3
Foundations of Deep Learning

3.1 Introduction

Machine learning is the research field of allowing computers to learn to act appropriately from sample data without being explicitly programmed. Deep learning is a class of machine learning algorithms that is built upon artificial neural networks. In fact, most of the vital building components of deep learning have existed for decades, while deep learning only gains its popularity in recent years. The idea of artificial neural networks dates back to 1940*s* when McCulloch-Pitts Neuron (McCulloch and Pitts, 1943) was first introduced. This linear model can recognize inputs from two categories by linearly aggregating information from inputs and then making the decision. Later on, the perceptron (Rosenblatt, 1958) was developed, which can learn its parameters given training samples. The research of neural networks revived in the 1980*s*. One of the breakthroughs during this period is the successful use of the back-propagation algorithm (Rumelhart et al., 1986; Le Cun and Fogelman-Soulié, 1987) to train deep neural network models. Note that the back-propagation algorithm has many predecessors dating to the 1960s and was first mentioned by Werbos to train neural networks (Werbos, 1994). The back-propagation algorithm is still the dominant algorithm to train deep models in the modern ages of deep learning. Deep learning research revived and gained unprecedented attention with the availability of "big data" and powerful computational resources in recent years. The emerging of fast GPUs allows us to train deep models with immense size while the increasingly large data ensures that these models can generalize well. These two advantages lead to the tremendous success of deep learning techniques in various research areas and result in immense real-world impact. Deep neural networks have outperformed state-of-the-art traditional methods by a large margin in multiple applications. Deep learning has significantly advanced the performance of the image recognition task. The ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) is the largest contest in image recognition, which is held each year between 2010 and 2017. In 2012, the deep convolutional neural network (CNN) won this challenge for the first time by a large margin, reducing top-5 error rate from 26.1% to 15.3% (Krizhevsky et al., 2012). Since then, the deep convolutional neural networks (CNNs) have consistently won the competition, which further pushed the error rate down to 3.57% (He et al., 2016). Deep learning has also dramatically improved the performance of speech recognition systems (Dahl et al., 2010; Deng et al., 2010; Seide et al., 2011). The introduction of deep learning techniques to speech recognition leads to a massive drop in error rates, which have stagnated for years. The deep learning techniques have massively accelerated the research field of Natural Language Processing (NLP). Recurrent Neural Networks such as LSTM (Hochreiter and Schmidhuber, 1997) have been broadly used in sequence-to-sequence tasks such as machine translation (Sutskever et al., 2014; Bahdanau et al., 2014) and dialogue systems (Vinyals and Le, 2015). As the research of "deep learning on graphs" has its root in deep learning, understanding some basic deep learning techniques is essential. Hence, in this chapter, we briefly introduce fundamental deep learning techniques, including feedforward neural networks, convolutional neural networks, recurrent neural networks, and autoencoders. They will serve as the foundations for deep learning on graphs. Though we focus on basic deep models in this chapter, we will extend our discussion to advanced deep models such as variational autoencoders and generative adversarial networks in the later chapters.

3.2 Feedforward Networks

Feedforward networks are the basis for many important deep learning techniques. A feedforward network is to approximate a certain function $f^*(\mathbf{x})$ using given data. For example, for the classification task, an ideal classifier $f^*(\mathbf{x})$ maps an input $\mathbf{x}$ to a target category y. In this case, a feedforward network is supposed to find a mapping $f(\mathbf{x}|\mathbf{\Theta})$ such that it can approximate the ideal classifier $f^*(\mathbf{x})$ well. Specifically, the goal of training the feedforward network is to learn the values of the parameters $\mathbf{\Theta}$ that can result in the best approximation to $f^*(\mathbf{x})$.

In feedforward networks, the information $\mathbf{x}$ flows from the input, through some intermediate computations, and finally to the output y. The intermediate computational operations are in the form of networks, which can typically be represented as the composition of several functions. For example, the feedforward network shown in Figure 3.1 has four functions $f^{(1)}, f^{(2)}, f^{(3)}, f^{(4)}$ connected in a chain and $f(\mathbf{x}) = f^{(4)}(f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x}))))$. In the feedforward network shown in Figure 3.1, $f^{(1)}$ is the first layer, $f^{(2)}$ is the second layer, $f^{(3)}$ is the third layer and the final layer $f^{(4)}$ is the output layer. The number of computational layers in the network defines the depth of the network. During the training of the neural network, we try to push the output $f(\mathbf{x})$ to be close to the ideal output, i.e., $f^*(\mathbf{x})$ or y. During the training process, the results from the output layer are directly guided by the training data. In contrast, all the intermediate layers do not obtain direct supervision from the training data. Thus, to approximate the ideal function $f^*(\mathbf{x})$ well, the learning algorithm decides the intermediate layers' parameters using the indirect supervision signal passing back from the output layer. Since no desired output is given for the intermediate layers from the training data during the training procedure, these intermediate layers are called hidden layers. As discussed before, each layer of

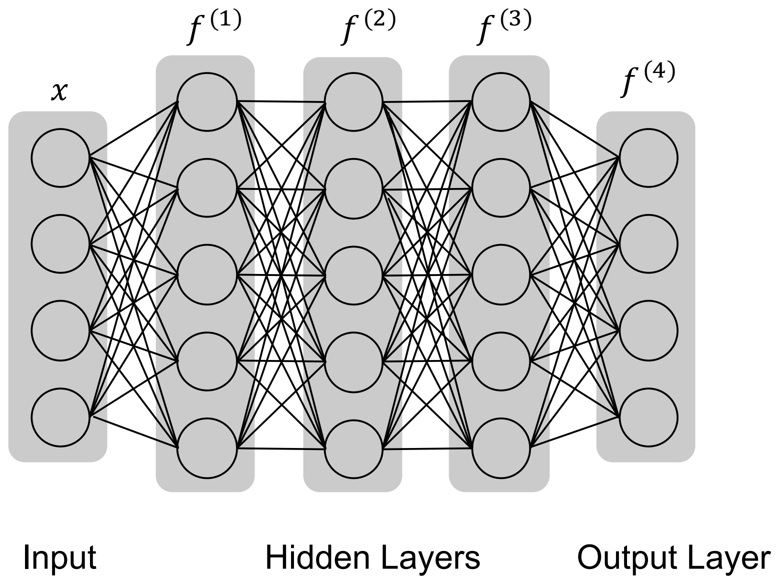


Figure 3.1 An illustrative example of feedforward networks

the neural network can be viewed as a vector-valued function, where both the input and output are vectors. The elements in the layer can be treated as nodes (or units). Thus, each layer can be considered as a set of vector-to-scalar functions where each node is a function. The networks are called neural networks as Neuroscience loosely inspires them. The operation in a node loosely mimics what happens on a neuron in the brain, activated when it encounters sufficient stimuli. A node gathers and transforms information from all the nodes in the previous layer and then passes it through an activation function, which determines to what extent the information can pass through to the next layer. The operation of gathering and transforming information is typically linear, while the activation function adds non-linearity to the neural network, which largely improves its approximation capability.

