padding for the moment.
- The 2D convolution of **x** * w is given by (9.24) and an example of a 2D convolution is shown in Fig. 9.21 [4].

$$X_{mn} = (\mathbf{x} * w)_{mn} = \sum_{j=0}^{N-1} \sum_{i=0}^{N-1} w[i,j] \cdot \mathbf{x}[m-i, n-j] \tag{9.24}$$

9.7.3.2 Stride and Padding

- The dimensions of the convoluted image depend on two parameters: *stride* and *padding*.
- The *stride* determines the number of pixels the filter window shifts per time and the *padding* determines if and what the input image should be padded when the filter window is at the image boundary, e.g., 0 padding.
- If the stride value is 1 and the padding is yes, the dimensions of the convoluted image are the same as the input image.
- In Fig. 9.21, the stride is 1 and there is no padding, and therefore the convoluted image loses two pixels at both ends of each row and column.

9.7.3.3 Bias

- In a CNN, the values in a filter w are regarded as the weights for the connections between the filter and the network.

- These weights are to be learned during the training of the network.
- Therefore, a bias b is added to compensate for the estimation error.

$$X'_{mn} = (\mathbf{x} * w)_{mn} + b = b + \sum_{j=0}^{N-1}\sum_{i=0}^{N-1} w[i,j] \cdot \mathbf{x}[m-i, n-j] \tag{9.25}$$

9.7.3.4 Volume Convolution in Layer C1

- Each of the S channels of the *volume filter* is first convoluted with each corresponding channel of the input image x.
- The S filtered channels are then combined to create a 2D feature map or image f_{mn}.

$$\begin{aligned} f_{mn} &= \sum_{k=1}^{S} (\mathbf{x} * w[:,:,k])_{mn} + b \\ &= b + \sum_{k=1}^{S}\sum_{j=0}^{N-1}\sum_{i=0}^{N-1} w[i,j,k] \cdot \mathbf{x}[m-i, n-j, k] \end{aligned} \tag{9.26}$$

9.7.3.5 Depth of the Feature Map Volume

- Multiple *volume filters* are used in a convolution layer to create a volume of feature maps.
- Each of the volume filters captures a particular type of image features.
- The number of volume filters R is called the *depth* of the feature map volume.
- The rth feature map is given by (9.27), $r = 0, 1, \ldots, R - 1$.

$$f^r_{mn} = b + \sum_{k=1}^{S}\sum_{j=0}^{N-1}\sum_{i=0}^{N-1} w^r[i,j,k] \cdot \mathbf{x}[m-i, n-j, k] \tag{9.27}$$

- Figure 9.22 shows how two volume filters w^0 and w^1 are used in layer C1 to create the two feature maps (light red) at the rightmost hand side [5].
- In the figure, the input image **x** is a color image with R, G, and B channels, each of the two volume filters also has three channels. The figure demonstrates the convolution of the three channels (*green*) of the first filter with a block (*yellow*) in image **x**. The convolutional output of the yellow block of image **x** is shown as the *pink* pixel in the first output image on the rightmost hand side of the figure.
- Although the input **x** is a 7 × 7 image, due to the stride value of the convolution is 2, the filtered output is just a 3 × 3 image. Therefore, in order to output a filtered image with the same size as the input image, we not only need to set the *stride* value as 1 but also need to *pad* the image with half the filter size.

