Chapter 6
Artificial Neural Networks and Backpropagation

6.1 Introduction

Inspired by the biological neural network, here we discuss its mathematical abstraction known as the artificial neural network (ANN). Although efforts have been made to model all aspects of the biological neuron using a mathematical model, all of them may not be necessary: rather, there are some key aspects that should not be neglected when modeling a neuron. This includes the weight adaptation and the nonlinearity. In fact, without them, we cannot expect any learning behavior.

In this chapter, we first describe a mathematical model for a single neuron, and explain its multilayer realization using a feedforward neural network. We then discuss standard methods of updating weight, often referred to as neural network training. One of the most important parts of neural network training is gradient computation, so the rest of this chapter discusses the main weight update techniques known as *backpropagation* in detail.

6.2 Artificial Neural Networks

6.2.1 Notation

Since the mathematical description of an artificial neural network involves several indices for neuron, layers, training sample, etc., here we would like to summarize them for reference so that they can be used in the rest of the chapter.

First, each training data set is usually represented as bold face lower case letters with the index n: for example, the following are used to indicate the n-th training-data-related variables:

$$\boldsymbol{x}_n, \boldsymbol{y}_n, \{\boldsymbol{x}_n, \boldsymbol{y}_n\}_{n=1}^N, \boldsymbol{o}_n, \boldsymbol{g}_n.$$

J. C. Ye, *Geometry of Deep Learning*, Mathematics in Industry 37,
https://doi.org/10.1007/978-981-16-6046-7_6

Second, with a slight abuse of notation, the subscript i and j for the light face lowercase letters denotes the i-th and j-th element of a vector: for example, o_i is the i-th element of the vector $\boldsymbol{o} \in \mathbb{R}^d$:

$$o_i = [\boldsymbol{o}]_i, \quad \text{or} \quad \boldsymbol{o} = \begin{bmatrix} o_1 & \cdots & o_d \end{bmatrix}^\top.$$

Similarly, the double index ij indicates the (i, j) element of a matrix: for example, w_{ij} is the (i, j)-th element of a matrix $\boldsymbol{W} \in \mathbb{R}^{p \times q}$:

$$w_{ij} = [\boldsymbol{W}]_{i,j} \quad \text{or} \quad \boldsymbol{W} = \begin{bmatrix} w_{11} & \cdots & w_{1q} \\ \vdots & \ddots & \vdots \\ w_{p1} & \cdots & w_{pq} \end{bmatrix}.$$

This index notation is often used to refer to the i-th or j-th neuron in each layer of a neural network. To avoid potential confusion, if we refer to the i-th element of the n-th training data vector $\boldsymbol{x}_n$ is referred to as $(\boldsymbol{x}_n)_i$. Next, to denote the l-th layer, the following superscript notation is used:

$$\boldsymbol{g}^{(l)}, \boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}, d^{(l)}.$$

Accordingly, by combining the training index n, for example $\boldsymbol{g}_n^{(l)}$ refers to the l-th layer $\boldsymbol{g}$ vector for the n-th training data. Finally, the t-th update using an optimizer such as the stochastic gradient method can be denoted by $[t]$: for example,

$$\boldsymbol{\Theta}[t], \boldsymbol{V}[t]$$

refer to the t-th update of the parameter map $\boldsymbol{\Theta}$ and $\boldsymbol{V}$, respectively.

6.2.2 Modeling a Single Neuron

Consider a typical biological neuron in Fig. 6.1 and its mathematical diagram in Fig. 6.2. Let $o_j, j = 1, \cdots, d$ denote the presynaptic potential from the j-th dendric synapse. For mathematical simplicity, we assume that the potential occurs synchronously, and arrives simultaneously at the axon hillock. At the axon hillock, they are summed together, and fires an action potential if the summed signal is greater than the specific threshold value. This process can be mathematically modeled as

$$net_i = \sigma\left(\sum_{j=1}^{d} w_{ij} o_j + b_i\right), \tag{6.1}$$

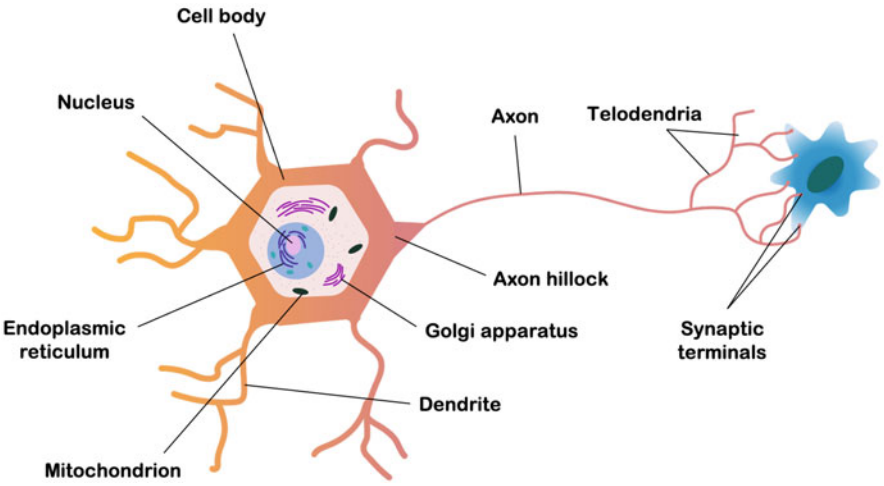

Fig. 6.1 Anatomy of neurons

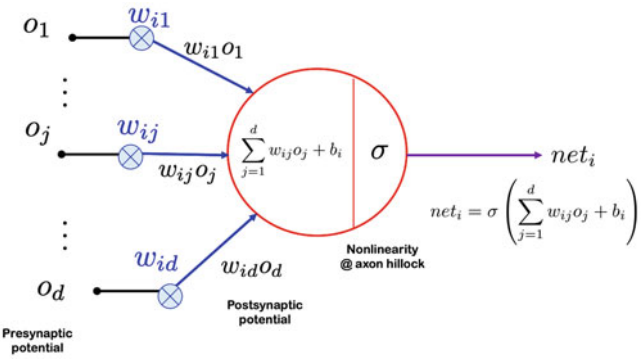

Fig. 6.2 A mathematical model of a single neuron

where net_i denotes the action potential arriving at the i-th synaptic terminal of the telodendria, and b_i is the bias term for the nonlinearity $\sigma(\cdot)$ at the axon hillock. Note that the w_{ij} is the weight parameter determined by the synaptic plasticity, and the positive values imply that $w_{ij}o_j$ are the excitatory postsynaptic potentials (EPSPs), whereas the negative weights correspond to the inhibitory postsynaptic potentials (IPSPs).

In artificial neural networks (ANNs), the nonlinearity $\sigma(\cdot)$ in (6.1) is modeled in various ways as shown in Fig. 6.3. This nonlinearity is often called the *activation function.* Nonlinearity may be perhaps the most important feature of neural networks, since learning and adaptation never happen without nonlinearity. The mathematical proof of this argument is somewhat complicated, so the discussion will be deferred to later.

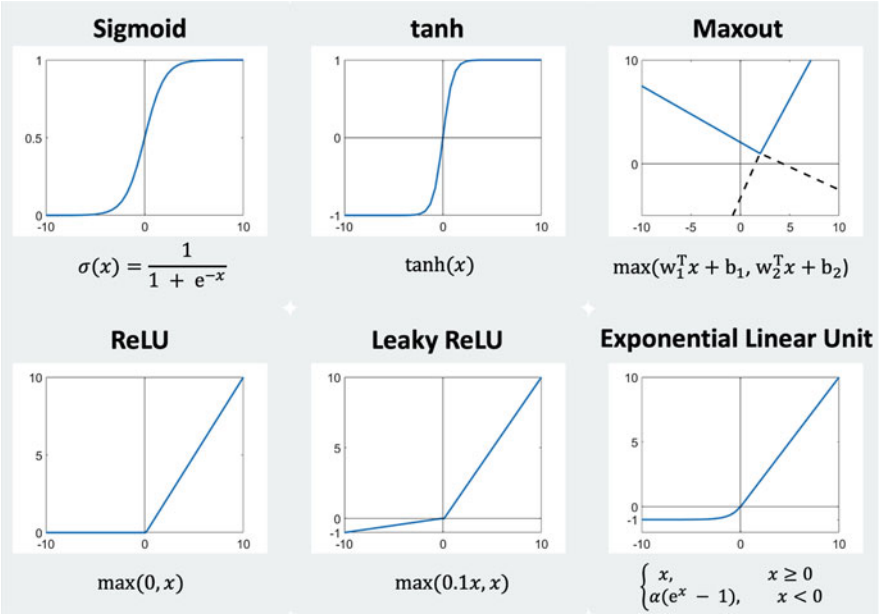

Fig. 6.3 Various forms of activation functions

Among the various forms of the activation functions, one of the most successful ones in modern deep learning is the rectified linear unit (ReLU), which is defined as [24]

$$\sigma(x) = \mathrm{ReLU}(x) := \max\{0, x\}. \tag{6.2}$$

The ReLU activation function is called *active* when the output is nonzero. It is believed that the non-vanishing gradient in the positive range contributed to the success of modern deep learning. Specifically, we have

$$\frac{\partial \mathrm{ReLU}(x)}{\partial c} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}, \tag{6.3}$$

which shows that the gradient is always 1 whenever the ReLU is active. Note that we set the gradient 0 at $x = 0$ by convention, since the ReLU is not differentiable at $x = 0$.

In evaluating the activation function $\sigma(x)$, the gain function, which refers to the input/output ratio, is also useful:

$$\gamma(x) := \frac{\sigma(x)}{x}, \quad x \neq 0. \tag{6.4}$$

For example, the ReLU satisfies the following important property:

$$\gamma(x) = \frac{\partial \sigma(x)}{\partial x} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}, \tag{6.5}$$

which will be used later in analyzing the backpropagation algorithm.

There is an additional advantage of using the ReLU compared to other nonlinearities. As will be explained in detail later, the ReLU divides the input and feature space into two disjoint sets, i.e. active and inactive areas, resulting in a piecewise linear approximation of a nonlinear mapping onto the partitioned geometry. Accordingly, a neural network within each partition can be viewed as locally linear, even though the overall map is highly nonlinear. This is the geometric picture of a deep neural network that we would like to highlight for readers in this book.

6.2.3 *Feedforward Multilayer ANN*

Biological neural networks are composed of multiple neurons that are connected to each other. This connection can have complicated topology, such as recurrent connection, asynchronous connection, inter-neurons, etc.

One of the most simple forms of the neural network connection is the multi-layer feedforward neural network as shown in Figs. 6.4 and 6.8. Specifically, let $o_j^{(l-1)}$ denote the j-th output of the $(l-1)$-th layer neuron, which is given as the j-th dendrite presynaptic potential input for the l-th layer neuron, and $w_{ij}^{(l)}$ corresponds to the synaptic weights at the l-th layer. Then, by extending the model in (6.1) we have

$$o_i^{(l)} = \sigma\left(\sum_{j=1}^{d^{(l)}} w_{ij}^{(l)} o_j^{(l-1)} + b_i^{(l)}\right), \tag{6.6}$$

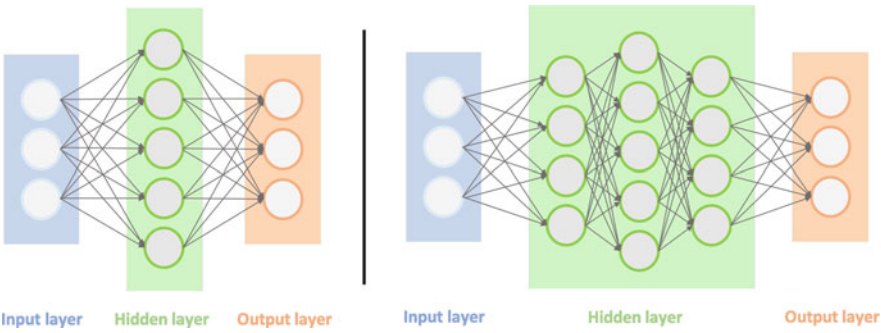

Fig. 6.4 Examples of multilayer feedforward neural networks

for $i = 1, \cdots, d^{(l)}$, where $d^{(l)}$ denotes the number of dendrites of the l-th layer neuron. This can be represented in a matrix form

$$\boldsymbol{o}^{(l)} = \boldsymbol{\sigma}\left(\boldsymbol{W}^{(l)}\boldsymbol{o}^{(l-1)} + \boldsymbol{b}^{(l)}\right), \tag{6.7}$$

where $\boldsymbol{W}^{(l)} \in \mathbb{R}^{d^{(l)} \times d^{(l-1)}}$ is the weight matrix whose (i, j) elements are given by $w_{ij}^{(l)}$, $\boldsymbol{\sigma}(\cdot)$ denotes the nonlinearity $\sigma(\cdot)$ applied for each elements of the vector, and

$$\boldsymbol{o}^{(l)} = \begin{bmatrix} o_1^{(l)} & \cdots & o_{d^{(l)}}^{(l)} \end{bmatrix}^\top \in \mathbb{R}^{d^{(l)}}, \tag{6.8}$$

$$\boldsymbol{b}^{(l)} = \begin{bmatrix} b_1^{(l)} & \cdots & b_{d^{(l)}}^{(l)} \end{bmatrix}^\top \in \mathbb{R}^{d^{(l)}}. \tag{6.9}$$

Another way to simplify the multilayer representation is using the hidden nodes from linear layers in between. Specifically, an L-layer feedforward neural network can be represented recursively using the hidden node $\boldsymbol{g}^{(l)}$ by

$$\boldsymbol{o}^{(l)} = \boldsymbol{\sigma}(\boldsymbol{g}^{(l)}), \quad \boldsymbol{g}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{o}^{(l-1)} + \boldsymbol{b}^{(l)}, \tag{6.10}$$

for $l = 1, \cdots, L$.