3.2.1 The Architecture

In a fully connected feedforward neural network, nodes in consecutive layers form a complete bipartite graph, i.e., a node in one layer is connected to all nodes in the other layer. A general view of this architecture is demonstrated in Figure 3.1. Next we introduce the details of the computation involved in the neural network. To start, we focus on a single node in the first layer. The input of the neural network is a vector $\mathbf{x}$ where we use $\mathbf{x}_i$ to denote its i-th element. All these elements can be viewed as nodes in the input layer. A node in the second layer (or the one after the input layer) is connected to all the nodes in the input layer. These connections between the nodes in the input layer and an arbitrary node in the second layer are illustrated in Figure 3.2. The operations in one node consist of two parts: 1) combining the elements of

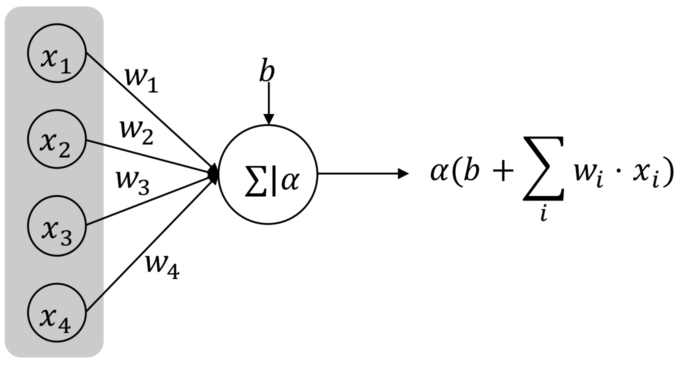


Figure 3.2 Operations in one node

the input linearly with various weights (or $\mathbf{w}_i$); 2) passing the value obtained in the previous step through an activation function. Mathematically, it can be represented as

$$h = \alpha(b + \sum_{i=1}^{4} \mathbf{w}_i \cdot \mathbf{x}_i),$$

where b is a bias term and $\alpha()$ is an activation function, which will be introduced in next section. We now generalize the operation to an arbitrary hidden layer. Assume that in the k-th layer of the neural network, we have $N^{(k)}$ nodes and the output of the layer can be represented as $\mathbf{h}^{(k)}$ with $\mathbf{h}_i^{(k)}$ denoting its i-th element. Then, to compute $\mathbf{h}_j^{(k+1)}$ in the $(k+1)-$th layer, the following operation is conducted:

$$\mathbf{h}_j^{(k+1)} = \alpha(b_j^{(k)} + \sum_{i=1}^{N^{(k)}} \mathbf{W}_{ji}^{(k)} \mathbf{h}_i^{(k)}). \tag{3.1}$$

Note that we use $\mathbf{W}_{ji}^{(k)}$ to denote the weight corresponding to the connection between $\mathbf{h}_i^{(k)}$ and $\mathbf{h}_j^{(k+1)}$ and $b_j^{(k)}$ is the bias term for calculating $\mathbf{h}_j^{(k+1)}$. The operations to calculate all the elements in the $(k+1)$-th layer can be summarized in the matrix form as:

$$\mathbf{h}^{(k+1)} = \alpha(\mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k)}),$$

where $\mathbf{W}^{(k)} \in \mathbb{R}^{N^{(k+1)} \times N^{(k)}}$ contains all weights and its j, i-th element is $\mathbf{W}_{ji}^{(k)}$ in Eq. (3.1) and $\mathbf{b}^{(k)}$ consists of all bias terms. Specifically, for the input layer, we have $\mathbf{h}^{(0)} = \mathbf{x}$. Recall that we use $f^{(k+1)}$ to represent the operation of the $(k+1)$-th layer in the neural network; thus we have

$$\mathbf{h}^{(k+1)} = f^{(k+1)}(\mathbf{h}^{(k)}) = \alpha(\mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k)}).$$

Note that the introduced operations are typical for hidden layers. The output layer usually adopts a similar structure, but different activation functions to transform the obtained information. We next introduce activation functions and the design of the output layer.

3.2.2 Activation Functions

An activation function decides whether or to what extent the input signal should pass. The node (or neuron) is activated if there is information passing through it. As introduced in the previous section, the operations of a neural network are linear without activation functions. The activation function introduces the nonlinearity into the neural network that can improve its approximation capability. In the following, we introduce some commonly used activation functions.

Rectifier

Rectifier is one of the most commonly used activation functions. As shown in Figure 3.3, it is similar to linear functions, and the only difference is that the rectifier outputs 0 on the negative half of its domain. In the neural network, the units employed this activation function are called as Rectifier Linear Units (ReLUs). The rectifier activation function is linear (identity) for all positive input values and 0 for all negative values. Mathematically, it is defined as:

$$\text{ReLU}(z) = \max\{0, z\}.$$

At each layer, only a few of the units are activated, which ensures efficient computation. One drawback of the rectified linear unit is that its gradient is 0 on the negative half of the domain. Hence if the unit is not activated, no supervision information can be passed back for training that unit. Some generalizations of ReLU have been proposed to overcome this drawback. Instead of setting the negative input to 0, leaky ReLU (Maas et al., 2013) performs a linear transformation with a small slope to the negative values as shown in Figure 3.4a. More specifically, the leaky ReLU can be mathematically represented as:

$$\text{LeakyReLU}(z) = \begin{cases} 0.01z & z < 0 \\ z & z \geq 0, \end{cases}$$

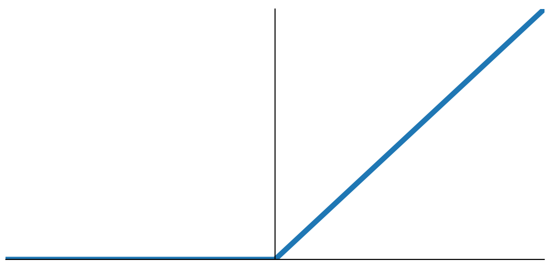


Figure 3.3 ReLU

Another generalization of ReLU is the exponential linear unit (ELU). It still has the identity transform for the positive values but it adopts an exponential transform for the negative values as shown in Figure 3.4b. Mathematically, the ELU activation function can be represented as:

$$\text{ELU}(z) = \begin{cases} c \cdot \exp(z - 1) & z < 0 \\ z & z \geq 0, \end{cases}$$

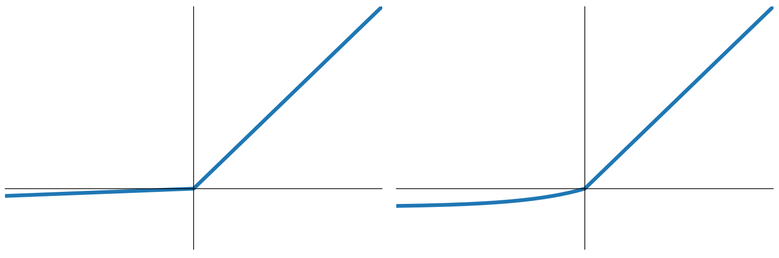

(a) Leaky ReLU (b) ELU

Figure 3.4 Generalizations of ReLU

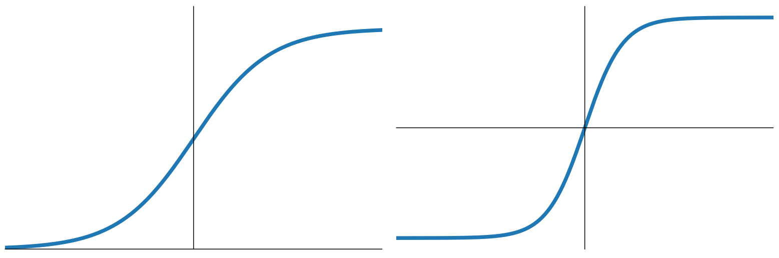

(a) Logistic Sigmoid (b) Tanh

Figure 3.5 Logistic sigmoid and hyperbolic tangent

where c is a positive constant controlling the slope of the exponential function for the negative values.