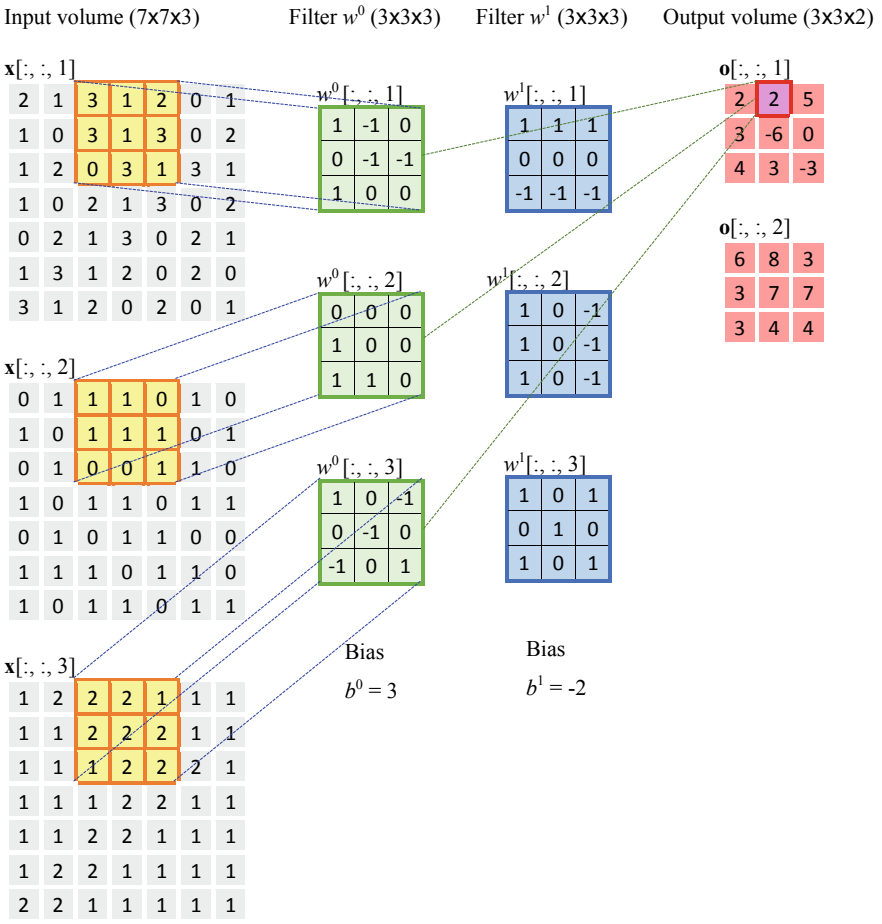


Fig. 9.22 An input image is convoluted with two volume filters w^0 and w^1. The two result feature maps are at the rightmost hand side (light red). The *stride* value is 2 and there is no padding

9.7.3.6 ReLU Activation

- The output from a volume filter or the feature map is essentially the weighted sum of the input layer.
- As in an ordinary neural network, it needs to pass an activation function.
- It has been found that in a convolution layer, the $max(0, x)$ function is more effective than a sigmoid function for the activation.
- $max(0, x)$ is basically a rectified linear function because it simply rectifies or refracts the negative half of $y = x$ to 0.
- Therefore, it is often called a *rectifier* and a node activated by the rectifier is also called a *rectified linear unit* or ReLU.

- So, we have

$$ReLU(x) = max(0, x) \tag{9.28}$$

- The output from node r of layer C1 is finally given as (9.29)

$$y_{mn}^{r} = ReLU\left(f_{mn}^{r}\right) \tag{9.29}$$

- The ReLU activation is usually implemented as a separate layer after the convolution layer.

9.7.3.7 Batch Normalization

In a CNN, learning rate or convergence speed is a major issue. Due to the convolution, the range of output values of the filters in each layer varies widely. In other words, the convolution has changed the original distribution of the input data, breaking the *independent* and *identically distributed* or i.i.d. assumption on input data. Worse still, each layer has to adapt to distribution drift from lower layers in order to revise its own weights. This makes the learning very inefficient and convergence very slow, especially for layers with *sigmoid* or *tanh* activation. This phenomenon is called the *internal covariance shift* or the change of data distribution from the input data distribution. In order to overcome this undesirable effect, a batch normalization procedure is introduced before the activation layer in an attempt to keep the mean and variance of the input data fixed, so that layers learn themselves more or less independent with each other. The basic idea is to normalize the input data of all layers in the network to have 0 mean and unit variance. In practice, the normalization is done to the input data or data to be activated dimension by dimension and batch by batch. Let us take a particular activation x (a single dimension of the input data), for example, there are m values from a mini-batch: $B = \{x_1, x_2, \ldots, x_m\}$. The algorithm of the batch normalization of x is given as following [6]:

Input: Values of x from a mini-batch $B = \{x_1, x_2, \ldots, x_m\}$
Parameters to be learned: γ, β
Output: Batch normalized values $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$$\begin{aligned}
&\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i && //mini-\text{batch mean}\\
&\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 && //mini-\text{batch variance}\\
&\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} && //sample\ normalisation\\
&BN_{\gamma,\beta}(x_i) = y_i \leftarrow \gamma \hat{x}_i && //batch\ normalisation
\end{aligned}$$

For convolution layers, the normalization is done by jointly normalizing all the activations in a mini-batch over all locations. Specifically, the pair of parameters γ and β is learnt per feature map instead of per activation [6].

By batch normalization, the values of input features to each layer are normalized into the same range, this reduces the oscillations of gradient descent when it approaches the minimum point and consequently makes it to converge faster. Another benefit of batch normalization is that it adds minor noise to each layer due to each training sample is mixed with other samples in a mini-batch, and this reduces the effect of overfitting. In practice, lower dropout rate is needed for a network with batch normalization.