6.3 Artificial Neural Network Training

6.3.1 Problem Formulation

For given training data $\{\boldsymbol{x}_n, \boldsymbol{y}_n\}_{n=1}^N$, a neural network training problem can be then formulated as follows:

$$\hat{\boldsymbol{\Theta}} = \arg\min_{\boldsymbol{\Theta}} c(\boldsymbol{\Theta}), \tag{6.11}$$

where the cost function is given by

$$c(\boldsymbol{\Theta}) := \sum_{n=1}^{N} \ell\left(\boldsymbol{y}_n, \boldsymbol{f}_{\boldsymbol{\Theta}}(\boldsymbol{x}_n)\right). \tag{6.12}$$

Here, $\ell(\cdot, \cdot)$ denotes a loss function, and $\boldsymbol{f}_{\boldsymbol{\Theta}}(\boldsymbol{x}_n)$ is a regression function with the input $\boldsymbol{x}_n$, which is parameterized by the parameter set $\boldsymbol{\Theta}$.

For the case of an L-layer feedforward neural network, the regression function $\boldsymbol{f}_\Theta(\boldsymbol{x}_n)$ in (6.12) can be represented by

$$\boldsymbol{f}_\Theta(\boldsymbol{x}_n) := \left(\boldsymbol{\sigma} \circ \boldsymbol{g}^{(L)} \circ \boldsymbol{\sigma} \circ \boldsymbol{g}^{(L-1)} \cdots \circ \boldsymbol{g}^{(1)}\right)(\boldsymbol{x}_n), \tag{6.13}$$

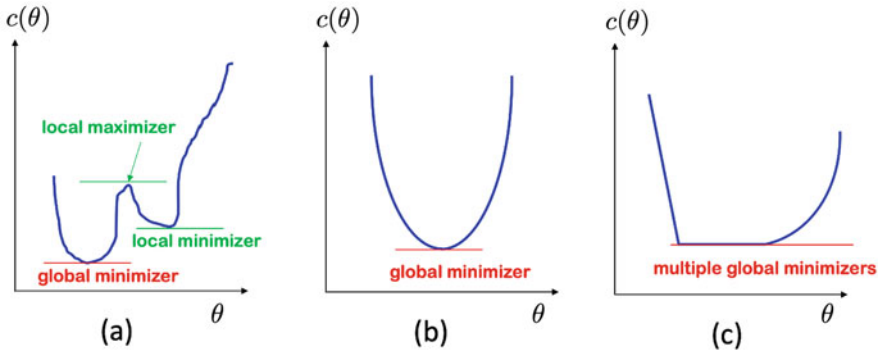

Fig. 6.5 Examples of cost functions for a 1-D optimization problem: (**a**) both local and global minimizers exists, (**b**) only a single global minimizer exist, (**c**) multiple global minimizers exist

where the parameter set $\boldsymbol{\Theta}$ is composed of the synaptic weight and bias for each layer:

$$\boldsymbol{\Theta} = \begin{bmatrix} \boldsymbol{W}^{(1)}, & \boldsymbol{b}^{(1)} \\ \vdots & \vdots \\ \boldsymbol{W}^{(L)}, & \boldsymbol{b}^{(L)} \end{bmatrix}. \tag{6.14}$$

As discussed before for kernel machines in Chap. 4, the formulation in (6.11) is so general that it covers classification, regression, etc., by simply changing the loss function (for example, l_2 loss for the regression, and the hinge loss for the classification). Unfortunately, in contrast to the kernel machines, one of the main difficulties in the neural network training is that the cost function $c(\boldsymbol{\Theta})$ is not convex, and indeed there exist many local minimizers (see Fig. 6.5). Therefore, the neural network training critically depends on the choice of optimization algorithm, initialization, step size, etc.

6.3.2 Optimizers

In view of the parameterized neural network in (6.13), the key question is how the minimizers for the optimization problem (6.11) can be found. As already mentioned, the main technical challenge of this minimization problem is that there are many local minimizers, as shown in Fig. 6.5a. Another tricky issue is that sometimes there are many global minimizers, as shown in Fig. 6.5c. Although all the global minimizers can be equally good in the training phase, each global minimizer may have different generalization performance in the test phase. This issue is important and will be discussed later. Furthermore, different global minimizers can be achieved depending on the specific choice of an optimizer, which is often called

the *implicit bias* or *inductive bias* of an optimization algorithm. This topic will also be discussed later.

One of the most important observations in designing optimization algorithms is that the following first-order necessary condition (FONC) holds at local minimizers.

Lemma 6.1 *Let $c : \mathbb{R}^p \mapsto \mathbb{R}$ be a differentiable function. If $\boldsymbol{\Theta}^*$ is a local minimizer, then*

$$\left.\frac{\partial c}{\partial \boldsymbol{\Theta}}\right|_{\boldsymbol{\Theta}=\boldsymbol{\Theta}^*} = \mathbf{0}. \tag{6.15}$$

Indeed, various optimization algorithms exploit the FONC, and the main difference between them is the way they avoid the local minimum and provide fast convergence. In the following, we start with the discussion of the classical gradient descent method and its stochastic extension called the *stochastic gradient descent (SGD)*, after which various improvements will be discussed.

6.3.2.1 Gradient Descent

For the given training data $\{\boldsymbol{x}_n, \boldsymbol{y}_n\}_{n=1}^N$, the gradient of the cost function in (6.12) is given by

$$\begin{aligned}\frac{\partial c}{\partial \boldsymbol{\Theta}}(\boldsymbol{\Theta}) &= \frac{\partial \left(\sum_{n=1}^N \ell\left(\boldsymbol{y}_n, \boldsymbol{f}_{\boldsymbol{\Theta}}(\boldsymbol{x}_n)\right)\right)}{\partial \boldsymbol{\Theta}} \\ &= \sum_{n=1}^N \frac{\partial \ell}{\partial \boldsymbol{\Theta}}\left(\boldsymbol{y}_n, \boldsymbol{f}_{\boldsymbol{\Theta}}(\boldsymbol{x}_n)\right),\end{aligned} \tag{6.16}$$

which is equal to the sum of the gradient at each of the training data. Since the gradient is the steep direction for the increasing cost function, the steep descent algorithm is to update the parameter in its opposite direction:

$$\begin{aligned}\boldsymbol{\Theta}[t+1] &= \boldsymbol{\Theta}[t] - \eta \left.\frac{\partial c}{\partial \boldsymbol{\Theta}}(\boldsymbol{\Theta})\right|_{\boldsymbol{\Theta}=\boldsymbol{\Theta}[t]} \\ &= \boldsymbol{\Theta}[t] - \eta \sum_{n=1}^N \left.\frac{\partial \ell}{\partial \boldsymbol{\Theta}}\left(\boldsymbol{y}_n, \boldsymbol{f}_{\boldsymbol{\Theta}}(\boldsymbol{x}_n)\right)\right|_{\boldsymbol{\Theta}=\boldsymbol{\Theta}[t]},\end{aligned} \tag{6.17}$$

where $\eta > 0$ denotes the step size and $\boldsymbol{\Theta}[t]$ is the t-th update of the parameter $\boldsymbol{\Theta}$. Figure 6.6a illustrates why gradient descent is a good way to minimize the cost for the convex optimization problem. As the gradient of the cost points toward the uphill direction of the cost, the parameter update should be in its negative direction.

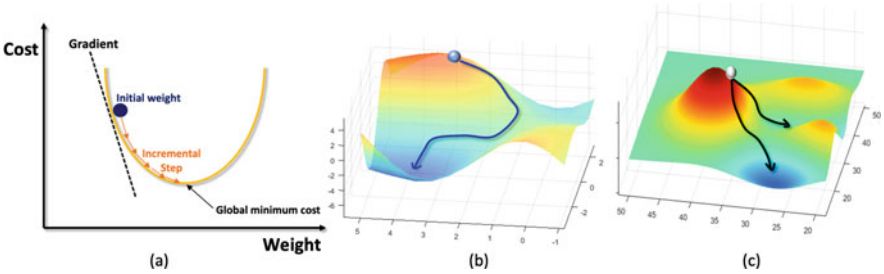

Fig. 6.6 Steepest gradient descent example: (**a**) convex cases, where steepest descent succeeds, (**b**) non-convex case, where the steepest descent cannot go uphill, (**c**) steepest gradient leads to different local minimizers depending on the initialization

After a small step, a new gradient is computed and a new search direction is found. By iterating the procedure, we can achieve the global minimum.

One of the downsides of the gradient descent method is that when the gradient becomes zero at a local minimizers at t^*, the update equation in (6.17) make the iteration stuck in the local minimizers, i.e.:

$$\boldsymbol{\Theta}[t+1] = \boldsymbol{\Theta}[t], \quad t \geq t^*. \tag{6.18}$$

For example, Fig. 6.6b,c show the potential limitation of the gradient descent. For the case of Fig. 6.6b, during the path toward the global minimum, there exists uphill directions, which cannot be overcome by the gradient methods. On the other hand, Fig. 6.6c shows that depending on the initialization, different local minimizers can be found by the gradient descent due to the different intermediate path. In fact, the situations in Fig. 6.6b,c are more likely situations in neural network training, since the optimization problem is highly non-convex due to the cascaded connection of nonlinearities. In addition, despite using the same initialization, the optimizer can converge to a completely different solution depending on the step size or certain optimization algorithms. In fact, algorithmic bias is a major research topic in modern deep learning, often referred to as *inductive bias*.

This can be another reason why neural network training is difficult and depends heavily on who is training the model. For example, even if multiple students are given the exact same training set, network architecture, GPU, etc., it is usually observed that some students are successfully training the neural network and others are not. The main reason for such a difference is usually due to their commitment and self-confidence, which leads to different optimization algorithms with different inductive biases. Successful students usually try different initializations, optimizers, different learning rates, etc. until the model works, while unsuccessful students usually stick to the parameters all the time without trying to carefully change them. Instead, they often claim that the failure is not their fault, but because of the wrong model they started with. If the training problem were convex, then regardless of the inductive bias they have in training, all students could be successful. Unfortunately,

neural network training is highly non-convex, so it is highly dependent on the student's inductive bias. The good news is that once students learn how to make a model work, the intuition they gain from such experiences usually works for training more complicated neural networks.

Indeed, advances in algorithms to optimize deep neural networks can be viewed as overcoming operator dependency. The following describes the various methods of systematically reducing the operator-dependent inductive bias for training neural networks, although the same problem still exists, albeit in a reduced manner, due to the non-convexity of the problems.

6.3.2.2 Stochastic Gradient Descent (SGD) Method

We say that the update equations in (6.17) are based on full gradients, since at each iteration we need to compute the gradient with respect to the whole data set. However, if n is large, computational cost for the gradient calculation is quite heavy. Moreover, by using the full gradient, it is difficult to avoid the local minimizer, since the gradient descent direction is always toward the lower cost value.

To address the problem, the SGD algorithm uses an easily computable estimate of the gradient using a small subset of training data. Although it is a bit noisy, this noisy gradient can even be helpful in avoiding local minimizers. For example, let $I[t] \subset \{1, \cdots, N\}$ denote a random subset of the index set $\{1, \cdots, N\}$ at the t-th update. Then, our estimate of the full gradient at the t-th iteration is given by

$$\frac{\partial c}{\partial \boldsymbol{\Theta}}(\boldsymbol{\Theta})\bigg|_{\boldsymbol{\Theta}=\boldsymbol{\Theta}[t]} \simeq \frac{N}{|I[t]|} \sum_{i \in I[t]} \frac{\partial \ell}{\partial \boldsymbol{\Theta}} \left(\boldsymbol{y}_n, \boldsymbol{f}_{\boldsymbol{\Theta}}(\boldsymbol{x}_n)\right)\bigg|_{\boldsymbol{\Theta}=\boldsymbol{\Theta}[t]}, \tag{6.19}$$

where $|I[t]|$ denotes the number of elements in $I[t]$. As the SGD utilizes a small random subset of the original training data set (i.e. $|I[t]| \ll N$) in calculating the gradient, the computational complexity for each update is much smaller than the original gradient descent method. Moreover, it is not exactly the same as the true gradient direction so that the resulting noise can provide a means to escape from the local minimizers.