Logistic Sigmoid and Hyperbolic Tangent

Prior to the ReLU, logistic sigmoid and the hyperbolic tangent functions are the most commonly adopted activation functions. The sigmoid activation function can be mathematically represented as follows:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}.$$

As shown in Figure 3.5a, the sigmoid function maps the input into the range of 0 to 1. Specifically, the more negative the input is, the closer the output is to 0 and the more positive the input is, the closer the output is to 1.

The hyperbolic tangent activation function is highly related to the sigmoid function with the following relation:

$$\tanh(z) = \frac{2}{1 + \exp(-2z)} - 1 = 2 \cdot \sigma(2z) - 1.$$

As shown in Figure 3.5b, the hyperbolic tangent function maps the input into the range of -1 to 1. Specifically, the more negative the input is, the closer the output is to -1 and the more positive the input is, the closer the output is to 1.

These two activation functions face the same saturation issue (Nwankpa et al., 2018). They saturate when the input z is a huge positive number or a very negative number. They are only sensitive to values that are close to 0. The phenomenon of the widespread saturation makes gradient-based training very difficult, as the gradient will be around 0 when z is either very positive or very negative. For this reason, these two activation functions are becoming less popular in feedforward networks.

3.2.3 Output Layer and Loss Function

The choice of the output layer and loss function varies according to the applications. Next, we introduce some commonly used output units and loss functions.

In regression tasks, a neural network needs to output continuous values. A simple way to achieve this is to perform an affine transformation (or an affinity) without the non-linear activation. Given input features (or features from previous layers) $\mathbf{h} \in \mathbb{R}^{d_{in}}$, a layer of linear units outputs a vector $\hat{\mathbf{y}} \in \mathbb{R}^{d_{ou}}$ as:

$$\hat{\boldsymbol{y}} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b},$$

where $\mathbf{W} \in \mathbb{R}^{d_{ou} \times d_{in}}$ and $\mathbf{b} \in \mathbb{R}^{d_{ou}}$ are the parameters to be learned. For a single sample, we can use a simple squared loss function to measure the difference between the predicted value $\hat{\mathbf{y}}$ and the ground truth $\mathbf{y}$ as follows:

$$\ell(\mathbf{y}, \hat{\mathbf{y}}) = (\mathbf{y} - \hat{\mathbf{y}})^2.$$

For classification tasks, the neural network needs to tell the classes of given samples. Instead of directly producing a discrete output indicating the predicted labels of a given sample, we usually produce a discrete probability distribution over the labels. Different output layers and loss functions are used depending on whether the prediction is binary or multi-way. Next, we discuss the details in these two scenarios.

Binary Targets

For the binary classification task, we assume that a sample is labeled as either 0 or 1. Then, to perform the prediction, we first need a linear layer to project the input (results from previous layers) into a single value. Following this, a sigmoid function is applied to map this value into the range of 0 to 1, which indicates the probability of the sample being predicted as label 1. In summary, this process can be modeled as:

$$\hat{y} = \sigma(\mathbf{W}\mathbf{h} + b),$$

where $\mathbf{h} \in \mathbb{R}^{d_{in}}$ and $\mathbf{W} \in \mathbb{R}^{1\times d_{in}}$. Specifically, $\hat{y}$ denotes the probability of predicting the input sample with label 1 while $1 - \hat{y}$ indicates the probability for label 0. With the output $\hat{y}$, we can employ the cross-entropy loss to measure the difference between the ground truth and the prediction as:

$$\ell(y, \hat{y}) = -y \cdot \log(\hat{y}) - (1 - y) \cdot \log(1 - \hat{y}).$$

During the inference, an input sample is predicted with label 1 if the predicted $\hat{y} > 0.5$ and with label 0, otherwise.

Categorical Targets

For the n-class classification task, we assume that the ground truth is denoted as integers between 0 and $n - 1$. Thus, we use a one-hot vector $\mathbf{y} \in \{0, 1\}^n$ to indicate the label where $\mathbf{y}_i = 1$ indicates that the sample is labeled as $i - 1$. To perform the prediction, we first need a linear layer to transform the input $\mathbf{h}$ to a n-dimensional vector $\mathbf{z} \in \mathbb{R}^n$ as:

$$\mathbf{z} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b},$$

where $\mathbf{W} \in \mathbb{R}^{n\times d_{in}}$ and $\mathbf{b} \in \mathbb{R}^n$. We then apply the softmax function to normalize $\mathbf{z}$ into a discrete probability distribution over the classes as:

$$\hat{\mathbf{y}}_i = \text{softmax}(z)_i = \frac{\exp(\mathbf{z}_i)}{\sum_j \exp\left(\mathbf{z}_j\right)}, i = 1, \ldots, n$$

where $\mathbf{z}_i$ denotes the i-th element of the vector $\mathbf{z}$ while $\hat{\mathbf{y}}_i$ is the i-th element of the output of the softmax function. Specifically, $\hat{\mathbf{y}}_i$ indicates the probability of the input sample being predicted with label $i - 1$. With the predicted $\hat{\mathbf{y}}$, we can employ the cross-entropy loss to measure the difference between the ground truth and the prediction as:

$$\ell(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=0}^{n-1} \mathbf{y}_i \log(\hat{\mathbf{y}}_i).$$

During the inference, an input sample is predicted with label $i - 1$ if $\hat{\mathbf{y}}_i$ is the largest among all output units.

3.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a popular family of neural networks. They are best known for processing regular grid-like data such as images. They are similar to the feedforward neural networks in many aspects. They also consist of neurons that have trainable weights and bias. Each neuron receives and transforms some information from previous layers. The difference is that some of the neurons in CNNs may have different designs from the ones we introduced for feedforward networks. More specifically, the convolution operation is proposed to design some of the neurons. The layers with the convolution operation are called the convolutional layers. The convolution operation typically only involves a small number of neurons in the previous layers, which enforces sparse connections between layers. Another vital operation in CNNs is the pooling operation, which summarizes the output of nearby neurons as the new output. The layers consist of the pooling operations are called the pooling layers. In this section, we first introduce the convolution operation and convolutional layers, then discuss the pooling layers and finally present an overall framework of CNNs.

3.3.1 The Convolution Operation and Convolutional Layer

In general, the convolution operation is a mathematical operation on two real functions to produce a third function (Widder and Hirschman, 2015). The convolution operation between two functions $f()$ and $g()$ can be defined as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau.$$

As an example of motivation, let us consider a continuous signal $f(t)$, where t denotes time and $f(t)$ is the corresponding value at time t. Suppose that the signal is somewhat noisy. To obtain a less noisy signal, we would like to average the value at time t with its nearby values. Furthermore, values corresponding to time that is closer to t may be more similar to that at time t and thus they should contribute more. Hence, we would like to take a weighted average over a few values that are close to time t as its new value. This can be modeled as a convolution operation between the signal $f(t)$ and a weight function $w(c)$, where c represents the closeness to the target t. The smaller c is, the larger the value of $w(c)$ is. The signal after the convolution operation can be represented as:

$$s(t) = (f * w)(t) = \int_{-\infty}^{\infty} f(\tau)w(t-\tau)d\tau.$$

Note that to ensure that the operation does a weighted average, $w(c)$ is constrained to integrate to 1, which makes $w(c)$ a probability density function. In general, the convolution operation is not necessary to be a weighted average operation and the function $w(t)$ does not need to meet these requirements.