9.7.4 Pooling or Subsampling Layer 1 (S1)

The feature maps' output from layer C1 usually has the same dimension as the input image. Their dimensions are too high to be connected to an ANN. Besides, the feature maps represent the finest details of the input image, these features are not as reliable. Therefore, it is tempting to downsample the feature maps so that features at a coarser level can be extracted. This is done by passing each feature map through a 2×2 subsampling function. Several types of subsampling functions can be used, such as *max*(), *average*(), L_2 *norm*, or *spectral transform* such as DWT and DCT.

For example, if the *max* function is used, the subsampling is called a *max-pooling*. Figure 9.23 demonstrates a *max-pooling* which reduces a 4×4 feature map to a 2×2 feature map.

9.7.5 Convolution Layer 2 (C2)

The outputs from the pooling layer 1 (S1) are subsampled feature maps from layer C1. New features can be computed from those feature maps by doing another round of convolution using new volume filters. The convolution procedure is the same as

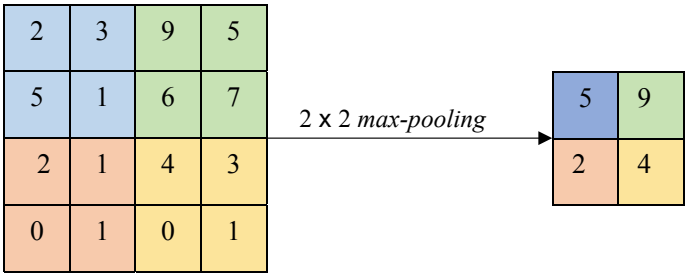


Fig. 9.23 Illustration of a max-pooling. The maximum value of each quarter block of the left image is computed as the output value of the max-pooling image at the right-hand side

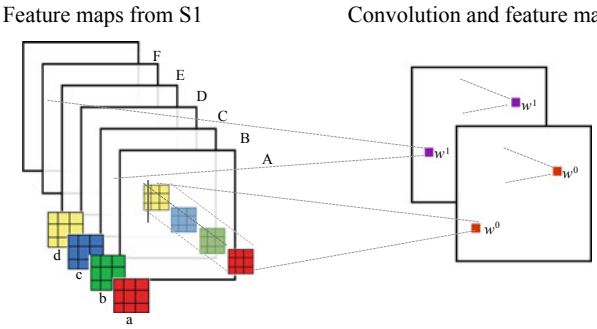


Fig. 9.24 Volume convolution in layer C2. Volume filter w^0 is convoluted with feature maps A–D while volume filter w^1 is convoluted with feature maps C–F

that in layer C1 except each volume filter in layer C2 uses different combinations of feature maps from layer S1.

For example, in Fig. 9.24, the two volume filters w^0 and w^1 are convoluted with different channels or feature maps output from layer S1. While filter w^0 is convoluted with feature maps A–D, filter w^1 is convoluted with feature maps C–F.

If the depth of S1 is R, the total number of combinations of R channels is given as follows:

$$\sum_{k=1}^{R} \binom{R}{k} = 2^R - 1 \tag{9.30}$$

where k is the number of channels in a filter in layer C2.

The convolution and pooling can be repeated for a number of rounds depending on the size of the input data. The feature maps from the final pooling layer are flattened to create a 1D feature vector, this feature vector is fed into the fully connected ANN at the backend of a CNN.

9.7.6 Hyperparameters

The performance of a CNN depends on the selection of the following hyperparameters. These parameters may be data dependent and need to be determined empirically.

- *Filter size*. The window size of the volume filter.
- *Stride*. The number of pixels per shift by the filter window at each layer.
- *Padding*. Whether padding will be used at the boundary of an input image and a feature map. What type of padding will be used, e.g., zero padding or wrap around padding.
- *Depths*. The number of channels or filters at each convolution layer.

- *Dropout-rates*. The percentage of neurons to be dropped out from each of the hidden layers at each iteration to prevent overfitting and cope with missing data.
- *Epochs*. The number of times the training algorithm will iterate over the entire training set before terminating.
- *Pooling size and function*. The size of a pooling function and the type of pooling function such as *max*(), *average*(), L_2 *norm*, etc.
- *Activation function*. The function used to generate a threshold output at each layer, such as *ReLU*(), *sigmoid*, *tanh*(), etc.
- The number of neurons in the fully connected layer of the ANN.

9.8 Implementation of CNN

To demonstrate a CNN in action, we show a high-performance CNN implementation by Oxford's Visual Geometry Group or VGGNet [7]. It has won the runner up of the 2014 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [Image-Net.org]. ImageNet is the largest hand-annotated visual dataset, and it holds image recognition competitions every year among researchers around the world. Compared with other high-performance CNN models, VGGNet is known for its simplicity because it is a *series network*.