6.3.2.3 Momentum Method

Another way to overcome the local minimum is to take into account the previous updates as additional terms to avoid getting stuck in local minima. Specifically, a desirable update equation may be written as

$$\boldsymbol{\Theta}[t+1] = \boldsymbol{\Theta}[t] - \eta \sum_{s=1}^{t} \beta^{t-s} \frac{\partial c}{\partial \boldsymbol{\Theta}}(\boldsymbol{\Theta}[s]) \tag{6.20}$$

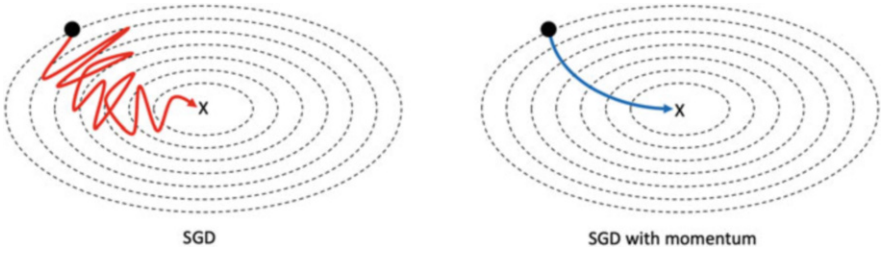

Fig. 6.7 Example trajectory of update in (**a**) stochastic gradient, (**b**) SGD with momentum method

for an appropriate forgetting factor $0 < \beta < 1$. This implies that the contribution from the past gradient is gradually reduced in calculating the current update direction. However, the main limitation of using (6.20) is that all the history of the past gradients should be saved, which requires huge GPU memory. Instead, the following recursive formulation is mostly used which provide the equivalent representation:

$$\begin{aligned} \boldsymbol{V}[t] &= \beta(\boldsymbol{\Theta}[t] - \boldsymbol{\Theta}[t-1]) - \eta \frac{\partial c}{\partial \boldsymbol{\Theta}}(\boldsymbol{\Theta}[t]), \\ \boldsymbol{\Theta}[t+1] &= \boldsymbol{\Theta}[t] + \boldsymbol{V}[t]. \end{aligned} \tag{6.21}$$

This type of method is called the *momentum method*, and is particularly useful when it is combined with the SGD. The example update trajectory of the SGD with momentum is shown in Fig. 6.7b. Compared to the fluctuating path, the momentum method provides a smoothed solution path thanks to the averaging effects from the past gradient, which results in fast convergence.

6.3.2.4 Other Variations

In neural networks, several other variants of the optimizers are often used, among which ADAGrad [25], RMSprop [26], and Adam [27] are most popular. The main ideas of these variants is that instead of using the fixed step size η for all elements of the gradient, an element-wise adaptive step size is used. For example, for the case of the steepest descent in (6.17), we use the following update equation:

$$\boldsymbol{\Theta}[t+1] = \boldsymbol{\Theta}[t] - \boldsymbol{\Upsilon}[t] \odot \frac{\partial c}{\partial \boldsymbol{\Theta}}(\boldsymbol{\Theta}[t]), \tag{6.22}$$

where $\boldsymbol{\Upsilon}[t]$ is a matrix with the step size and $\odot$ is the element-wise multiplication. In fact, the main difference in these algorithms is how to update the matrix $\boldsymbol{\Upsilon}[t]$ at each iteration. For more details for specific update rules, see the original papers [25–27].

6.4 The Backpropagation Algorithm

In the previous section, various optimization algorithms for neural network training were discussed based on the assumption that the gradient $\frac{\partial c}{\partial \boldsymbol{\Theta}}(\boldsymbol{\Theta}[t])$ is computed. However, given the complicated nonlinear nature of the feedforward neural network, the computation of the gradient is not trivial.

In machine learning, backpropagation (backprop, or BP) [28] is a standard way of calculating the gradient in training feedforward neural networks, by providing an explicit and computationally efficient way of computing the gradient. The term backpropagation and its general use in neural networks were originally derived in Rumelhart, Hinton and Williams [28]. Their main idea is that although the multilayer neural network is composed of complicated connections of neurons with a large number of unknown weights, the recursive structure of the multilayer neural network in (6.10) lends itself to computationally efficient optimization methods.

6.4.1 Derivation of the Backpropagation Algorithm

The following lemma, which was previously introduced in Chap. 1, is useful in deriving the BP algorithm:

Lemma 6.2 *Let $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ and $\boldsymbol{x} \in \mathbb{R}^n$. Then, we have*

$$\frac{\partial \boldsymbol{A}\boldsymbol{x}}{\partial \mathrm{VEC}(\boldsymbol{A})} = \boldsymbol{x} \otimes \boldsymbol{I}_m. \tag{6.23}$$

Lemma 6.3 *For the vectors $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, we have*

$$\mathrm{VEC}(\boldsymbol{x}\boldsymbol{y}^\top) = (\boldsymbol{y} \otimes \boldsymbol{I}_m)\boldsymbol{x}, \tag{6.24}$$

where $\boldsymbol{I}_m$ denotes the $m \times m$ identity matrix.

For the derivation of the backpropagation algorithm, we tentatively assume that the bias terms are zero, i.e. $\boldsymbol{b}^{(l)} = \boldsymbol{0}$, $l = 1, \cdots, L$. In this case, the neural network parameter $\boldsymbol{\Theta}$ in (6.14) can be simplified as

$$\boldsymbol{\Theta} = \begin{bmatrix} \boldsymbol{W}^{(1)} \\ \vdots \\ \boldsymbol{W}^{(L)} \end{bmatrix}, \tag{6.25}$$

where $\boldsymbol{W}^{(l)} \in \mathbb{R}^{d^{(l)} \times d^{(l-1)}}$. Using the denominator layout as explained in Chap. 1, we have

$$\frac{\partial c}{\partial \boldsymbol{\Theta}} = \begin{bmatrix} \frac{\partial c}{\partial \boldsymbol{W}^{(1)}} \\ \vdots \\ \frac{\partial c}{\partial \boldsymbol{W}^{(L)}} \end{bmatrix}, \tag{6.26}$$

so that the weight at the l-th layer can be updated with the increment:

$$\Delta \boldsymbol{\Theta} = \begin{bmatrix} \Delta \boldsymbol{W}^{(1)} \\ \vdots \\ \Delta \boldsymbol{W}^{(L)} \end{bmatrix}, \quad \text{where} \quad \Delta \boldsymbol{W}^{(l)} = -\eta \frac{\partial c}{\partial \boldsymbol{W}^{(l)}}. \tag{6.27}$$

Therefore, $\partial c / \partial \boldsymbol{W}^{(l)}$ should be specified. More specifically, for a given training data set $\{\boldsymbol{x}_n, \boldsymbol{y}_n\}_{n=1}^N$, recall that the cost function $c(\boldsymbol{\Theta})$ in (6.12) is given by

$$c(\boldsymbol{\Theta}) = \sum_{n=1}^{N} \ell \left(\boldsymbol{y}_n, \boldsymbol{f}_{\Theta}(\boldsymbol{x}_n) \right), \tag{6.28}$$

where $\boldsymbol{f}_{\Theta}(\boldsymbol{x}_n)$ is defined in (6.13). Now define the l-th layer variable with respect to the n-th training data:

$$\boldsymbol{o}_n^{(l)} = \boldsymbol{\sigma}(\boldsymbol{g}_n^{(l)}), \quad \boldsymbol{g}_n^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{o}_n^{(l-1)}, \tag{6.29}$$

for $l = 1, \cdots, L$, with the initialization

$$\boldsymbol{o}_n^{(0)} := \boldsymbol{x}_n, \tag{6.30}$$

where the bias is assumed zero. Then, we have

$$\boldsymbol{o}_n^{(L)} = \boldsymbol{f}_{\Theta}(\boldsymbol{x}_n),$$

Using the chain rule for the denominator convention (see Eq. (1.40))

$$\frac{\partial c(\boldsymbol{g}(\boldsymbol{u}))}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{g}(\boldsymbol{u})}{\partial \boldsymbol{u}} \frac{\partial c(\boldsymbol{g})}{\partial \boldsymbol{g}} \tag{6.31}$$

we have

$$\frac{\partial c}{\partial \mathrm{VEC}(\boldsymbol{W}^{(l)})} = \sum_{n=1}^{N} \frac{\partial \boldsymbol{g}_n^{(l)}}{\partial \mathrm{VEC}(\boldsymbol{W}^{(l)})} \frac{\partial \ell\left(\boldsymbol{y}_n, \boldsymbol{o}_n^{(L)}\right)}{\partial \boldsymbol{g}_n^{(l)}}.$$

Furthermore, Lemma 6.2 informs us

$$\frac{\partial \boldsymbol{g}_n^{(l)}}{\partial \mathrm{VEC}(\boldsymbol{W}^{(l)})} = \boldsymbol{o}_n^{(l-1)} \otimes \boldsymbol{I}_{d^{(l)}}. \tag{6.32}$$

We further define the term:

$$\boldsymbol{\delta}_n^{(l)} := \frac{\partial \ell\left(\boldsymbol{y}_n, \boldsymbol{o}_n^{(L)}\right)}{\partial \boldsymbol{g}_n^{(l)}}, \tag{6.33}$$

which can be calculated using the chain rule (6.31) as follows:

$$\begin{aligned}\boldsymbol{\delta}_n^{(l)} &= \frac{\partial \boldsymbol{o}_n^{(l)}}{\partial \boldsymbol{g}_n^{(l)}} \frac{\partial \boldsymbol{g}_n^{(l+1)}}{\partial \boldsymbol{o}_n^{(l)}} \cdots \frac{\partial \boldsymbol{o}_n^{(L)}}{\partial \boldsymbol{g}_n^{(L)}} \frac{\partial \ell\left(\boldsymbol{y}_n, \boldsymbol{o}_n^{(L)}\right)}{\partial \boldsymbol{o}_n^{(L)}} \\ &= \boldsymbol{\Lambda}_n^{(l)} \boldsymbol{W}^{(l+1)\top} \boldsymbol{\Lambda}_n^{(l+1)} \boldsymbol{W}^{(l+2)\top} \cdots \boldsymbol{W}^{(L)\top} \boldsymbol{\Lambda}_n^{(L)} \boldsymbol{\epsilon}_n \end{aligned} \tag{6.34}$$

for $l = 1, \cdots, L$, and the error term $\boldsymbol{\epsilon}_n$ is computed by

$$\boldsymbol{\epsilon}_n = \frac{\partial \ell\left(\boldsymbol{y}_n, \boldsymbol{o}_n^{(L)}\right)}{\partial \boldsymbol{o}_n^{(L)}}.$$

In (6.34), we use

$$\boldsymbol{\Lambda}_n^{(l)} := \frac{\partial \boldsymbol{o}_n^{(l)}}{\partial \boldsymbol{g}_n^{(l)}} = \frac{\partial \boldsymbol{\sigma}\left(\boldsymbol{g}_n^{(l)}\right)}{\partial \boldsymbol{g}_n^{(l)}} \in \mathbb{R}^{d^{(l)} \times d^{(l)}}, \tag{6.35}$$

which is calculated using the denominator layout as explained in Chap. 1, and

$$\frac{\partial \boldsymbol{g}_n^{(l+1)}}{\partial \boldsymbol{o}_n^{(l)}} = \frac{\partial \boldsymbol{W}^{(l+1)} \boldsymbol{o}_n^{(l)}}{\partial \boldsymbol{o}_n^{(l)}} = \boldsymbol{W}^{(l+1)\top}, \tag{6.36}$$

which is obtained using the denominator convention (see (1.41) in Chap. 1). Accordingly, we have

$$\begin{aligned}\frac{\partial c}{\partial \mathrm{VEC}(\boldsymbol{W}^{(l)})} &= \sum_{n=1}^{N} \frac{\partial \boldsymbol{g}_n^{(l)}}{\partial \mathrm{VEC}(\boldsymbol{W}^{(l)})} \frac{\partial \ell\left(\boldsymbol{y}_n, \boldsymbol{o}_n^{(L)}\right)}{\partial \boldsymbol{g}_n^{(l)}} \\ &= \sum_{n=1}^{N} \left(\boldsymbol{o}_n^{(l-1)} \otimes \boldsymbol{I}_{d^{(l)}}\right) \boldsymbol{\delta}_n^{(l)} \\ &= \sum_{n=1}^{N} \mathrm{VEC}\left(\boldsymbol{\delta}_n^{(l)} \boldsymbol{o}_n^{(l-1)\top}\right),\end{aligned}$$

where we use (6.32) and (6.33) for the second equality, and Lemma 6.3 for the last equality. Finally, we have the following derivative of the cost with respect to $\boldsymbol{W}^{(l)}$:

$$\begin{aligned}\frac{\partial c}{\partial \boldsymbol{W}^{(l)}} &= \mathrm{UNVEC}\left(\frac{\partial c}{\partial \mathrm{VEC}(\boldsymbol{W}^{(l)})}\right) \\ &= \mathrm{UNVEC}\left(\sum_{n=1}^{N} \mathrm{VEC}\left(\boldsymbol{\delta}_n^{(l)} \boldsymbol{o}_n^{(l-1)\top}\right)\right) \\ &= \sum_{n=1}^{N} \boldsymbol{\delta}_n^{(l)} \boldsymbol{o}_n^{(l-1)\top},\end{aligned}$$

where we use the linearity of $\mathrm{UNVEC}(\cdot)$ operator for the last equality. Therefore, the weight update increment is given by

$$\begin{aligned}\Delta \boldsymbol{W}^{(l)} &= -\eta \frac{\partial c}{\partial \boldsymbol{W}^{(l)}} \\ &= -\eta \sum_{n=1}^{N} \boldsymbol{\delta}_n^{(l)} \boldsymbol{o}_n^{(l-1)\top}. \end{aligned} \tag{6.37}$$

6.4.2 *Geometrical Interpretation of BP Algorithm*

This weight update scheme in (6.37) is the key in BP. Not only is the final form of the weight update in (6.37) very concise, but it also has a very important geometric meaning, which deserves further discussion. In particular, the update is totally determined by the outer product of the two terms $\boldsymbol{\delta}_n^{(l)}$ and $\boldsymbol{o}_n^{(l-1)}$, i.e. $\boldsymbol{\delta}_n^{(l)} \boldsymbol{o}_n^{(l-1)\top}$. Why are these terms so important? This is the main discussion point in this section.