In practice, data is usually discrete with fixed intervals. For example, the signal $f(t)$ may only be sampled at integer values of time t. Assuming $f()$ and $w()$ in the previous example are both defined on integer values of time t, then the convolution can be written as:

$$s(t) = (f * w)(t) = \sum_{\tau=-\infty}^{\infty} f(\tau)w(t-\tau).$$

We further consider that in most of the cases, the function $w()$ is only non-zero within a small window. In other words, only local information contributes to the new value of a target position. Suppose that the window size is $2n + 1$, i.e., $w(c) = 0$ for $c < n$ and $c > n$, then the convolution can be further modified as:

$$(f * w)(t) = \sum_{\tau=t-n}^{t+n} f(\tau)w(t-\tau).$$

In the case of neural networks, t can be considered as the indices of the units in the input layer. The function $w()$ is called as a kernel or a filter. The convolution operation can be represented as a sparsely connected graph. The convolutional layers can be explained as sliding the kernel over the input layer and calculating the output correspondingly. An example of the layers consisting of the convolution operation can be found in Figure 3.6.

Example 3.1 Figure 3.6 shows a convolutional layer, where the input and output have the same size. To maintain the size of the output layer, the input layer is padded with two additional units (the dashed circles) with a value 0. The kernel of the convolution operation is shown on the right of the figure. For simplicity, the nonlinear activation function is not shown in the figure. In this example, $n = 1$, and the kernel function is defined only at 3 nearby locations.

In the practical machine learning scenario, we often deal with data with more than one dimension such as images. The convolution operation can be extended to data with high dimensions. For example, for a 2-dimensional image I, the convolution operation can be performed with a 2-dimensional kernel K as:

$$S(i, j) = (I * K)(i, j) = \sum_{\tau=i-n}^{i+n} \sum_{j=\gamma-n}^{\gamma+n} I(\tau, \gamma)K(i-\tau, j-\gamma).$$

Next, we discuss some key properties of the convolutional layer. Without the loss of generality, we consider the convolutional layer for single-dimensional

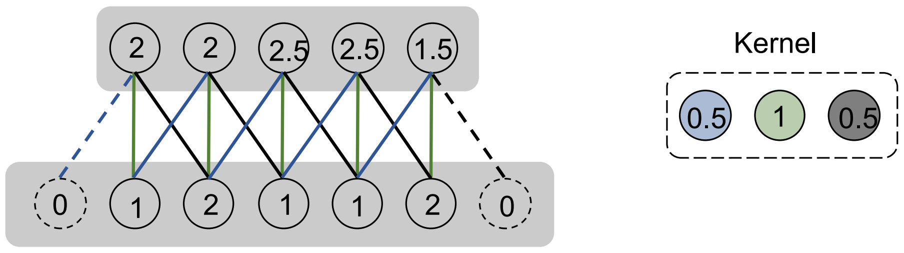
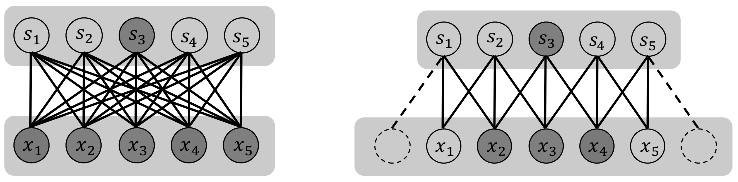


Figure 3.6 An example of a convolutional layer

data. These properties can also be applied to high dimensional data. Convolutional layers mainly have three key properties including *sparse connections, parameter sharing* and *equivariant representation*.

Sparse Connection


s_1 s_2 s_3 s_4 s_5
x_1 x_2 x_3 x_4 x_5
s_1 s_2 s_3 s_4 s_5
x_1 x_2 x_3 x_4 x_5

(a) Traditional neural network layers (b) Convolutional neural network layers

Figure 3.7 Dense and sparse connectivity

In traditional neural network layers, the interactions between the input and the output units can be described by a matrix. Each element of this matrix defines an independent parameter for the interaction between each input unit and each output unit. However, the convolutional layers usually have sparse connections between layers when the kernel is only non-zero on a limited number of input units. A comparison between the traditional neural network layers and the convolutional neural network layers is demonstrated in Figure 3.7. In this figure, we highlight one output unit S_3 and the corresponding input units that affect S_3. Clearly, in the densely connected layers, a single output unit is affected by all the input units. However, in the convolutional neural network layers, the output unit S_3 is only affected by 3 input units $\mathbf{x}_1, \mathbf{x}_2$ and $\mathbf{x}_3$, which are called as the *receptive field* of S_3. One of the major advantages of the sparse connectivity is that it can largely improve the computational efficiency. If there are N input and M output units, there are $N \times M$ parameters in the traditional neural network layers. The time complexity for a single computation pass of this layer is $O(N \times M)$. While the convolutional layers with the same number of input and output units only have $K \times M$ parameters (we do not consider parameter sharing here, which will be discussed in the next subsection), when its kernel size is K. Correspondingly, the time complexity is reduced to $O(K \times M)$. Typically, the kernel size K is much smaller than the number of input units N. In other words, the computation of convolutional neural networks is much more efficient than that of traditional neural networks.

Parameters Sharing

As aforementioned, there are $K \times M$ parameters in the convolutional layers. However, this number can be further reduced due to *parameter sharing* in the convolutional layers. *Parameter sharing* refers to sharing the same set of parameters when performing the calculation for different output units. In the convolutional layers, the same kernel is used to calculate the values of all the output units. This process naturally leads to parameter sharing. An illustrative example is shown in Figure 3.8, where we use colors to denote different parameters. In this example, we have a kernel size of 3, which results in 3 parameters. In general, for convolutional layers with the kernel size of K, there are K parameters. Comparing with $N \times M$ parameters in the traditional neural network layers, K is much smaller, and consequently, the requirement for memory is much lower.

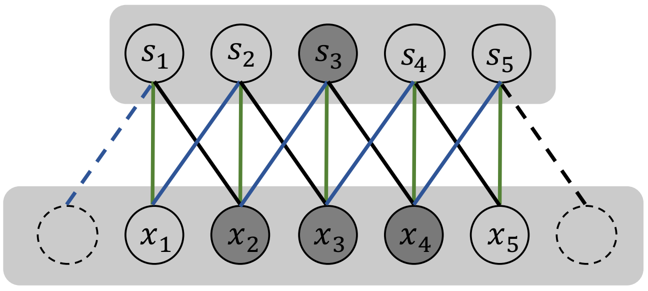


Figure 3.8 Parameter sharing

Equivariant Representation

The parameter sharing mechanism naturally introduces another important property of CNNs, called as *equivariant* to translation. A function is said to be equivariant if the output changes in the same way as the input changes. More specifically, a function $f()$ is equivariant to another function $g()$ if $f(g(x)) = g(f(x))$. In the case of the convolution operation, it is not difficult to verify that it is equivariant to translation functions such as shifts. For example, if we shift the input units in Figure 3.8 to the right by 1 unit, we can still find the same output pattern that is also shifted to the right by 1 unit. This property is important in many applications where we care more about whether a certain feature appears than where it appears. For example, when recognizing whether an image contains a cat or not, we care whether there are some important features in the image indicating the existence of a cat instead of where these features locate in the image. The property of *equivariant* to translation of CNNs is crucial to their success in the area of image classification (Krizhevsky et al., 2012; He et al., 2016).