9.8.1 CNN Architecture

The architecture of VGGNet is shown in the following list which is obtained by using Matlab code: net = vgg16; net.Layers. The 16 core layers are highlighted using bold font and are organized into five blocks: conv1—conv5. Therefore, it is often referred to as VGG16. It is a typical stacked convolution + pooling layers followed by fully connected ANN. The purpose of the softmax layer is to convert any vector of real numbers into a vector of probabilities, which correspond to the likelihoods that an input image is a member of a particular class.

01	'input'	Image Input	224 × 224 × 3 images with 'zerocenter' normalization
02	'**conv1_1**'	Convolution	64 3 × 3 × 3 convolutions with stride [1 1] and padding [1 1 1 1]
03	'relu1_1'	ReLU	ReLU
04	'**conv1_2**'	Convolution	64 3 × 3 × 64 convolutions with stride [1 1] and padding [1 1 1 1]
05	'relu1_2'	ReLU	ReLU
06	'pool1'	Max Pooling	2 × 2 max pooling with stride [2 2] and padding [0 0 0 0]
07	'**conv2_1**'	Convolution	128 3 × 3 × 64 convolutions with stride [1 1] and padding [1 1 1 1]

08	'relu2_1'	ReLU	ReLU
09	'**conv2_2**'	Convolution	128 3 × 3 × 128 convolutions with stride [1 1] and padding [1 1 1 1]
10	'relu2_2'	ReLU	ReLU
11	'pool2'	Max Pooling	2 × 2 max pooling with stride [2 2] and padding [0 0 0 0]
12	'**conv3_1**'	Convolution	256 3 × 3 × 128 convolutions with stride [1 1] and padding [1 1 1 1]
13	'relu3_1'	ReLU	ReLU
14	'**conv3_2**'	Convolution	256 3 × 3 × 256 convolutions with stride [1 1] and padding [1 1 1 1]
15	'relu3_2'	ReLU	ReLU
16	'**conv3_3**'	Convolution	256 3 × 3 × 256 convolutions with stride [1 1] and padding [1 1 1 1]
17	'relu3_3'	ReLU	ReLU
18	'pool3'	Max Pooling	2 × 2 max pooling with stride [2 2] and padding [0 0 0 0]
19	'**conv4_1**'	Convolution	512 3 × 3 × 256 convolutions with stride [1 1] and padding [1 1 1 1]
20	'relu4_1'	ReLU	ReLU
21	'**conv4_2**'	Convolution	512 3 × 3 × 512 convolutions with stride [1 1] and padding [1 1 1 1]
22	'relu4_2'	ReLU	ReLU
23	'**conv4_3**'	Convolution	512 3 × 3 × 512 convolutions with stride [1 1] and padding [1 1 1 1]
24	'relu4_3'	ReLU	ReLU
25	'pool4'	Max Pooling	2 × 2 max pooling with stride [2 2] and padding [0 0 0 0]
26	'**conv5_1**'	Convolution	512 3 × 3 × 512 convolutions with stride [1 1] and padding [1 1 1 1]
27	'relu5_1'	ReLU	ReLU
28	'**conv5_2**'	Convolution	512 3 × 3 × 512 convolutions with stride [1 1] and padding [1 1 1 1]
29	'relu5_2'	ReLU	ReLU
30	'**conv5_3**'	Convolution	512 3 × 3 × 512 convolutions with stride [1 1] and padding [1 1 1 1]
31	'relu5_3'	ReLU	ReLU
32	'pool5'	Max Pooling	2 × 2 max pooling with stride [2 2] and padding [0 0 0 0]
33	'**fc6**'	Fully Connected	4096 fully connected layer
34	'relu6'	ReLU	ReLU
35	'drop6'	Dropout	50% dropout
36	'**fc7**'	Fully Connected	4096 fully connected layer
37	'relu7'	ReLU	ReLU

38	'drop7'	Dropout	50% dropout
39	'**fc8**'	Fully Connected	1000 fully connected layer
40	'prob'	Softmax	softmax
41	'output'	Classification	crossentropyex with 'tench' and 999 other classes

VGGNet shows that the depth of the network is a critical component for good performance. The number of filters (depth) increases from 64 to 512 as it goes deeper into the network. The consecutive use of 3×3 convolutions has the effect of causing more nonlinearity as more than one ReLU functions have been applied at each stage of convolutions.

Another advantage of using consecutive convolutions is the increasing size of the receptive field. This is because two consecutive 3×3 convolutions have the effective receptive field of a single 5×5 convolution, while three-stacked 3×3 convolutions have the receptive field of a single 7×7 one [7].