First, recall that $\boldsymbol{o}_n^{(l-1)}$ is the $(l-1)$-th layer neural network output given by (6.29). Since this term is calculated in the forward path of the neural network, it is nothing but the *forward-propagated input* to the l-th layer neuron. Second, recall that

$$\boldsymbol{\epsilon}_n = \frac{\partial \ell\left(\boldsymbol{y}_n, \boldsymbol{o}_n^{(L)}\right)}{\partial \boldsymbol{o}_n^{(L)}}.$$

If we use the l_2 loss, this term becomes

$$\begin{aligned}\boldsymbol{\epsilon}_n &= \frac{\partial\left(\frac{1}{2}\|\boldsymbol{y}_n - \boldsymbol{o}_n^{(L)}\|^2\right)}{\partial \boldsymbol{o}_n^{(L)}} \\ &= \boldsymbol{o}_n^{(L)} - \boldsymbol{y}_n,\end{aligned}$$

which is indeed the estimation error of the neural network output. Since we have

$$\boldsymbol{\delta}_n^{(l)} = \boldsymbol{\Lambda}_n^{(l)} \boldsymbol{W}^{(l+1)\top} \boldsymbol{\Lambda}_n^{(l+1)} \boldsymbol{W}^{(l+2)\top} \cdots \boldsymbol{W}^{(L)\top} \boldsymbol{\Lambda}_n^{(L)} \boldsymbol{\epsilon}_n\,, \tag{6.38}$$

this implies that $\boldsymbol{\delta}_n^{(l)}$ is indeed *the backward-propagated estimation error* down to the l-th layer. Therefore, we can find that the weight update is determined by the outer product of the forward-propagated input and backward-propagated estimation error.

In terms of calculation, the forward and backward terms $\boldsymbol{o}_n^{(l)}$ and $\boldsymbol{\delta}_n^{(l)}$ can be efficiently calculated using recursive formulae. More specifically, we have

$$\boldsymbol{o}_n^{(l-1)} = \boldsymbol{\sigma}\left(\boldsymbol{W}^{(l-1)} \boldsymbol{o}_n^{(l-2)}\right), \tag{6.39}$$

$$\boldsymbol{\delta}_n^{(l)} = \boldsymbol{\Lambda}_n^{(l)} \boldsymbol{W}^{(l+1)\top} \boldsymbol{\delta}_n^{(l+1)}, \tag{6.40}$$

with the initialization by

$$\boldsymbol{o}_n^{(0)} = \boldsymbol{x}_n, \quad \boldsymbol{\delta}_n^{(L)} = \boldsymbol{\epsilon}_n. \tag{6.41}$$

The geometric interpretation and recursive formulae are illustrated in Fig. 6.8.

6.4.3 Variational Interpretation of BP Algorithm

The variational principle is a scientific principle used within the *calculus of variations* [29], which develops general methods for finding functions that minimize the value of quantities that depend upon those functions. The calculus of variations

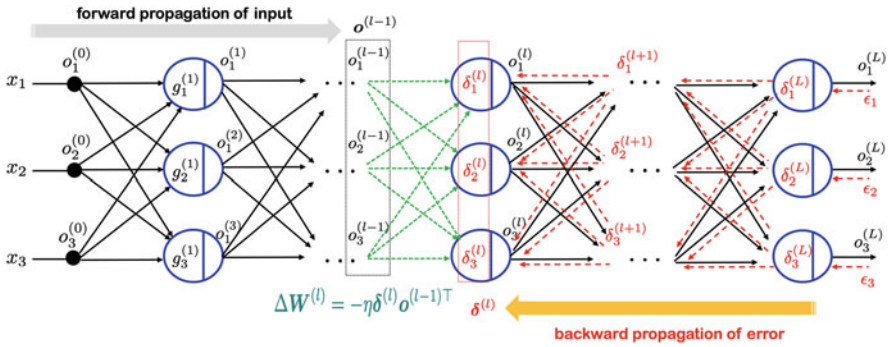

Fig. 6.8 Geometry of backpropagation

is a field of mathematical analysis pioneered by Isaac Newton, which uses variations to reduce the energy function [29].

Given the incremental variation in (6.37), we are therefore interested in finding whether it indeed reduces the energy function. For this, let us consider a simplified form of the loss function with l_2 loss with $N = 1$. In the following, we show that for the case of neural networks with ReLU activation functions, the BP algorithm is indeed equivalent to the variational approach.

More specifically, let the baseline energy function, which refers to the cost function before the perturbation, be given by

$$\ell(\boldsymbol{y}, \boldsymbol{o}^{(L)}) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{o}^{(L)}\|^2, \tag{6.42}$$

where the subscript n for the training data index is neglected here for simplicity and

$$\boldsymbol{o}^{(L)} := \boldsymbol{\sigma}\left(\boldsymbol{W}^{(L)}\boldsymbol{o}^{(L-1)}\right), \tag{6.43}$$

One of the important observations is that for the case of the ReLU, (6.43) can be represented by

$$\boldsymbol{o}^{(L)} := \boldsymbol{\Gamma}^{(L)}\boldsymbol{g}^{(L)}, \quad \text{where} \quad \boldsymbol{g}^{(L)} = \boldsymbol{W}^{(L)}\boldsymbol{o}^{(L-1)}, \tag{6.44}$$

where $\boldsymbol{\Gamma}^{(L)} \in \mathbb{R}^{d^{(L)} \times d^{(L)}}$ is a diagonal matrix with 0 and 1 values given by

$$\boldsymbol{\Gamma}^{(L)} = \begin{bmatrix} \gamma_1 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & \gamma_j & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & \gamma_{d^{(L)}} \end{bmatrix}, \tag{6.45}$$

where

$$\gamma_j = \gamma\left([\boldsymbol{g}^{(L)}]_j\right), \tag{6.46}$$

where $[\boldsymbol{g}^{(L)}]_j$ denotes the j-th element of the vector $\boldsymbol{g}^{(L)}$ and $\gamma(\cdot)$ is defined in (6.4). Thanks to (6.5), we have

$$\boldsymbol{\Gamma}^{(l)} = \boldsymbol{\Lambda}^{(l)}, \quad l = 1, \cdots, L, \tag{6.47}$$

where $\boldsymbol{\Lambda}^{(l)}$ is defined as the derivative of the activation function in (6.35). Therefore, using the recursive formula, we have

$$\boldsymbol{o}^{(L)} = \boldsymbol{\Lambda}^{(L)}\boldsymbol{W}^{(L)} \cdots \boldsymbol{\Lambda}^{(l)}\boldsymbol{W}^{(l)}\boldsymbol{o}^{(l-1)}. \tag{6.48}$$

Using this, we now investigate whether the cost decreases with the perturbed weight

$$\Delta \boldsymbol{W}^{(l)} = -\eta \boldsymbol{\delta}^{(l)}\boldsymbol{o}^{(l-1)\top}. \tag{6.49}$$

When the step size η is sufficiently small, then the ReLU activation patterns from $\boldsymbol{W}^{(l)} + \Delta \boldsymbol{W}^{(l)}$ do not change from those by $\boldsymbol{W}^{(l)}$ (this issue will be discussed later), so that the new cost function value is given by

$$\widehat{\ell}(\boldsymbol{y}, \boldsymbol{o}^{(L)}) := \|\boldsymbol{y} - \boldsymbol{\Lambda}^{(L)}\boldsymbol{W}^{(L)} \cdots \boldsymbol{\Lambda}^{(l)}(\boldsymbol{W}^{(l)} + \Delta \boldsymbol{W}^{(l)})\boldsymbol{o}^{(l-1)}\|^2.$$

Recall that we have

$$\begin{aligned}\boldsymbol{\delta}^{(L)} &= \boldsymbol{o}^{(L)} - \boldsymbol{y} \\ &= \boldsymbol{\Lambda}^{(L)}\boldsymbol{W}^{(L)} \cdots \boldsymbol{\Lambda}^{(l)}\boldsymbol{W}^{(l)}\boldsymbol{o}^{(l-1)} - \boldsymbol{y}.\end{aligned}$$

Accordingly, we have

$$\begin{aligned}\widehat{\ell}(\boldsymbol{y}, \boldsymbol{o}^{(L)}) &= \| - \boldsymbol{\delta}^{(L)} - \boldsymbol{\Lambda}^{(L)}\boldsymbol{W}^{(L)} \cdots \boldsymbol{\Lambda}^{(l)}\Delta \boldsymbol{W}^{(l)}\boldsymbol{o}^{(l-1)}\|^2 \\ &= \| - \boldsymbol{\delta}^{(L)} + \eta \boldsymbol{\Lambda}^{(L)}\boldsymbol{W}^{(L)} \cdots \boldsymbol{\Lambda}^{(l)}\boldsymbol{\delta}^{(l)}\boldsymbol{o}^{(l-1)\top}\boldsymbol{o}^{(l-1)}\|^2 \\ &= \left\| \left(\boldsymbol{I} - \eta \|\boldsymbol{o}^{(l-1)}\|^2 \boldsymbol{M}^{(l)}\right) \boldsymbol{\delta}^{(L)} \right\|^2,\end{aligned} \tag{6.50}$$

where we use $\|\boldsymbol{o}^{(l-1)}\|^2 = \boldsymbol{o}^{(l-1)\top}\boldsymbol{o}^{(l-1)}$ and

$$\boldsymbol{M}^{(l)} = \boldsymbol{\Lambda}^{(L)}\boldsymbol{W}^{(L)} \cdots \boldsymbol{W}^{(l+1)}\boldsymbol{\Lambda}^{(l)}\boldsymbol{\Lambda}^{(l)}\boldsymbol{W}^{(l+1)\top} \cdots \boldsymbol{W}^{(L)\top}\boldsymbol{\Lambda}^{(L)},$$

which comes from (6.38). Now, we can easily see that for all $\boldsymbol{x} \in \mathbb{R}^{d^{(L)}}$ we have

$$\boldsymbol{x}^\top \boldsymbol{M}^{(l)} \boldsymbol{x} = \|\boldsymbol{\Lambda}^{(l)} \boldsymbol{W}^{(l+1)\top} \cdots \boldsymbol{W}^{(L)\top} \boldsymbol{\Lambda}^{(L)} \boldsymbol{x}\|^2 \geq 0, \tag{6.51}$$

so that $\boldsymbol{M}^{(l)}$ is positive semidefinite, i.e. its eigenvalues are non-negative. Furthermore, we have

$$\begin{aligned} \left\| \left(\boldsymbol{I} - \eta \|\boldsymbol{o}^{(l-1)}\|^2 \boldsymbol{M}^{(l)} \right) \boldsymbol{\delta}^{(L)} \right\|^2 &\leq \lambda_{\max}^2 \left(\boldsymbol{I} - \eta \|\boldsymbol{o}^{(l-1)}\|^2 \boldsymbol{M}^{(l)} \right) \\ &\quad \times \|\boldsymbol{\delta}^{(L)}\|^2, \end{aligned} \tag{6.52}$$

where $\lambda_{\max}(\boldsymbol{A})$ denotes the largest eigenvalue of $\boldsymbol{A}$. In addition, we have

$$\lambda_{\max}^2 \left(\boldsymbol{I} - \eta \|\boldsymbol{o}^{(l-1)}\|^2 \boldsymbol{M}^{(l)} \right) = \left(1 - \eta \|\boldsymbol{o}^{(l-1)}\|^2 \lambda_{\max} \left(\boldsymbol{M}^{(l)} \right) \right)^2.$$

Therefore, if the largest eigenvalue satisfies

$$0 \leq \lambda_{\max} \left(\boldsymbol{M}^{(l)} \right) \leq \frac{2}{\eta \|\boldsymbol{o}^{(l-1)}\|^2}, \tag{6.53}$$

we can show

$$\widehat{\ell}(\boldsymbol{y}, \boldsymbol{o}^{(L)}) \leq \|\boldsymbol{\delta}^{(l)}\|^2 = \ell(\boldsymbol{y}, \boldsymbol{o}^{(L)}),$$

so the cost function value decreases with the perturbation.

It is important to emphasize that this strong convergence result is due to the unique property of the ReLU in (6.47), which is never satisfied with other activation functions. This may be another reason for the success of the ReLU in modern deep learning. Having said this, care should be taken since this argument is true only for sufficiently small step size η, so that the ReLU activation patterns after the perturbation do not change. In fact, this may be another reason to choose an appropriate step size in the optimization algorithm.

6.4.4 *Local Variational Formulation*

Another way of understanding BP is via propagation of the cost function. As shown in Fig. 6.8, after the forward and backward propagation of the input and error, respectively, the resulting optimization problem for the weight update at the l-th layer is given by

$$\min_{\boldsymbol{W}} \| - \boldsymbol{\delta}^{(l)} - \boldsymbol{W} \boldsymbol{o}^{(l-1)}\|^2. \tag{6.54}$$

Note that we have a minus sign in front of $\boldsymbol{\delta}^{(l)}$ inspired by its global counterpart in (6.50). By inspection, we can easily see that the optimal solution for (6.54) is given by

$$\boldsymbol{W}^* = -\frac{1}{\|\boldsymbol{o}^{(l-1)}\|^2}\boldsymbol{\delta}^{(l)}\boldsymbol{o}^{(l-1)\top}, \tag{6.55}$$

since plugging (6.55) in (6.54) makes the cost function zero. Therefore, the optimal search direction for the weight update should be given by

$$\Delta \boldsymbol{W}^{(l)} = -\eta \boldsymbol{\delta}^{(l)}\boldsymbol{o}^{(l-1)\top}, \tag{6.56}$$

which is equivalent to (6.49). The take-away message here is that as long as we can obtain the back-propagated error and the forward-propagated input, we can obtain a local variational formulation, which can be solved by any means.