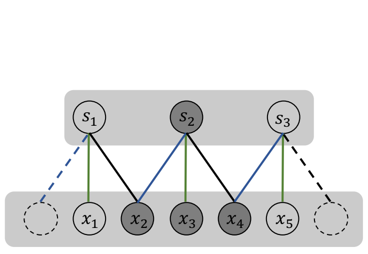
3.3.2 Convolutional Layers in Practice

In practice, when we discuss convolution in CNNs, we do not refer to the exact convolution operation as it is defined mathematically. The convolutional layers used in practice differ slightly from the definition. Typically, the input is not only a grid of real values. Instead, it is a grid of vector-valued input. For example, in a colored image consisting of $N \times N$ pixels, three values are associated with each pixel, representing red, green, and blue, respectively. Each color denotes a *channel* of the input image. Generally, the i-th *channel* of the input image consists of the i-th element of the vectors at all positions of the input. The length of the vector at each position (e.g., pixel in the case of image) is the number of *channels*. Hence, the convolution typically involves three dimensions, while it only "slides" in two dimensions (i.e., it does not slide in the dimension of channels). Furthermore, in typical convolutional layers, multiple distinct kernels are applied in parallel to extract features from the input layer. Consequently, the output layer is also multi-channel, where the results for each kernel correspond to each output channel. Let us consider an input image I with L channels. The convolution operation with P kernels can be formulated as:

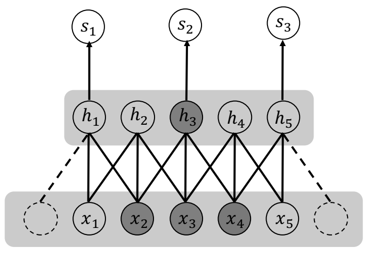
$$S(i, j, p) = (I * K_p)(i, j) = \sum_{l=1}^{L} \sum_{\tau=i-n}^{i+n} \sum_{j=\gamma-n}^{\gamma+n} I(\tau, \gamma, l) K_p(i-\tau, j-\gamma, l), p = 1, \dots P \tag{3.2}$$

where K_p is the p-th kernel with $(2n + 1)^2 \cdot L$ parameters. The output clearly consists of P channels.

In many cases, to further reduce the computation complexity, we can regularly skip some positions when sliding the kernel over the input. The convolution can be only performed every s positions, where the number s is usually



(a) Strided convolutions



(b) Convolutions with downsampling

Figure 3.9 Strided convolutions can be viewed as convolutions with downsampling.

called the *stride*. We call the convolutions with stride as strided convolutions. An illustrative example of strided convolutions is illustrated in Figure 3.9a, where the stride is $s = 2$. The strided convolution can be also viewed as a downsampling over the results of the regular convolution as shown in Figure 3.9b. The strided convolution with stride s can be represented as:

$$S(i, j, p) = \sum_{l=1}^{L} \sum_{\tau=i-n}^{i+n} \sum_{j=\gamma-n}^{\gamma+n} I(\tau, \gamma, l) K_p((i-1)\cdot s + 1 - \tau, (j-1)\cdot s + 1 - \gamma, l).$$

When stride is $s = 1$, the strided convolution operation is equivalent to the non-strided convolution operation as described in Eq. (3.2). As mentioned before, zero padding is usually applied to the input to maintain the size of the output. The size of padding, the size of receptive field (or the size of kernel) and the stride determine the size of the output when the input size is fixed. More specifically, consider a 1-D input with size N. Suppose that the padding size is Q, the size of the receptive field is F and the size of stride is s, the size of the output O can be calculated with the following formulation:

$$O = \frac{N - F + 2Q}{s} + 1. \tag{3.3}$$

Example 3.2 The input size of the strided convolution shown in Figure 3.9a is $N = 5$. Its kernel size is $F = 3$. Clearly, the size of zero-padding is $Q = 1$. Together with stride $s = 2$, we can calculate the output size using Eq. (3.3):

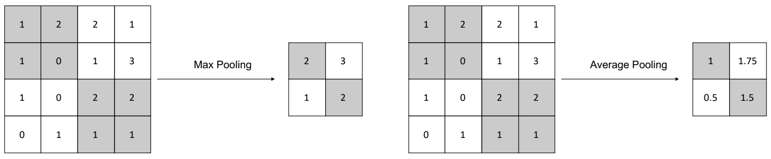
$$O = \frac{N - F + 2Q}{s} + 1 = \frac{5 - 3 + 2 \times 1}{2} + 1 = 3.$$

3.3.3 Non-linear Activation Layer

Similar to feedforward neural networks, nonlinear activation is applied to every unit after the convolution operation. The activation function widely used in CNNs is the ReLU. The process of applying the non-linear activation is also called the *detector* stage or the *detector* layer.

3.3.4 Pooling Layer

A *pooling* layer is usually followed after the convolution layer and the detector layer. The pooling function summarizes the statistic of a local neighborhood to denote this neighborhood in the resulting output. Hence, the width and height of the data is reduced after the pooling layer. However, the depth (the number of channels) of the data does not change. The commonly used pooling operations include max pooling and average pooling as demonstrated in Figure 3.10. These pooling operations take a 2×2 local neighborhood as input and produce a single value based on them. As the names indicate, the max pooling operation takes the maximum value in the local neighborhood as the output while the average pooling takes the average value of the local neighborhood as its output.



(a) Max pooling (b) Average pooling

Figure 3.10 Pooling methods in CNNs

3.3.5 An Overall CNN Framework

With the convolution and pooling operations introduced, we now introduce an overall framework of convolutional neural networks with classification as the downstream task. As shown in Figure 3.11, the overall framework for classification can be roughly split into two components – the feature extraction component and the classification component. The feature extraction component, which consists of convolution and pooling layers, extracts features from the input. While the classification component is built upon fully connected

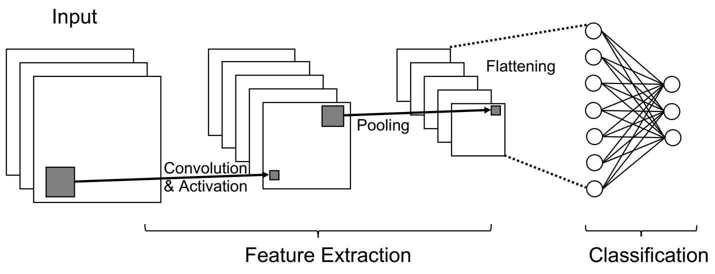


Figure 3.11 An overall framework of convolutional neural networks

feedforward neural networks. A flattening operation connects these two components. It flattens the feature matrices in multiple channels extracted by the feature extraction component to a single feature vector, which is served as the input to the classification component. Note that in Figure 3.11, only a single convolutional layer, and a single pooling layer are illustrated. However, in practice, we usually stack multiple convolutional and pooling layers. Similarly, in the classification component, the feedforward neural networks can consist of multiple fully connected layers.