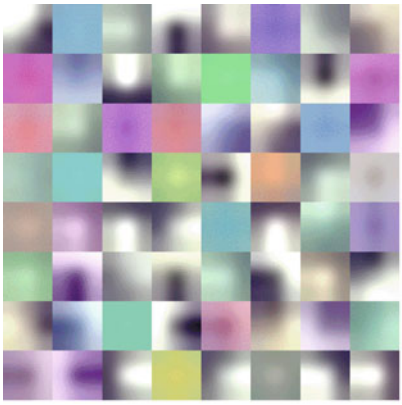
9.8.2 Filters of the Convolution Layers

To validate a CNN, it is valuable to inspect and examine the internal structure of the network. Figure 9.25 shows the first 64 pretrained filters from 6 layers of VGG16 net. It can be observed that the filters typically capture the blobs, edges, regularity, directionality, and other features of an image. Filters in early layers (layers 2 and 7) typically focus on pixels, blobs, and edges, as the network goes deeper, the filters become coarser, where low-level features are organized into shapes and parts of objects.

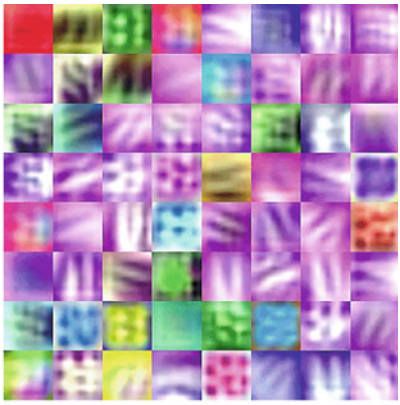
9.8.3 Filters of the Fully Connected Layers

If the convolution layers try to capture the texture and shape features from images, the fully connected layers attempt to organize the features into objects. Figure 9.26a and b shows the first 10 channels of layers "fc6" (layer 33) and "fc7" (layer 36), respectively. This phenomenon of learning objects is more obvious in the final fully connected layer where the class names are known (using Matlab code: net.Layers (end).Classes). Figure 9.26c shows 20 channels from "fc8" (layer 39) with class names as following: goldfish, tiger shark, hammerhead shark, ostrich, great gray owl, African crocodile, mud turtle, academic gown computer keyboard, cowboy boot, accordion, cowboy hat, crane (machine), crash helmet, ambulance, analog clock, balloon, dining table, dumbbell, and acoustic guitar.

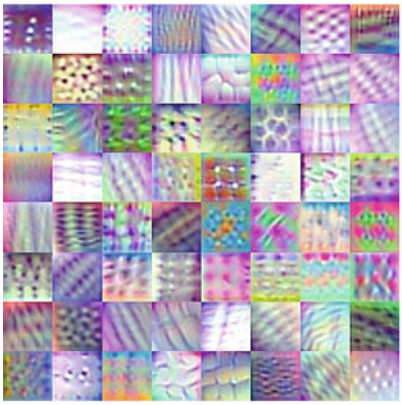
It can be observed from Fig. 9.26c that the signature patterns and shapes of these objects are well captured. More interestingly, the filters have learnt multiple copies of the same object to adapt to changes.