6.5 Exercises

1. Derive the general form of the activation function $\sigma(x)$ that satisfies the following differential equation:

$$\frac{\sigma(x)}{x} = \frac{\partial \sigma(x)}{\partial x}$$

2. Show that (6.21) is equivalent to (6.20).
3. Recall that L-layer feedforward neural network can be represented recursively by

$$\boldsymbol{o}^{(l)} = \boldsymbol{\sigma}(\boldsymbol{g}^{(l)}), \quad \boldsymbol{g}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{o}^{(l-1)} + \boldsymbol{b}^{(l)}, \tag{6.57}$$

for $l = 1, \cdots, L$. When the training data size is 1, the weight update is given by

$$\Delta \boldsymbol{W}^{(l)} = -\gamma \boldsymbol{\delta}^{(l)}\boldsymbol{o}^{(l-1)\top}, \tag{6.58}$$

where $\gamma > 0$ is the step size and

$$\boldsymbol{\delta}^{(l)} := \frac{\partial \ell\left(\boldsymbol{y}, \boldsymbol{o}^{(L)}\right)}{\partial \boldsymbol{g}^{(l)}}. \tag{6.59}$$

   a. Derive the update equation similar to (6.58) for the bias term, i.e. $\Delta \boldsymbol{b}^{(l)}$.
   b. Suppose the weight matrix $\boldsymbol{W}^{(l)}, l =, \cdots, L$ is a diagonal matrix. Draw the network connection architecture similarly to Fig. 6.8. Then, derive the

backprop algorithm for the diagonal term of the weight matrix, assuming that the bias is zero. You must use the chain rule to derive this.

4. Let a two-layer ReLU neural network $\boldsymbol{f}_{\boldsymbol{\Theta}}$ have an input and output dimension for each layer in $\mathbb{R}^2$, i.e. $\boldsymbol{f}_{\boldsymbol{\Theta}} : \boldsymbol{x} \in \mathbb{R}^2 \mapsto \boldsymbol{f}_{\boldsymbol{\Theta}}(\boldsymbol{x}) \in \mathbb{R}^2$. Suppose that the parameter $\boldsymbol{\Theta}$ of the network is composed of weight and bias:

$$\boldsymbol{\Theta} = \left\{ \boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)} \right\}, \tag{6.60}$$

which are initialized as follows:

$$\boldsymbol{W}^{(1)} = \boldsymbol{W}^{(2)} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}, \quad \boldsymbol{b}^{(1)} = \boldsymbol{b}^{(2)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{6.61}$$

Then, for a given l_2 loss function

$$\ell(\boldsymbol{\Theta}) = \frac{1}{2} \| \boldsymbol{y} - \boldsymbol{f}(\boldsymbol{x}) \|^2 \tag{6.62}$$

and a training data

$$\boldsymbol{x} = [1, -1]^\top, \quad \boldsymbol{y} = [1, 0]^\top, \tag{6.63}$$

compute the weight and bias update for the first two iterations of the backpropagation algorithm. It is suggested that the unit step size, i.e. $\gamma = 1$, be used.

5. We are now interested in extending (6.54) for the training data composed of N samples.

   a. Show that the following equality holds for the local variation formulation:

$$\min_{\boldsymbol{W}} \sum_{n=1}^{N} \| - \boldsymbol{\delta}_n^{(l)} - \boldsymbol{W} \boldsymbol{o}_n^{(l-1)} \|^2 = \min_{\boldsymbol{W}} \| - \boldsymbol{\Delta}^{(l)} - \boldsymbol{W} \boldsymbol{O}^{(l-1)} \|_F^2, \tag{6.64}$$

   where $\| \cdot \|_F$ denotes the Frobenious norm and

$$\boldsymbol{\Delta}^{(l)} = \begin{bmatrix} \boldsymbol{\delta}_1^{(l)} & \cdots & \delta_N^{(l)} \end{bmatrix}, \quad \boldsymbol{O}^{(l-1)} = \begin{bmatrix} \boldsymbol{o}_1^{(l-1)} & \cdots & \boldsymbol{o}_N^{(l-1)} \end{bmatrix}.$$

   b. Show that there exists a step size $\gamma > 0$ such that the weight perturbation

$$\Delta \boldsymbol{W}^{(l)} = -\gamma \sum_{n=1}^{N} \boldsymbol{\delta}_n^{(l)} \boldsymbol{o}_n^{(l-1)\top}$$

   reduces the cost value in (6.64).

6. Suppose that our activation function is sigmoid. Derive the BP algorithm for the L-layer neural network. What is the main difference of the BP algorithm compared to the network with a ReLU? Is this an advantage or disadvantage? Answer this question in terms of variational perspective.
7. Now we are interested in extending the model in (6.6) to a convolutional neural network model

$$o_i^{(l)} = \sigma\left(\sum_{j=1}^{d^{(l)}} h_{i-j}^{(l)} o_j^{(l-1)} + b_i^{(l)}\right), \tag{6.65}$$

for $i = 1, \cdots, d^{(l)}$, where $h_i^{(l)}$ is the i-th element of the filter $\boldsymbol{h}^{(l)} = [h_1^{(l)}, \cdots, h_p^{(l)}]^\top$.

a. If we want to represent this convolutional neural network in a matrix form,

$$\boldsymbol{o}^{(l)} = \boldsymbol{\sigma}\left(\boldsymbol{W}^{(l)}\boldsymbol{o}^{(l-1)} + \boldsymbol{b}^{(l)}\right), \tag{6.66}$$

what is the corresponding weight matrix $\boldsymbol{W}^{(l)}$? Please show the structure of $\boldsymbol{W}^{(l)}$ explicitly in terms of $\boldsymbol{h}^{(l)}$ elements.

b. Derive the backpropagation algorithm for the filter update $\Delta \boldsymbol{h}^{(l)}$.

Chapter 7
Convolutional Neural Networks

7.1 Introduction

A convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, widely used for analyzing and processing images. Multilayer perceptrons, which we discussed in the previous chapter, usually require fully connected networks, where each neuron in one layer is connected to all neurons in the next layer. Unfortunately, this type of connections inescapably increases the number of weights. In CNNs, the number of weights can be significantly reduced using their shared-weights architecture originated from translation invariant characteristics of the convolution.

A convolutional neural network was first developed by Yann LeCun for handwritten zip code identification [21], inspired by the famous experiments by Hubel and Wiesel for a cat's primary visual cortex [20]. Recall that Hubel and Wiesel found that simple cells in the primary visual cortex of a cat respond best to edge-like stimuli at a particular orientation, position, and phase within their relatively small receptive fields. Yann LeCun realized that the aggregation of LGN (lateral geniculate nucleus) cells with the same receptive field is similar to the convolution operation, which led him to construct a neural network as the cascaded applications of convolution, nonlinearity, and image subsampling, followed by fully connected layers that determine linear hyperplanes in the feature space for the classification tasks. The resulting network architecture, shown in Fig. 7.1, is called LeNet [21].

While the algorithm worked, training to learn 10 digits required 3 days! Many factors contributed to the slow speed, including the vanishing gradient problem, which will be discussed later. Therefore, simpler models that use task-specific handcrafted features such as support vector machines (SVMs) or kernel machines [11] were popular choices in the 1990s and 2000s, because of the artificial neural network's (ANN) computational cost and a lack of understanding of its working mechanism. In fact, the lack of understanding of the ANN has been the main criticism of many contemporary scientists, including the famous Vladmir Vapnik,

J. C. Ye, *Geometry of Deep Learning*, Mathematics in Industry 37,
https://doi.org/10.1007/978-981-16-6046-7_7

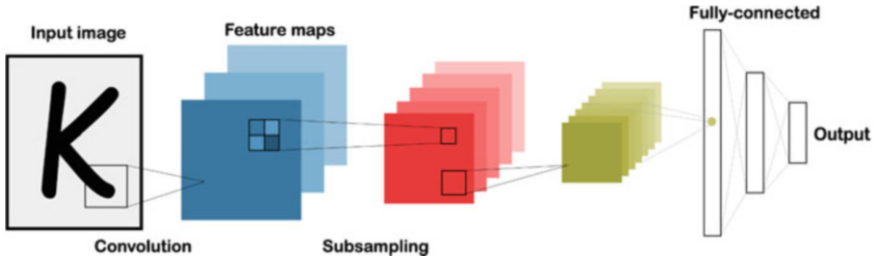

Fig. 7.1 LeNet: the first CNN proposed by Yann LeCun for zip code identification [21]

the inventor of the SVM. In the preface of his classical book entitled *The Nature of Statistical Learning Theory* [10], Vapnik expressed his concern saying that "Among artificial intelligence researchers the *hardliners* had considerable influence (it is precisely they who declared that complex theories do not work, simple algorithms do)".

Ironically, the advent of the SVM and kernel machines has led to a long period of decline in neural network research, often referred to as the "AI winter". During the AI winter, the neural network researchers were largely considered pseudo-scientists and even had difficulty in securing research funding. Although there have been several notable publications on neural networks during the AI winter, the revival of convolutional neural network research, up to the level of general public acceptance, has had to wait until the series of deep neural network breakthroughs at the ILSVRC (ImageNet Large Scale Visual Recognition Competition).

In the following section, we give a brief overview of the history of modern CNN research that has contributed to the revival of research on neural networks.

7.2 History of Modern CNNs

7.2.1 AlexNet

ImageNet is a large visual database designed for use in visual object recognition software research [8]. ImageNet contains more than 20,000 categories, consisting of several hundred images. Since 2010, the ImageNet project has an annual software contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [7], where software programs compete to correctly classify and detect objects and scenes. Around 2011, a good ILSVRC classification error rate, which was based on classical machine learning approaches, was about 27%.

In the 2012 ImageNet Challenge, Krizhevsky et al. [9] proposed a CNN architectures, shown in Fig. 7.2, which is now known as AlexNet. The AlexNet architecture is composed of five convolution layers and three fully connected layers. In fact, the basic components of AlexNet were nearly the same as those of LeNet by

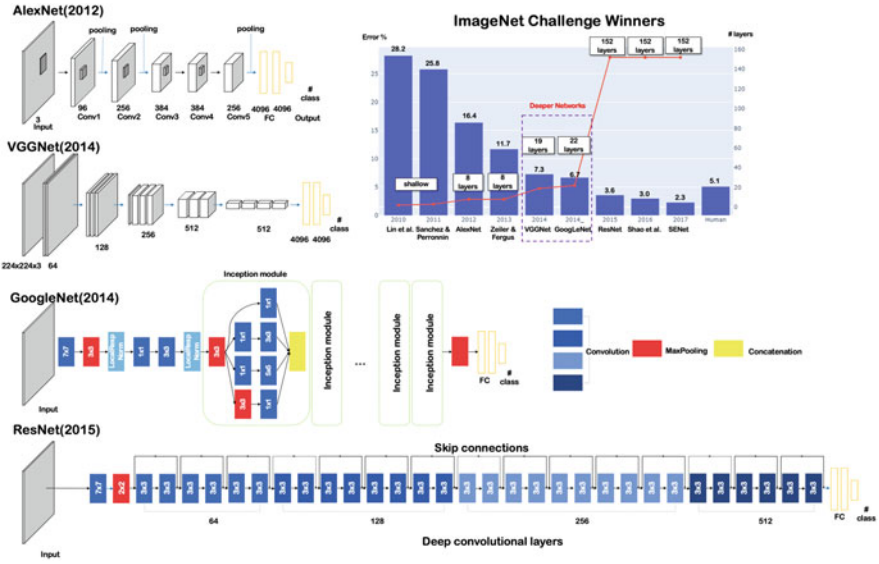

Fig. 7.2 The ImageNet challenges and the CNN winners that have completely changed the landscape of artificial intelligence

Yann LeCun [21], except the new nonlinearity using the rectified linear unit (ReLU). AlexNet got a Top-5 error rate (rate of not finding the true label of a given image among its top 5 predictions) of 15.3%. The next best result in the challenge, which was based on the classical kernel machines, trailed far behind (26.2%).

In fact, the celebrated victory of AlexNet declared the start of a "new era" in data science, as witnessed by more than 75k citations according to Google Scholar as of January 2021. With the introduction of AlexNet, the world was no longer the same, and all the subsequent winners at the ImageNet challenges were deep neural networks, and nowadays CNN surpasses the human observers in ImageNet classification. In the following, we introduce several subsequent CNN architectures which have made significant contributions in deep learning research.

7.2.2 GoogLeNet

GoogLeNet [30] was the winner at the 2014 ILSVRC (see Fig. 7.2). As the name "GoogLeNet" indicates, it is from Google, but one may wonder why it is not written as "GoogleNet". This is because the researchers of "GoogLeNet" tried to pay tribute to Yann LeCun's LeNet [21] by containing the word "LeNet".