3.4 Recurrent Neural Networks

Many tasks, such as speech recognition, machine translation, and sentiment classification, need to handle sequential data, where each data sample is represented as a sequence of values. Given a sentence (a sequence of words) in one language, machine translation aims to translate it into another language. Thus, both the input and output are sequences. Sentiment classification predicts the sentiment of a given sentence or document where the input is a sequence, and the output is a value to indicate the sentiment class. We may try to use standard neural network models to deal with sequential data, where each element in the sequence can be viewed as an input unit in the input layer. However, this strategy is not sufficient for sequential data due to two main reasons. First, standard network models often have fixed input and output size; however, sequences (either input or output) can have different lengths for different data samples. Second and more importantly, standard network models do not share parameters to deal with input from different positions of the sequence. For example, in language-related tasks, given two sentences of "I went to the Yellow Stone National park last summer" and "Last summer, I went to the Yellow Stone National park", we expect the model to figure out that the time is "last summer" in both sentences, although it appears in different positions of the sentences. A natural way to achieve this is the idea of parameter sharing as similar to CNNs. The recurrent neural networks (RNNs) have been introduced to solve these two challenges. RNNs are to recurrently apply the same functions to each element of the sequence one by one. Since all positions in the sequence are processed using the same functions, parameter sharing is naturally realized among different positions. Meanwhile, the same functions can be repeatably applied regardless of the length of the sequence, which can inherently handle sequences with varied lengths.

3.4.1 The Architecture of Traditional RNNs

A sequence with the length n can be denoted as $(\mathbf{x^{(1)}}, \mathbf{x^{(2)}}, \ldots, \mathbf{x^{(n)}})$. As shown in Figure 3.12, the traditional RNN model takes one element of the sequence at a time and processes it with a block of neural networks. The block of neural networks often takes not only the element but also the information flowed from the previous block as input. As a result, the information in the early positions of the sequence can flow through the entire sequence. The blocks of neural networks are identical. The RNN model in Figure 3.12 has an output $\mathbf{y}^{(i)}$ at each position i, which is not mandatory for RNN models.

The block of neural networks has two inputs and also produces two outputs. We use $\mathbf{y}^{(i)}$ to denote the output and $\mathbf{h}^{(i)}$ to denote the information flowing to the next position. To process the first element, $\mathbf{h}^{(0)}$ is often initialized as $\mathbf{0}$. The procedure for dealing with the i-th element can be formulated as:

$$\begin{aligned}\mathbf{h}^{(i)} &= \alpha_h(\mathbf{W}_{hh} \cdot \mathbf{h}^{(i-1)} + \mathbf{W}_{hx}\mathbf{x}^{(i-1)} + \mathbf{b}_h)\\ \mathbf{y}^{(i)} &= \alpha_y(\mathbf{W}_{yh}\mathbf{h}^{(i)} + \mathbf{b}_y),\end{aligned}$$

where $\mathbf{W}_{hh}$, $\mathbf{W}_{hx}$, and $\mathbf{W}_{yh}$ are the matrices to perform linear transformations; $\mathbf{b}_h$ and $\mathbf{b}_y$ are the bias terms; and $\alpha_h()$ and $\alpha_y()$ are two activation functions.

When dealing with sequential data, it is crucial to capture the long-term dependency in the sequence. For example, in language modeling, two words that appear far away in the sentence can be tightly related. However, it turns out that the traditional RNN model is not good at capturing long-term dependency. The main issue is that the gradients propagated over many stages tend to either vanish or explode. Both phenomenons cause problems for the training procedure. The gradient explosion will damage the optimization process, while the vanishing gradient makes the guidance information in the later positions challenging to affect the computations in the earlier positions. To solve these issues, gated RNN models have been proposed. The Long short-term

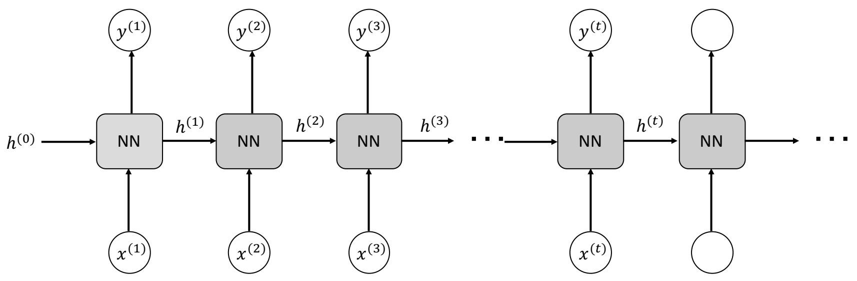


Figure 3.12 The architecture of traditional RNNs

memory (LSTM) (Hochreiter and Schmidhuber, 1997) and gated recurrent unit (GRU) (Cho et al., 2014a) are two representative gated RNN models.

3.4.2 Long Short-Term Memory

The overall structure of the LSTM is the same as that of the traditional RNN model. It also has the chain structure with identical neural network blocks applying to the elements of the sequence. The key difference is that a set of gating units are utilized to control the information flow in LSTM. As shown in Figure 3.13, the information flowing through consecutive positions in a sequence includes the *cell state* $\mathbf{C}^{(t-1)}$ and the *hidden state* $\mathbf{h}^{(t-1)}$. The cell state serves as the information from the previous states that are propagated to the next position, and the hidden state helps decide how the information should be propagated. The hidden state $\mathbf{h}^{(t)}$ also serves as the output of this position if necessary e.q., in sequence to sequence applications .

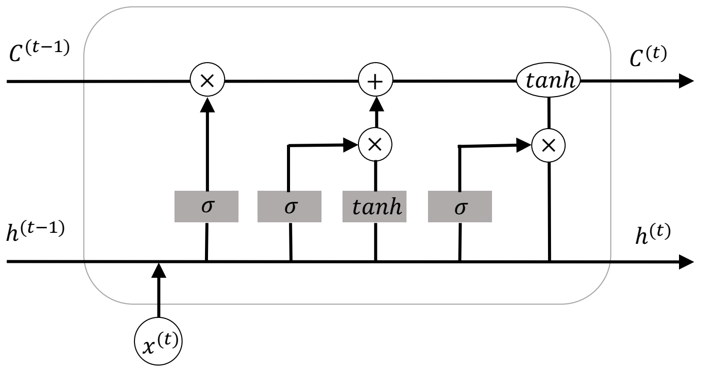


Figure 3.13 A block of LSTM

The first step of the LSTM is to decide what information from previous cell state we are going to discard. The decision is made by a *forget gate*. The forget gate considers the previous hidden state $\mathbf{h}^{(t-1)}$ and the new input $\mathbf{x}^{(t)}$ and outputs a value between 0 to 1 for each of the elements in the cell state $\mathbf{C}^{(t-1)}$. The corresponding value of each element controls how the information in each element is discarded. The outputs can be summarized as a vector $\mathbf{f}_t$, which has the same dimension as the cell state $\mathbf{C}^{(t-1)}$. More specifically, the forget gate can be formulated as:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \cdot \mathbf{x}^{(t)} + \mathbf{U}_f \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_f),$$

where $\mathbf{W}_f$ and $\mathbf{U}_f$ are the parameters, $\mathbf{b}_f$ is the bias term, and $\sigma()$ is the sigmoid activation function, which maps values to the range between 0 and 1.