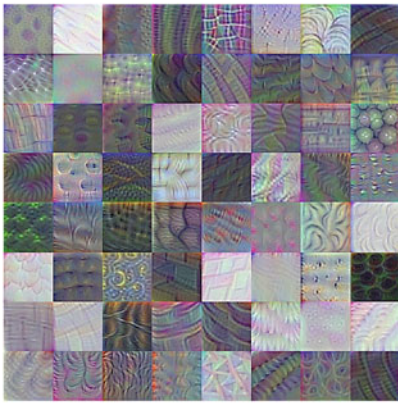
(a) Filters of layer 2



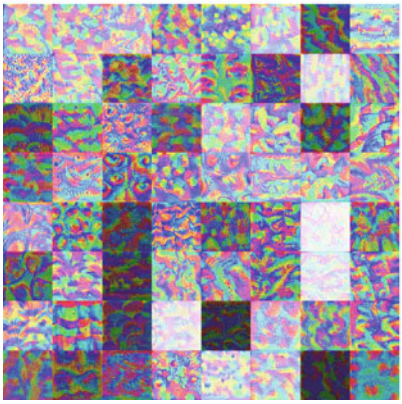
(b) Filters of layer 7



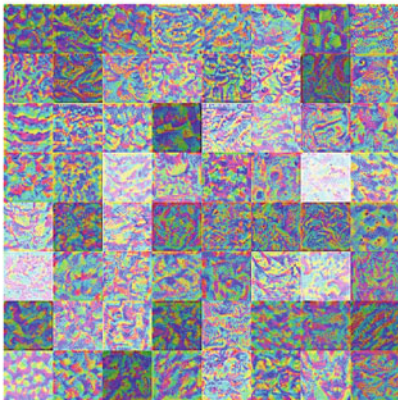
(c) Filters of layer 12



(d) Filters of layer 19

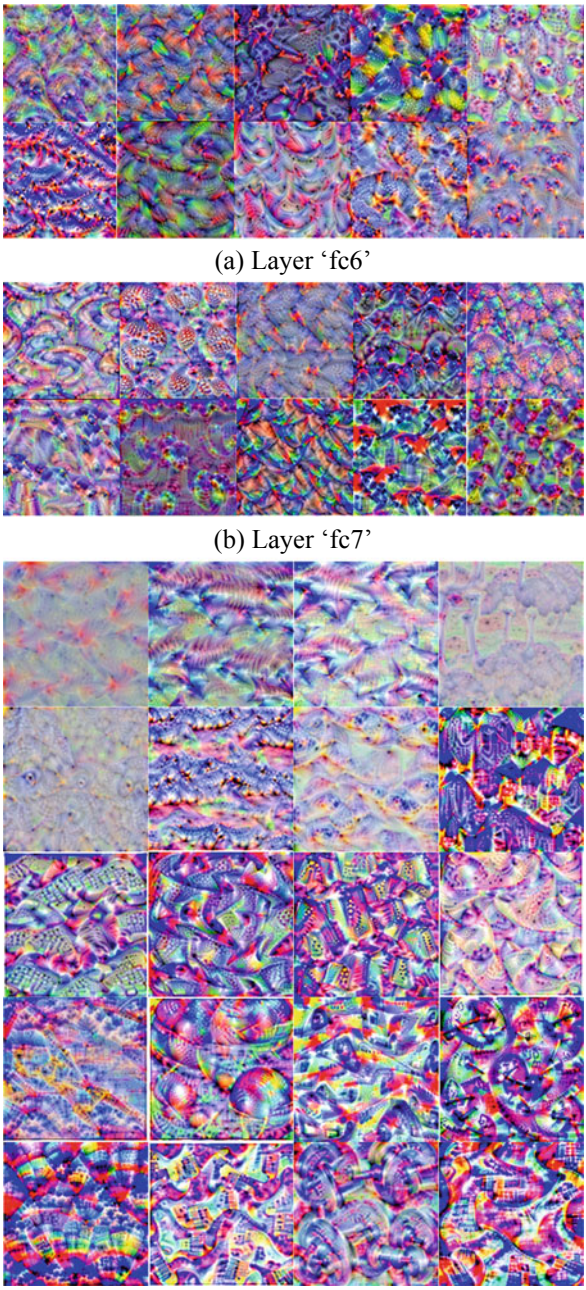


(e) Filters of layer 26



(e) Filters of layer 30

Fig. 9.25 Pretrained filters of 6 of the 13 VGG16 convolution layers



(a) Layer 'fc6'

(b) Layer 'fc7'

(c) Layer 'fc8'