The network architecture is quite different from AlexNet due to the so-called inception module[30], shown in Fig. 7.3. Specifically, at each inception module, there exist different sizes/types of convolutions for the same input and stack-

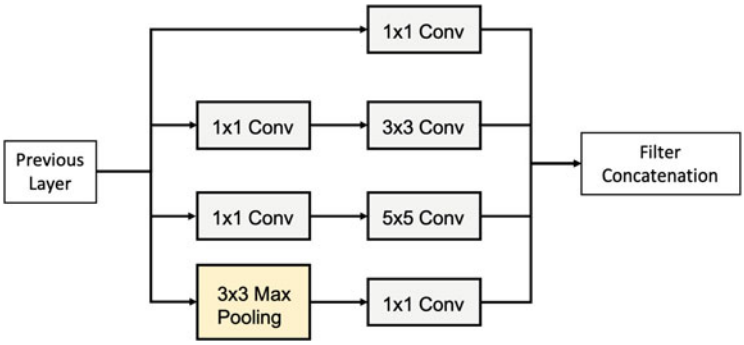

Fig. 7.3 Inception module in GoogLeNet

ing all the outputs. This idea was inspired by the famous 2010 science fiction film *Inception*, in which Leonardo DiCaprio starred. In the film, the renowned director Christoper Nolan wanted to explore "the idea of people sharing a dream space... That gives you the ability to access somebody's unconscious mind." The key concept which GoogLeNet borrowed from the film was the "dream within a dream" strategy, which led to the "network within a network" strategy that improves the overall performance.

7.2.3 VGGNet

VGGNet [31] was invented by the VGG (Visual Geometry Group) from University of Oxford for the 2014 ILSVRC (see Fig. 7.2). Although VGGNet was not the winner of the 2014 ILSVRC (GoogLeNet was the winner at that time, and the VGGNet came second), VGGNet has made a prolonged impact in the machine learning community due to its modular and simple architecture, yet resulting in a significant performance improvement over AlexNet [9]. In fact, the pretrained VGGNet model captures many important image features; therefore, it is still widely used for various purposes such as perceptual loss [32], etc. Later we will use VGGNet to visualize CNNs.

As shown in Fig. 7.2, VGGNet is composed of multiple layers of convolution, max pooling, the ReLU, followed by fully connected layers and softmax. One of the most important observations of VGGNet is that it achieves an improvement over AlexNet by replacing large kernel-sized filters with multiple 3×3 kernel-sized filters. As will be shown later, for a given receptive field size, cascaded application of a smaller size kernel followed by the ReLU makes the neural network more expressive than one with a larger kernel size. This is why VGGNet provided significantly improved performance over AlexNet despite its simple structure.

7.2.4 ResNet

In the history of ILSVRC, the Residual Network (ResNet) [33] is considered another masterpiece, as shown in its citation record of more than 68k as of January 2020.

Since the representation power of a deep neural network increases with the network depth, there has been strong research interest in increasing the network depth. For example, AlexNet [9] from 2012 LSVRC had only five convolutional layers, while the VGG network [31] and GoogLeNet [30] from 2014 LSVRC had 19 and 22 layers, respectively. However, people soon realized that a deeper neural network is hard to train. This is because of the vanishing gradient problem, where the gradient can be easily back-propagated to layers closer to the output, but is difficult to be back-propagated far from the output layer since the repeated multiplication may make the gradient so small. As discussed in the previous chapter, the ReLU nonlinearity partly mitigates the problem, since the forward and backward propagation are symmetric, but still the deep neural network turns out to be difficult to train due to an unfavorable *optimization landscape* [34]; this issue will be reviewed later.

As shown in Fig. 7.2, there exist bypass (or skip) connections in the ResNet, representing an identity mapping. The bypass connection was proposed to promote the gradient back-propagation. Thanks to the skip connection, ResNet makes it possible to train up to hundreds or even thousands of layers, achieving a significant performance improvement. Recent researches reveals that the bypass connection also improves the forward propagation, making the representation more expressive [35]. Furthermore, its optimization landscape can be significantly improved thanks to bypass connections that eliminate many local minimizers [35, 36].

7.2.5 DenseNet

DenseNet (Dense Convolutional Network) [37] exploits the extreme form of skip connection as shown in Fig. 7.4. In DenseNet, at each layer there exists skip connections from all preceding layers to obtain additional inputs.

Since each layer receives inputs from all preceding layers, the representation power of the network increases significantly, which makes the network compact, thereby reducing the number of channels. With dense connections, the authors demonstrated that fewer parameters and higher accuracy are achieved compared to ResNet [37].

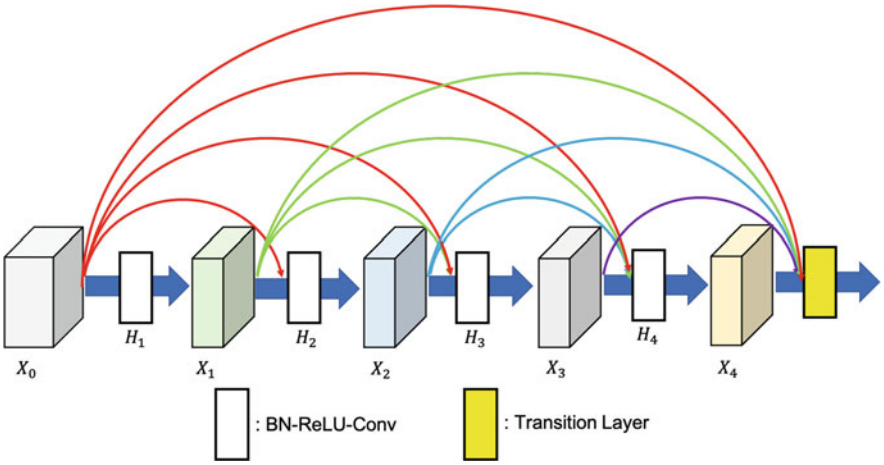

Fig. 7.4 Architecture of DenseNet

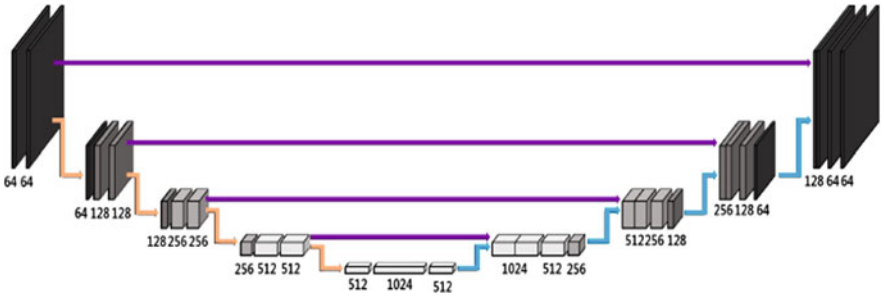

Fig. 7.5 Architecture of U-Net

7.2.6 U-Net

Unlike the aforementioned networks that are designed for ImageNet classification task, the U-Net architecture [38] in Fig. 7.5 was originally proposed for biomedical image segmentation, and is widely used for inverse problems [39, 40].

One of the unique aspects of U-Net is its symmetric encoder–decoder architecture. The encoder part consists of 3×3 convolution, batch normalization [41], and the ReLU. In the decoder part, upsampling and 3×3 convolution are used. Also, there are max pooling layers and skip connections through channel concatenation.

The multi-scale architecture of U-Net significantly increases the receptive field, which may be the main reason for the success of U-Net for segmentation, inverse problems, etc., where global information from all over the images is necessary to update the local image information. This issue will be discussed later. Moreover, the skip connection is important to retain the high-frequency content of the input signal.

The symmetric and multi-scale architecture of U-Net inspired many signal processing discoveries [42], providing important insights into understanding the geometry of deep neural networks.

7.3 Basic Building Blocks of CNNs

Although the aforementioned CNN architectures appear complicated, a closer look at them reveals that they are nothing but cascaded combinations of simple building blocks such as convolution, pooling/unpooling, ReLU, etc. These components are even considered as basic or "primitive" tools in signal processing. In fact, the emergence of the superior performance from the combination of the basic tools is one of the mysteries of deep neural networks, which will be discussed extensively later. In the meanwhile, this section provides a detailed explanation of the basic building blocks of CNNs.

7.3.1 Convolution

The convolution is an operation that originates from fundamental properties of linear time invariant (LTI) or linear spatially invariant (LSI) systems. Specifically, for a given LSI system, let $\boldsymbol{h}$ denote the impulse response, then the output image $\boldsymbol{y}$ with respect to the input image $\boldsymbol{x}$ can be computed by

$$\boldsymbol{y} = \boldsymbol{h} * \boldsymbol{x}, \tag{7.1}$$

where $*$ denotes the convolution operation. For example, the 3×3 convolution case for 2-D images can be represented element by element as follows:

$$y[m,n] = \sum_{p,q=-1}^{1} h[p,q]x[m-p,n-q], \tag{7.2}$$

where $y[m,n]$, $h[m,n]$ and $x[m,n]$ denote the (m,n)-element of the matrices $\boldsymbol{Y}$, $\boldsymbol{H}$ and $\boldsymbol{X}$, respectively. One example of computing this convolution is illustrated in Fig. 7.6, where the filter is already flipped for visualization.

It is important to note that the convolution used in CNNs is richer than the simple convolution in (7.1) and Fig. 7.6. For example, a three channel input signal can generate a single channel output as shown in Fig. 7.7a, which is often referred to as multi-input single-output (MISO) convolution. In another example shown in Fig. 7.7b, a 5×5 filter kernel is used to generate 6 (resp. 10) output channels from 3 (resp. and 6) input channels. This is often called the multi-input multi-output

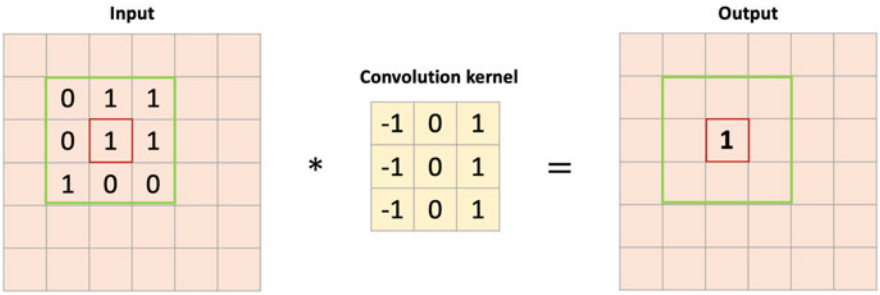

Fig. 7.6 An example of convolution with 3×3 filter

(MIMO) convolution. Finally, in Fig. 7.7c, the 1×1 filter kernel is used to generate 32 output channels from 64 input channels.

All these seemingly different convolutional operations can be written in a general MIMO convolution form:

$$\boldsymbol{y}_i = \sum_{j=1}^{c_{in}} \boldsymbol{h}_{i,j} * \boldsymbol{x}_j, \quad i = 1, \cdots, c_{out}, \tag{7.3}$$

where c_{in} and c_{out} denote the number of input and output channels, respectively, $\boldsymbol{x}_j$, $\boldsymbol{y}_i$ refer to the j-th input and the i-th output channel image, respectively, and $\boldsymbol{h}_{i,j}$ is the convolution kernel that contributes to the i-th channel output by convolving with the j-th input channel images. For the case of 1×1 convolution, the filter kernel becomes

$$\boldsymbol{h}_{i,j} = w_{ij}\delta[0, 0],$$

so that (7.3) becomes the weighted sum of input channel images as follows:

$$\boldsymbol{y}_i = \sum_{j=1}^{c_{in}} w_{ij}\boldsymbol{x}_j, \quad i = 1, \cdots, c_{out}. \tag{7.4}$$

7.3.2 Pooling and Unpooling

A pooling layer is used to progressively reduce the spatial size of the representation to reduce the number of parameters and amount computation in the network. The pooling layer operates on each feature map independently. The most common approaches used in pooling are max pooling and average pooling as shown in Fig. 7.8b. In this case, the pooling layer will always reduce the size of each feature