The next step is to determine what information from the new input $\mathbf{x}^{(t)}$ should be stored in the new cell state. Similar to the *forget gate*, an *input gate* is designed to make the decision. The *input gate* is formulated as:

$$\mathbf{i}_t = \sigma(\mathbf{W}_i \cdot \mathbf{x}^{(t)} + \mathbf{U}_i \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_i).$$

The input information $\mathbf{x}^{(t)}$ is processed by a few layers of neural networks to generate candidate values $\tilde{\mathbf{C}}^{(t)}$, which are used to update the cell state. The process of generating $\tilde{\mathbf{C}}^{(t)}$ is as:

$$\tilde{\mathbf{C}}^{(t)} = \tanh(\mathbf{W}_c \cdot \mathbf{x}^{(t)} + \mathbf{U}_c \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_c).$$

Then, we generate the new cell state $C^{(t)}$ by combining the old cell state $\mathbf{C}^{(t-1)}$ and the new candidate cell $\tilde{\mathbf{C}}^{(t)}$ as:

$$\mathbf{C}^{(t)} = \mathbf{f}_t \odot \mathbf{C}^{(t-1)} + \mathbf{i}_t \odot \tilde{\mathbf{C}}^{(t)},$$

where the notation $\odot$ denotes the Hadamard product, i.e., element-wise multiplication.

Finally we need to generate the hidden state $\mathbf{h}^{(t)}$, which can flow to the next position and serve as the output for this position at the same time if necessary. The hidden state is based on the updated cell state $\mathbf{C}^{(t)}$ with an *output gate* determining which parts of the cell state to be preserved. The output gate is formulated in the same way as the forget gate and the input gate as:

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot \mathbf{x}^{(t)} + \mathbf{U}_o \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_o).$$

The new hidden state $\mathbf{h}^{(t)}$ is then generated as follows:

$$\mathbf{h}^{(t)} = \mathbf{o}_t \odot \tanh(\mathbf{C}^{(t)}).$$

The entire process of the LSTM is shown in the Figure 3.13 and can be summarized as:

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_f \cdot \mathbf{x}^{(t)} + \mathbf{U}_f \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_f)\\
\mathbf{i}_t &= \sigma(\mathbf{W}_i \cdot \mathbf{x}^{(t)} + \mathbf{U}_i \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_i)\\
\mathbf{o}_t &= \sigma(\mathbf{W}_o \cdot \mathbf{x}^{(t)} + \mathbf{U}_o \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_o)\\
\tilde{\mathbf{C}}^{(t)} &= \tanh(\mathbf{W}_c \cdot \mathbf{x}^{(t)} + \mathbf{U}_c \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_c)\\
\mathbf{C}^{(t)} &= \mathbf{f}_t \odot \mathbf{C}^{(t-1)} + \mathbf{i}_t \odot \tilde{\mathbf{C}}^{(t)}\\
\mathbf{h}^{(t)} &= \mathbf{o}_t \odot \tanh(\mathbf{C}^{(t)}).
\end{aligned} \tag{3.4}$$

For convenience, we summarize the block of neural networks in LSTM for processing the t-th position described in Eq. (3.4) as:

$$\mathbf{C}^{(t)}, \mathbf{h}^{(t)} = \text{LSTM}(\mathbf{x}^{(t)}, \mathbf{C}^{(t-1)}, \mathbf{h}^{(t-1)}). \tag{3.5}$$

3.4.3 Gated Recurrent Unit

The gated recurrent unit (GRU) as shown in Figure 3.14 can be viewed as a variant of the LSTM where the forget gate and the input gate are combined as the *update gate* and the cell state and the hidden state are merged as the same one. These changes lead to a simpler gated RNN model which is formulated as:

$$\begin{aligned}
\mathbf{z}_t &= \sigma(\mathbf{W}_z \cdot \mathbf{x}^{(t)} + \mathbf{U}_z \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_z)\\
\mathbf{r}_t &= \sigma(\mathbf{W}_r \cdot \mathbf{x}^{(t)} + \mathbf{U}_r \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_r)\\
\tilde{\mathbf{h}}^{(t)} &= \tanh(\mathbf{W} \cdot \mathbf{x}^{(t)} + \mathbf{U} \cdot (\mathbf{r}_t \odot \mathbf{h}^{(t-1)}) + \mathbf{b})\\
\mathbf{h}^{(t)} &= (\mathbf{1} - \mathbf{z}_t) \odot \mathbf{h}^{(t-1)} + \mathbf{z}_t \odot \tilde{\mathbf{h}}^{(t)},
\end{aligned} \tag{3.6}$$

where $\mathbf{z}_t$ is the update gate and $\mathbf{r}_t$ is the reset gate. For convenience, we summarize the process in Eq. (3.6) as:

$$\mathbf{h}^{(t)} = \text{GRU}(\mathbf{x}^{(t)}, \mathbf{h}^{(t-1)}). \tag{3.7}$$

3.5 Autoencoders

An autoencoder can be viewed as a neural network that tries to reproduce the input as its output. Specifically, it has an intermediate hidden representation **h**, which describes a *code* to denote the input. An autoencoder consists of two

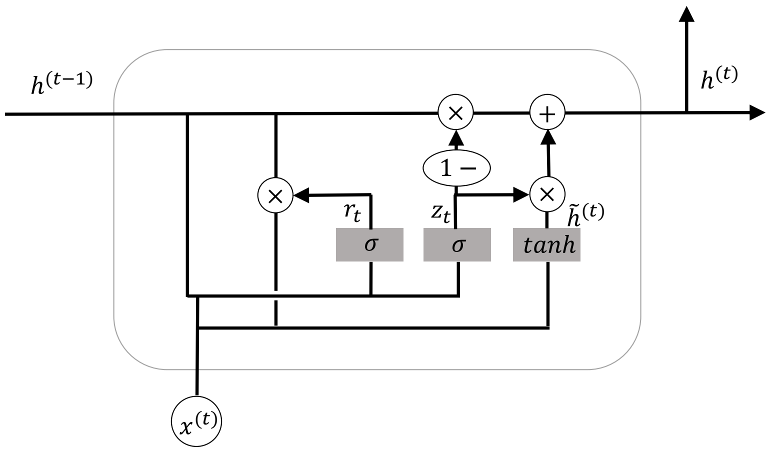


Figure 3.14 A block of GRU

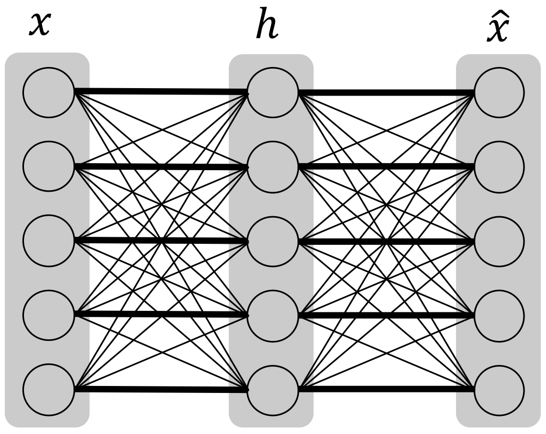


Figure 3.15 An autoencoder memorizes the input to the output. The bold connection indicates the memorization from the input to the output and the other connections are not used (with weights 0) in the autoencoder.