Fig. 9.26 Filters of the fully connected layers

9.8.4 Feature Maps of Convolution Layers

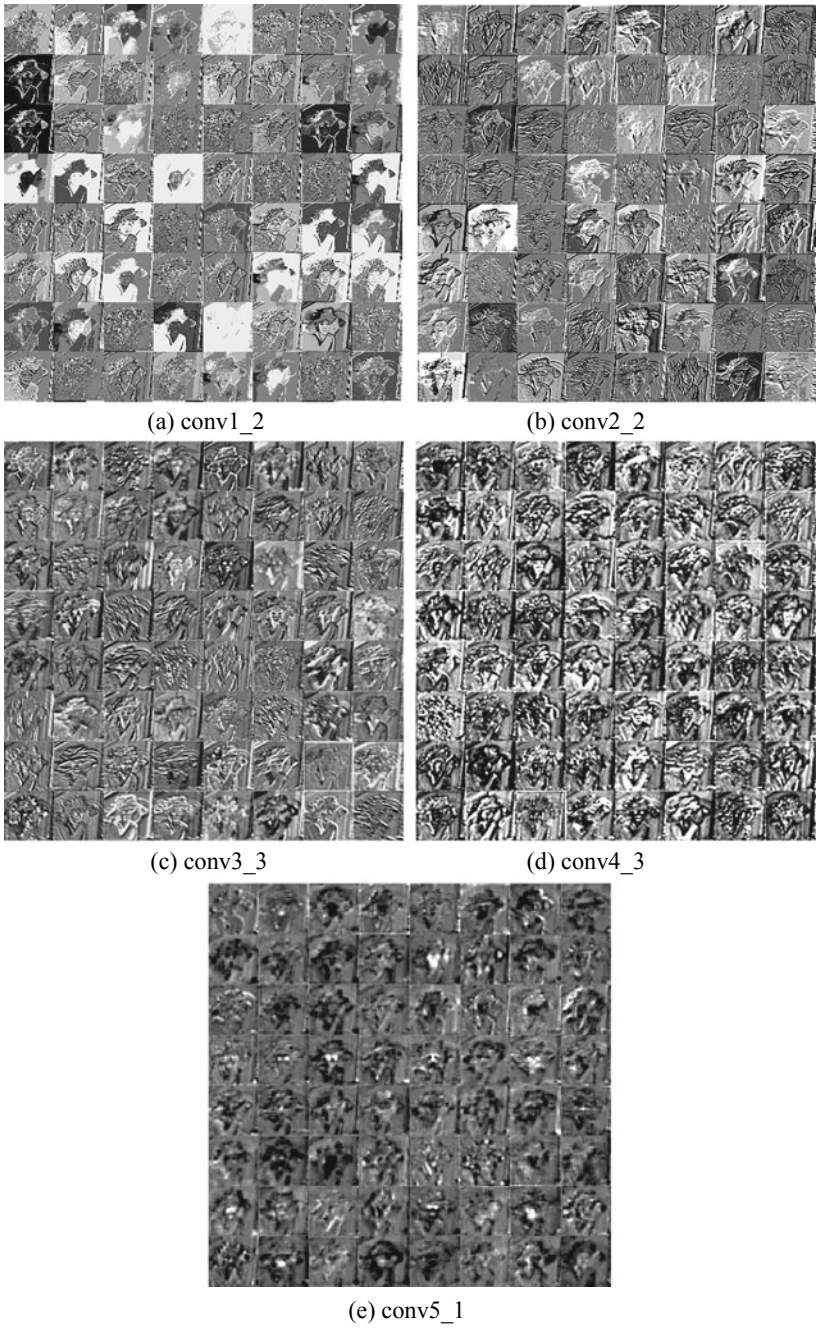
To further understand the implementation of a CNN, it is more revealing to examine the convolution process by using a real input image. A CNN is basically a combination of two components: *convolution layers* and *fully connected layers*. The convolution layers are responsible for feature extraction and the fully connected layers are responsible for the classification. The convolution component is the main powerhouse of a CNN model. Given an input image, the different filters in the convolution layers detect features such as edges, blobs, and regions, which represent eyes, ears, legs, feather, leaves, water, sand, windows, wheels, etc. The CNN does not know if they are eyes, ears, legs, etc., it learns to detect them as features by memorizing a lot of them in the input images. The fully connected layers learn how to use these features to classify the images into different classes.

One important thing to note is that due to the nature of consecutive convolution and pooling, the features leant from the CNN is evolutional or hierarchical. In other words, the CNN is a learning process from fine features to coarse features. The convolution layers learn such fine to coarse features by building on top of each other. The first layers detect edges, the next layers combine them to detect shapes, and the following layers merge shape information to infer objects such as eyes, ears, legs, etc. Figure 9.27 demonstrates this evolutional process by showing the first 64 feature maps from each of the five blocks of the lady image in Fig. 3.2: conv1_2, conv2_2, conv3_3, conv4_3, and conv5_1. It can be observed that the prominent features (hat, face) in the image are well captured by the filters. It is interesting to find that each filter captures different aspects of the image such as the surface and outline of the hat, the face, eyes, cloth, hand, background, etc. It can also be seen from the figure that the features from the first block of layers (conv1_1) are sparse and show the fine details/edges of the image, and as the network goes deeper, the features become coarser and coarser due to pooling, until the final block of layers where only the most prominent features (e.g., eyes, mouth) in the input image survive.

Figure 9.28 uses heat maps of the channels from each of the blocks to demonstrate how the prominent features of a face image have been tracked by the network. It can be seen that the eyes of the face are well tracked by the network and as the network goes deeper, the face pattern becomes coarser and coarser until it is completely blurred.

Although conventional feature extraction methods can also extract similar kind of coarse features for classification, a CNN model can combine many types of such kind of coarse features to form a set of more powerful features which lead to more accurate classification.