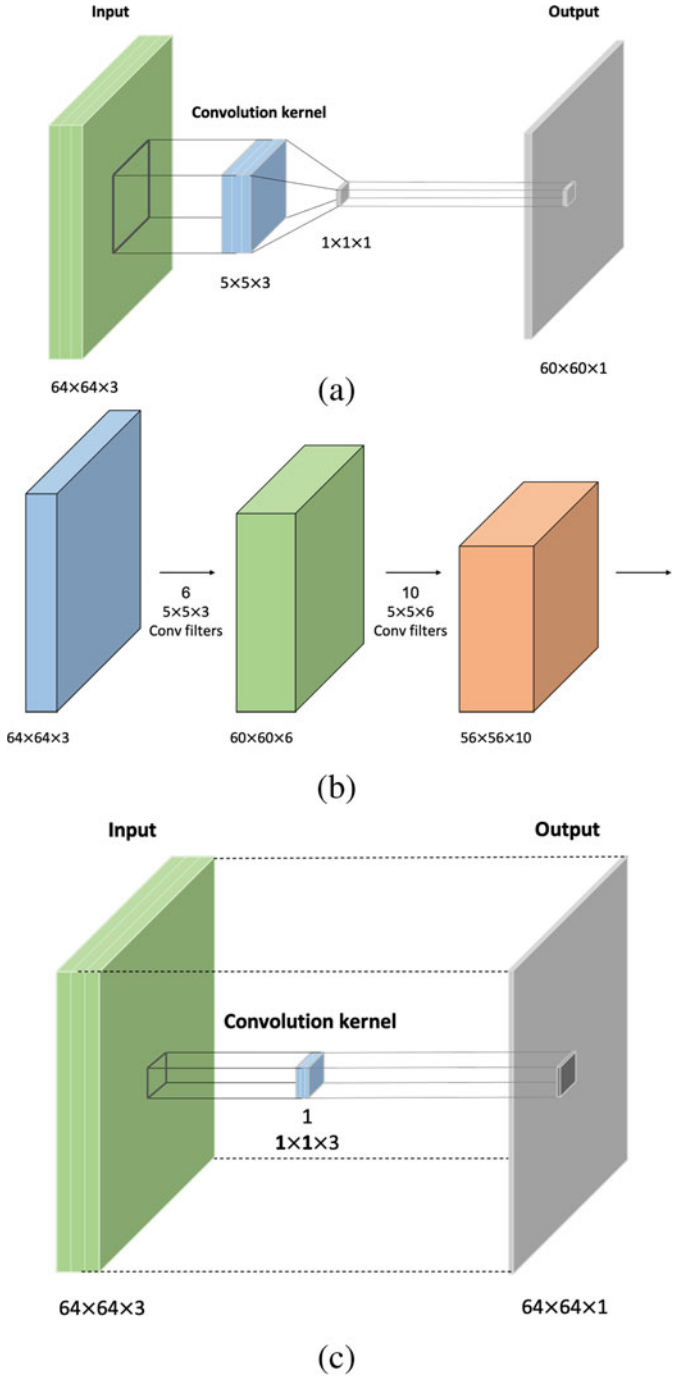

Fig. 7.7 Various convolutions used in CNNs. (**a**) Multi-input single-output (MISO) convolution, (**b**) Multi-input multi-output (MIMO) convolution, (**c**) 1×1 convolution

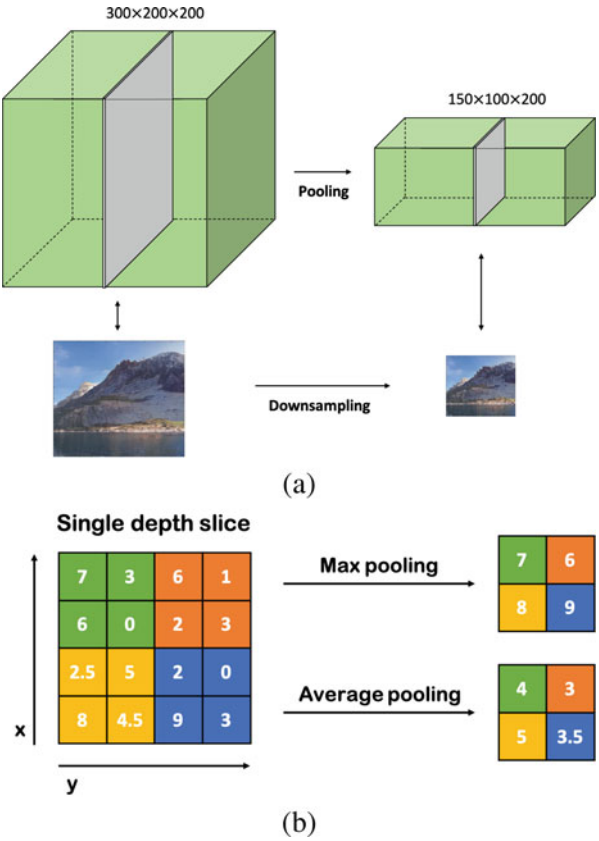

Fig. 7.8 (**a**) Pooling and unpooling operation, (**b**) max and average pooling operation

map by a factor of 2. For example, a max (average) pooling layer in Fig. 7.8b applied to an input image of 16×16 produces an output pooled feature map of 8×8.

On the other hand, unpooling is an operation for image upsampling. For example, in a narrow meaning of unpooling with respect to max pooling, one can copy the max pooled signal at the original location as shown in Fig. 7.9a. Or one could perform a transpose operation to copy all the pooled signal to the enlarged area as shown in Fig. 7.9b, which is often called the deconvolution. Regardless of the definition, unpooling tries to enlarge the downsampled image.

It was believed that a pooling layer is necessary to impose the spatial invariance in classification tasks [43]. The main ground for this claim is that small movements in the position of the feature in the input image will result in a different feature map after the convolution operation, so that spatially invariant object classification may be difficult. Therefore, downsampling to a lower resolution version of an input signal without the fine detail may be useful for the classification task by imposing invariance to translation.

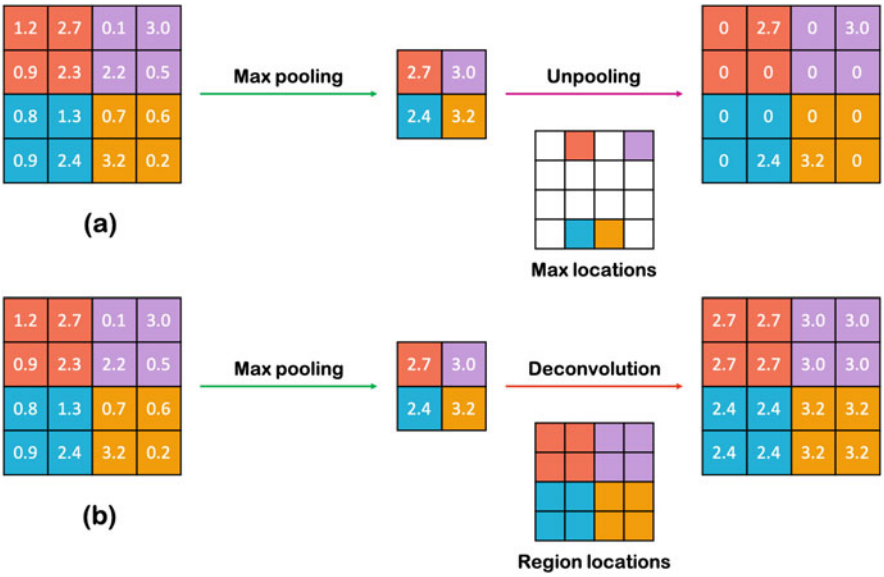

Fig. 7.9 Two ways of unpooling. (**a**) Copying to the original location (unpooling), (**b**) copying to all neighborhood (deconvolution)

However, these classical views have been challenged even by the deep learning godfather, Geoffrey Hinton. In "Ask Me Anything" column on Reddit he said, "the pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster. If the pools do not overlap, pooling loses valuable information about where things are. We need this information to detect precise relationships between the parts of an object. . . ".

Regardless of Geoffrey Hinton's controversial comment, the undeniable advantage of the pooling layer results from the increased size of the receptor field. For example, in Fig. 7.10a,b we compare the effective receptive field sizes, which determine the areas of input image affecting a specific point at the output image of a single resolution network and U-Net, respectively. We can clearly see that the receptive field size increases linearly without pooling, but can be expanded exponentially with the help of a pooling layer. In many computer vision tasks, a large receptive field size is useful to achieve better performance. So the pooling and unpooling are very effective in these applications.

Before we move on to the next topic, a remaining question is whether there exists a pooling operation which does not lose any information but increases the receptive field size exponentially. If there is, then it does address Geoffrey Hinton's concern. Fortunately, the short answer is yes, since there exists an important advance in this field from the geometric understanding of deep neural networks [40, 42]. We will cover this issue later when we investigate the mathematical principle.

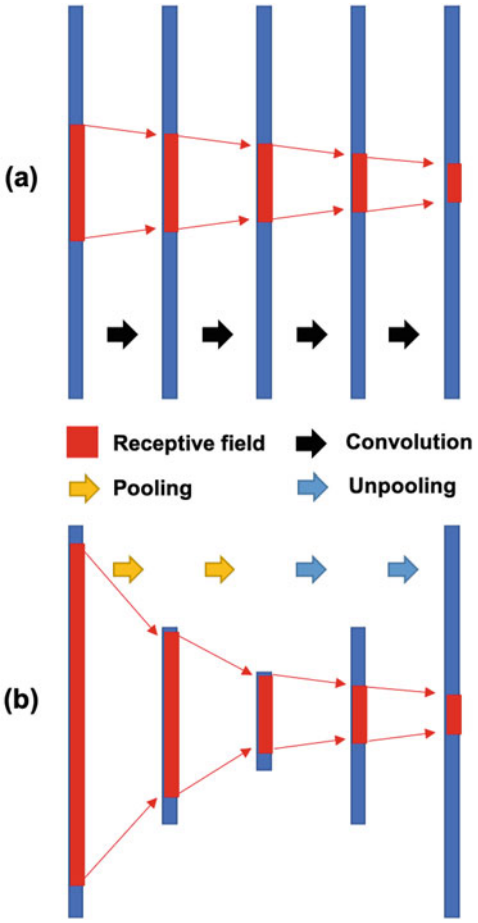

Fig. 7.10 Receptive fields of networks (**a**) without pooling layers, (**b**) with pooling layers

7.3.3 Skip Connection

Another important building block, which has been pioneered by ResNet [33] and also by U-Net [38], is the skip connection. For example, as shown in Fig. 7.11, the feature map output from the internal block is given by

$$\boldsymbol{y} = \mathcal{F}(\boldsymbol{x}) + \boldsymbol{x},$$

where $\mathcal{F}(\boldsymbol{x})$ is the output of the standard layers in the CNN with respect to the input $\boldsymbol{x}$, and the additional term $\boldsymbol{x}$ at the output comes directly from the input.

Thanks to the skipped branch, ResNet [33] can easily approximate the identity mapping, which is difficult to do using the standard CNN blocks. Later we will show that additional advantages of the skip connection come from removing local minimizers, which makes the training much more stable [35, 36].

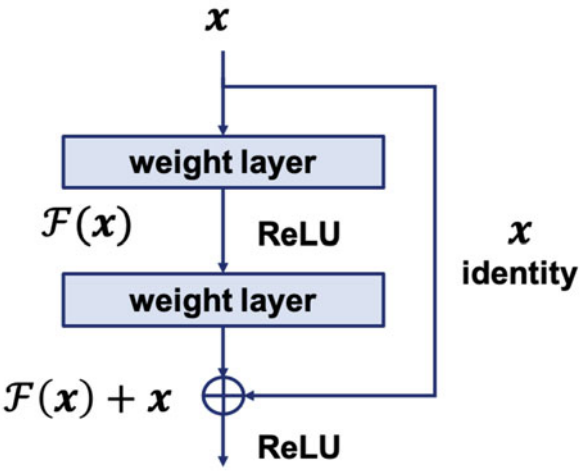

Fig. 7.11 Skip connection in ResNet

7.4 Training CNNs

7.4.1 Loss Functions

When a CNN architecture is chosen, the filter kernel should be estimated. This is usually done during a training phase by minimizing a loss function. Specifically, given input data $\boldsymbol{x}$ and its label $\boldsymbol{y} \in \mathbb{R}^m$, an average loss is defined by

$$c(\boldsymbol{\Theta}) := E[\ell\left(\boldsymbol{y}, \boldsymbol{f}_{\boldsymbol{\Theta}}(\boldsymbol{x})\right)], \tag{7.5}$$

where $E[\cdot]$ denotes the mean, $\ell(\cdot)$ is a loss function, and $\boldsymbol{f}_{\boldsymbol{\Theta}}(\boldsymbol{x})$ is a CNN with input $\boldsymbol{x}$, which is parameterized by the filter kernel parameter set $\boldsymbol{\Theta}$. In (7.5), the mean is usually taken empirically from training data.

For the multi-class classification problem using CNNs, one of the most widely used losses is the softmax loss [44]. This is a multi-class extension of the binary logistic regression classifier we studied before. A softmax classifier produces normalized class probabilities, and also has a probabilistic interpretation. Specifically, we perform the softmax transform:

$$\widehat{\boldsymbol{p}}(\boldsymbol{\Theta}) = \frac{\boldsymbol{e}^{\boldsymbol{f}_{\Theta}(\boldsymbol{x})}}{\mathbf{1}^{\top}\boldsymbol{e}^{\boldsymbol{f}_{\Theta}(\boldsymbol{x})}}, \tag{7.6}$$

where $\boldsymbol{e}^{\boldsymbol{f}_\Theta(\boldsymbol{x})}$ denotes the element-by-element application of the exponential. Then, using the softmax loss, the average loss is computed by

$$c(\boldsymbol{\Theta}) = -E\left[\sum_{i=1}^{m} y_i \log \widehat{p}_i(\boldsymbol{\Theta})\right], \tag{7.7}$$

where y_i and $\widehat{p}_i$ denote the i-th elements of $\boldsymbol{y}$ and $\widehat{\boldsymbol{p}}$, respectively. If the class label $\boldsymbol{y} \in \mathbb{R}^m$ is normalized to have probabilitistic meaning, i.e. $\mathbf{1}^\top \boldsymbol{y} = 1$, then (7.7) is indeed the cross entropy between the target class distribution and the estimated class distribution.

For the case of regression problems using CNNs, which are quite often used for image processing tasks such as denoising, the loss function is usually defined by the norm, i.e.

$$c(\boldsymbol{\Theta}) = E\|\boldsymbol{y} - \boldsymbol{f}_\Theta(\boldsymbol{x})\|_p^p \tag{7.8}$$

where $p = 1$ for the l_1 loss and $p = 2$ for the l_2 loss.

7.4.2 Data Split

In training CNNs, available data sets should be first split into three categories: training, validation, and test data sets, as shown in Fig. 7.12. The training data is also split into *mini-batches* so that each mini-batch can be used for stochastic gradient computation. The training data set is then used to estimate the CNN filter kernels, and the validation set is used to monitor whether there exists any overfitting issue in the training.

For example, Fig. 7.13a shows the example of overfitting that can be monitored during the training using the validation data. If this type of overfitting happens, several approaches should be taken to achieve stable training behavior as shown in Fig. 7.13b. Such a strategy will be discussed in the following section.

Fig. 7.12 Available data split into training, validation, and test data sets

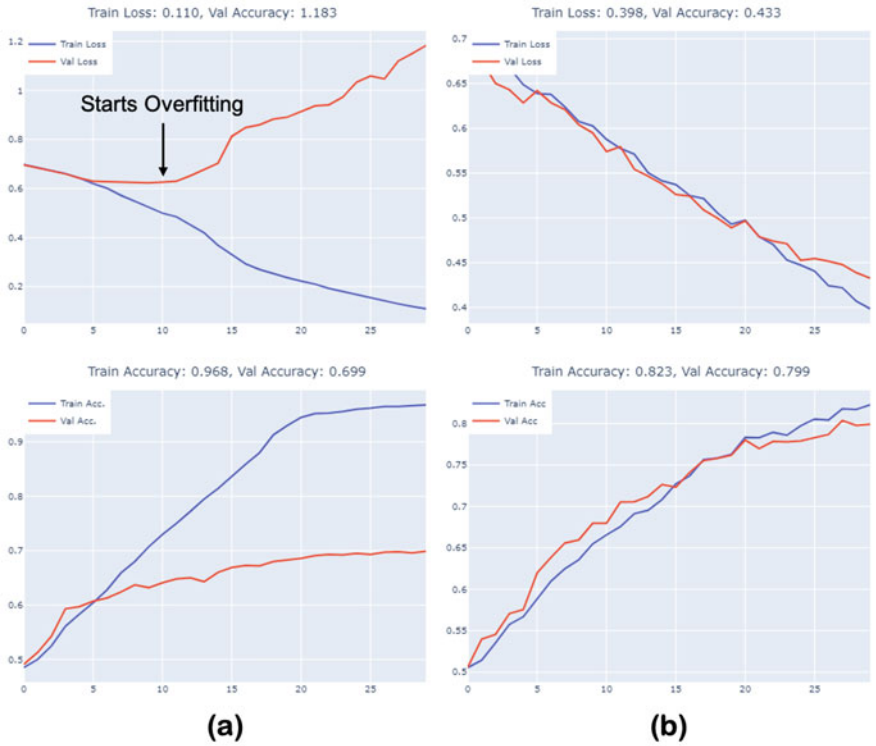

Fig. 7.13 Neural network training dynamics: (**a**) overfitting problems, (**b**) no overfitting

7.4.3 Regularization

When we observe the overfitting behaviors similar to Fig. 7.13a, the easiest solution is to increase the training data set. However, in many real-world applications, the training data are scarce. In this case, there are several ways to regularize the neural network training.

7.4.3.1 Data Augmentation

Using data augmentation we generate artificial training instances. These are new training instances created, for example, by applying geometric transformations such as mirroring, flipping, rotation, on the original image so that it doesn't change the label information.

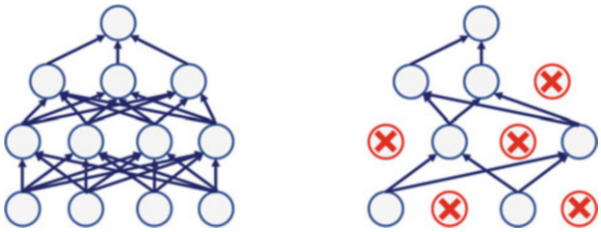

Fig. 7.14 Example of dropout

7.4.3.2 Parameter Regularization

Another way to mitigate the overfitting problem is by adding a regularization term for the original loss. For example, we can convert the loss in (7.5) to the following form:

$$c_{reg}(\boldsymbol{\Theta}) := E[\ell\left(\boldsymbol{y}, \boldsymbol{f}_{\boldsymbol{\Theta}}(\boldsymbol{x})\right)] + R(\boldsymbol{\Theta}), \tag{7.9}$$

where $R(\boldsymbol{\Theta})$ is a regularization function. Recall that similar techniques were used in the kernel machines.

7.4.3.3 Dropout

Another unique regularization used for deep learning is the dropout [45]. The idea of a dropout is relatively simple. During the training time, at each iteration, a neuron is temporarily "dropped" or disabled with probability p. This means all the inputs and outputs to some neurons will be disabled at the current iteration. The dropped-out neurons are resampled with probability p at every training step, so a dropped-out neuron at one step can be active at the next one. See Fig. 7.14. The reason that the dropout prevents overfitting is that during the random dropping, the input signal for each layer varies, resulting in additional data augmentation effects.

7.5 Visualizing CNNs

As already mentioned, hierarchical features arise in the brain during visual information processing. A similar phenomenon can be observed in the convolution neural network, once it is properly trained. In particular, VGGNet provides very intuitive information that is well correlated with the visual information processing in the brain.

For example, Fig. 7.15 illustrates the input signal that maximizes the filter response at specific channels and layers of VGGNet [31]. Remember that the filters

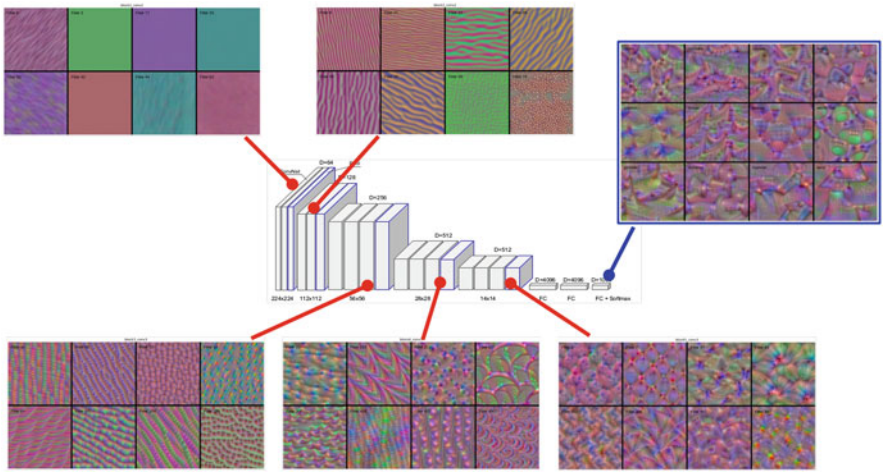

Fig. 7.15 Input images that maximize filter responses at specific channels and layers of VGGNet

are of size 3×3, so rather than visualizing the filters, an input image where this filter activates the most is displayed for specific channel and layer filters. In fact, this is similar to the Hubel and Wiesel experiments where they analyzed the input image that maximizes the neuronal activation.

Figure 7.15 shows that at the earlier layers the input signal maximizing filter response is composed of directional edges similar to the Hubel and Wiesel experiment. As we go deeper into the network, the filters build on each other and learn to code more complex patterns. Interestingly, the input images that maximize the filter response get more complicated as the depth of the layer increases. In one of the filter sets, we can see several objects in different orientations, as the particular position in the picture is not important as long as it is displayed somewhere where the filter is activated. Because of this, the filter tries to identify the object in multiple positions by encoding it in multiple places in the filter.

Finally, the blue box in Fig. 7.15 shows the input images that maximize the response on the last softmax level in the specific classes. In fact, this corresponds to the visualization of the input images that maximize the class categories. In a certain category, an object is displayed several times in the images. The emergence of the hierarchical feature from simple edges to the high-level concept is similar to visual information processing in the brain.

Finally, Fig. 7.16 visualizes the feature maps on the different levels of VGGNets in relation to a cat picture. Since the output of a convolution layer is a 3D volume, we will only visualize some of the images. As can be seen from Fig. 7.16, a feature map develops from edge-like features of the cat to information with the lower-resolution, which describes the location of the cat. In the later levels, the feature map works with a probability map in which the cat is located.

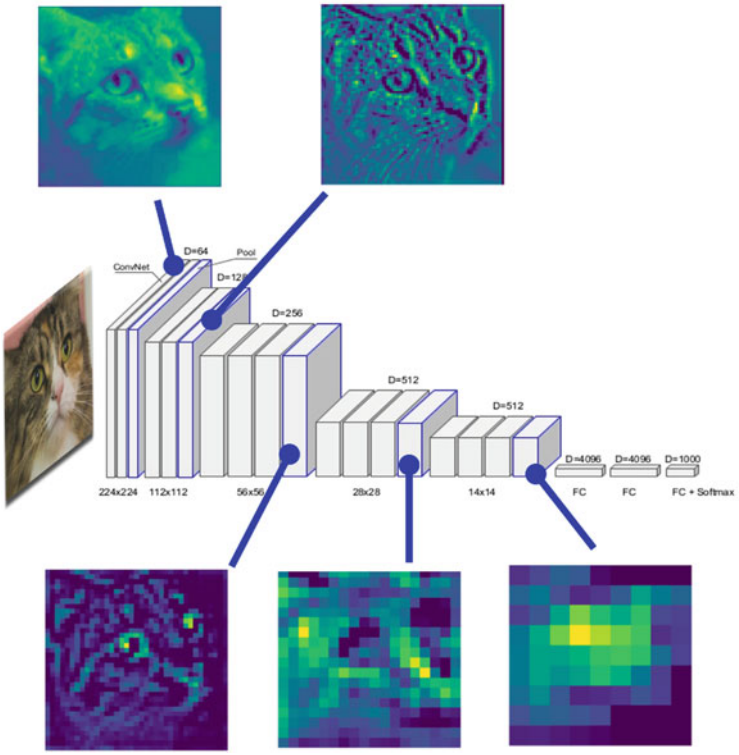

Fig. 7.16 Visualization of feature maps at several channels and layers of VGGNets when the input image is a cat

7.6 Applications of CNNs

CNN is the most widely used neural network architecture in the age of modern AI. Similar to the visual information processing in the brain, the CNN filters are trained in such a way that hierarchical features can be captured effectively. This can be one of the reasons for CNN's success with many image classification problems, low-level image processing problems, and so on.

In addition to commercial applications in unmanned vehicles, smartphones, commercial electronics, etc., another important application is in the field of medical imaging. CNN has been successfully used for disease diagnosis, image segmentation and registration, image reconstruction, etc.

For example, Fig. 7.17 shows a segmentation network architecture for cancer segmentation. Here, the label is the binary mask for cancer, and the backbone CNN is based on the U-Net architecture, where there exists a softmax layer at the end for pixel-wise classification. Then, the network is trained to classify the background

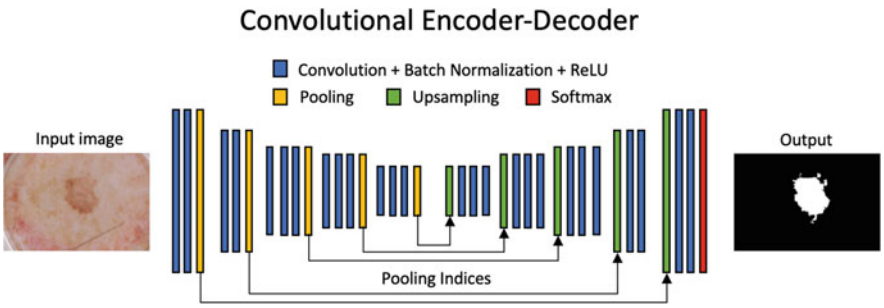

Fig. 7.17 Cancer segmentation using U-Net

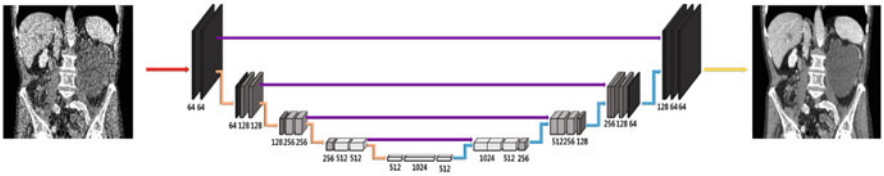

Fig. 7.18 CNN-based low-dose CT denoising

and the cancer regions. Very similar architecture can be also used for noise removal in low-dose CT images, as shown in Fig. 7.18. Instead of using the softmax layer, the network is trained with a regression loss of l_1 or l_2 using the high-quality, low-noise images as a reference. In fact, one of the amazing and also mysterious parts of deep learning is that a similar architecture works for different problems simply by changing the training data.

Because of this simplicity in designing and training CNNs, there are many exciting new startups targeting novel medical applications of AI. As the importance of global health care increases with the COVID-19 pandemic, medical imaging and general health care are undoubtedly among the most important areas of AI. Therefore, for the application of AI to health, opportunities are so numerous that we need many young, bright researchers who can invest their time and effort in AI research to improve human health care.

7.7 Exercises

1. Consider the VGGNet in Fig. 7.2. In its original implementation, the convolution kernel was 3×3.
   a. What is the total number of convolution filter sets in VGGNet?
   b. Then, what is the total number of trainable parameters in VGGNet including convolution filters and fully connected layers? (Hint: for the fully connected