Overfitting is a common problem on image classification because usually there are too few training samples, resulting in a model with poor generalization



(a) conv1_2 (b) conv2_2

(c) conv3_3 (d) conv4_3

(e) conv5_1

Fig. 9.27 Feature maps from different convolution layers of VGG16

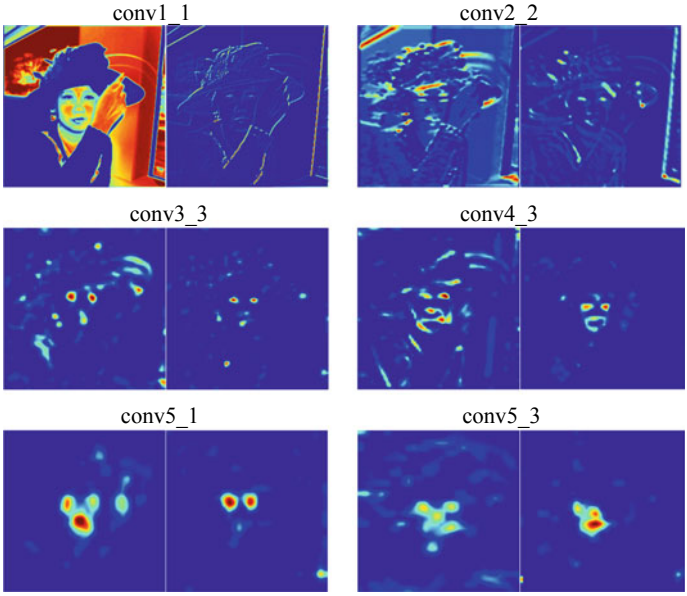


Fig. 9.28 Channels from each of the five blocks of VGG16 net

performance. One solution to overfitting is to use *data augmentation*. Data augmentation is a method to generate more training data from the current training set. It is an artificial way to boost the size of the training set, reducing overfitting.

Data augmentation is typically done by data transformations and processing, such as rotation, shifting, resizing, adding noise, contrast change, etc. It should be noted that data augmentation is only performed on the training data, not on the validation or test set.

9.8.5 Matlab Implementation

Matlab's Deep Learning Toolbox™ has a number of built-in networks which are pretrained on ImageNet, including ResNet-50, AlexNet, GoogleNet, VGG-16, and VGG-19. The following is a code scheme of using VGG-16 for image classification [8]. The code provides a step-by-step implementation of a CNN. Training images need to be first categorized and organized into subfolders, and the name of each subfolder represents the label of the image category, e.g., *bird*, *people*, *tiger*, etc.

```
% Load the Pretrained VGG-16 network

net = vgg16();

net.Layers;  %inspect the network architecture

% Extract and Display Feature Maps

% Extract the filters for convolutional layer 1: conv1_1
filter1 = net.Layers(2).Weights;
filter1 = mat2gray(filter1);
filter1 = imresize(filter1,5);
figure
montage(filter1)
title('First convolutional layer filters')

% Prepare Training and Testing Image Sets

% Loading images into an imageDataStore Object, e.g., rootFolder=c:\myImages,

subFolder=cnnImages, myFolder='c:\myImages\cnnImages'

myFolder = fullfile('rootFolder', 'subFolder');

categories = {'cat1', 'cat2',…, 'catn'};

imstore = imageDatastore(fullfile(myFolder, categories), 'LabelSource',

'foldernames');

% Split the dataset into training set (30%) and testing set (70%)
[trainingSet, testingSet] = splitEachLabel(imstore, 0.3, 'randomize');

% Normalise Dataset images to required size and RGB format
imSize = net.Layers(1).InputSize;
normTrainingSet = augmentedImageDatastore(imSize, trainingSet,
'ColorPreprocessing', 'gray2rgb');
normTestingSet = augmentedImageDatastore(imSize, testingSet,
'ColorPreprocessing', 'gray2rgb');

% Extract features from the last fully connected layer
fcFeature = 'fc8';
trainingFeatures = activations(net, normTrainingSet, fcFeature, ...
       'MiniBatchSize', 32, 'OutputAs', 'columns');

% Train a Multiclass SVM Classifier Using the Extracted Features

% Get training labels from the trainingSet
trainingLabels = trainingSet.Labels;
classifier = fitcecoc(trainingFeatures, trainingLabels, ...
 'Learners', 'Linear', 'Coding', 'onevsall', 'ObservationsIn', 'columns');

% Test the SVM Classifier on New Images

newImage = imread(fullfile('rootFolder', 'subFolder', 'imageName'));

% Normalise the new images.
normImage = augmentedImageDatastore(imSize, newImage, 'ColorPreprocessing',
'gray2rgb');

% Extract image features using the CNN
imFeatures = activations(net, normImage, fcFeature, 'OutputAs', 'columns');

% Make a prediction using the classifier
label = predict(classifier, imFeatures, 'ObservationsIn', 'columns')
```