components: 1) an encoder $\mathbf{h} = f(\mathbf{x})$, which encodes the input $\mathbf{x}$ into a code $\mathbf{h}$, and 2) a decoder which aims to reconstruct $\mathbf{x}$ from the code $\mathbf{h}$. The decoder can be represented as $\hat{\mathbf{x}} = g(\mathbf{h})$. If an autoencoder works perfectly in reproducing the input, it is not especially useful. Instead, autoencoders are to approximately reproduce the input by including some restrictions. More specifically, they compress necessary information of the input in the hidden code $\mathbf{h}$ to re-

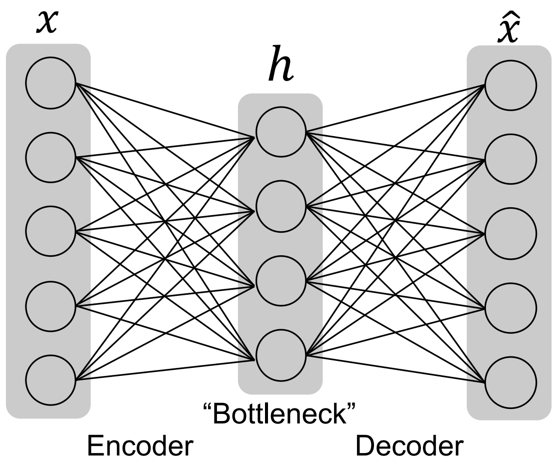


Figure 3.16 A general framework of autoencoder

produce satisfactory output. A general framework of autoencoders is shown in Figure 3.16. The input $\mathbf{x}$ is pushed through a "bottleneck", which controls the information can be preserved in the code $\mathbf{h}$. Then, a decoder network utilizes the code $\mathbf{h}$ to output $\hat{\mathbf{x}}$ which reconstructs the input $\mathbf{x}$. The network of an autoencoder can be trained by minimizing the reconstruction error:

$$\ell(\mathbf{x}, \hat{\mathbf{x}}) = \ell(\mathbf{x}, g(f(\mathbf{x}))), \tag{3.8}$$

where $\ell(\mathbf{x}, \hat{\mathbf{x}})$ measures the difference between $\mathbf{x}$ and $\hat{\mathbf{x}}$. For example, we can use the mean squared error as l. The design of the "bottleneck" is important for autoencoders. Ideally, as shown in Figure 3.15, without a "bottleneck", an autoencoder can simply learn to memorize the input and pass it through the decoder to reproduce it, which can render the autoencoder useless. There are different ways to design the "botteleneck" (i.e. adding constraints to the autoencoder). A natural way is to constrain the number of dimensions of the code $\mathbf{h}$, which leads to the *undercomplete autoencoder*. We can also add a regularizer term to discourage memorization between input and output, which leads to *regularized autoencoder*.

3.5.1 Undercomplete Autoencoders

Constraining the number of dimensions in the *code* $\mathbf{h}$ to be smaller than the input $\mathbf{x}$ is a simple and natural way to design the "bottleneck". An autoencoder with code dimension smaller than the input dimension is called an "undercomplete" autoencoder. An illustrative example of an "undercomplete" autoencoder is shown in Figure 3.16, where both the encoder and decoder only contain a single layer of networks and the hidden layer has fewer units than the input layer. By minimizing the reconstruction error, the model can preserve the most important features of the input in the hidden code.

3.5.2 Regularized Autoencoders

We can also make the autoencoder deeper by stacking more layers for both the encoder and decoder. For deep autoencoders, we must be careful about their capacity. Autoencoders may fail to learn anything useful if the encoder and decoder are given too much capacity. To prevent the autoencoder learning an identity function, we can include a regularization term in the loss function of the autoencoder as:

$$\ell(\mathbf{x}, g(f(\mathbf{x}))) + \eta \cdot \Omega(\mathbf{h}), \tag{3.9}$$

where $\Omega(\mathbf{h})$ is the regularization term applied to code $\mathbf{h}$ and η is a hyper-parameter controlling the impact of the regularization term.

In (Olshausen and Field, 1997), L_1 norm of the code $\mathbf{h}$ is adopted as the regulaization term as follows:

$$\Omega(\mathbf{h}) = \|\mathbf{h}\|_1. \tag{3.10}$$

The L_1 norm based regularization term enforces the code $\mathbf{h}$ to be sparse. In this case, the autoencoder is also named as a *sparse autoencoder*.

Another way to enforce the sparsity in the code is to constraint the neurons in the code $\mathbf{h}$ to be inactive most of the time. Here by "inactive", we mean that the value of a neuron in $\mathbf{h}$ is in a low level. We use $\mathbf{h}$ to denote the hidden code so far, which doesn't explicitly show what input leads to this code. Hence, to explicitly express the relation, for a given input $\mathbf{x}$, we use $\mathbf{h}(\mathbf{x})$ to denote its code learned by the autoencoder. Then, the average hidden code over a set of samples $\{\mathbf{x}_{(i)}\}_{i=1}^m$ is as:

$$\bar{\mathbf{h}} = \sum_{i=1}^{m} \mathbf{h}(\mathbf{x}_{(i)}). \tag{3.11}$$

Then, we would like to enforce each element in the hidden code to be close to a small value ρ. For example, ρ could be set to 0.05. In (Ng *et al.*, n.d.), each element in the hidden code is treated as a Bernoulli random variable with its corresponding value in $\bar{\mathbf{h}}$ as mean. These random variables are constraint to be close to the Bernoulli random variable with ρ as mean by KL-divergence as follows:

$$\Omega(\mathbf{h}) = \sum_j \left(\rho \log \frac{\rho}{\bar{\mathbf{h}}[j]} + (1-\rho) \log \frac{1-\rho}{1-\bar{\mathbf{h}}[j]}\right). \tag{3.12}$$

The autoencoder with the regularization term in Eq. (3.12) can also be called *sparse autoencoder*. While the regularization term can be applied to *under-complete autoencoder*, it can also work alone to serve as the "bottleneck". With the regularization terms, the hidden code **h** is not necessary to have a smaller dimension than the input.

3.6 Training Deep Neural Networks

In this section, we discuss the training procedure of deep neural networks. We briefly introduce gradient descent and its variants, which are popular approaches to train neural networks. We then detail the backpropagation algorithm, which is an efficient dynamic algorithm to calculate the gradients of the parameters of the neural networks.

3.6.1 Training with Gradient Descent

To train the deep learning models, we need to minimize a loss function $\mathcal{L}$ with respect to the parameters we want to learn. Generally, we denote the loss function as $\mathcal{L}(\mathbf{W})$ where $\mathbf{W}$ denotes all parameters needed to be optimized. Gradient descent and its variants are commonly adopted to minimize the loss function in deep learning. Gradient descent (Cauchy, n.d.) is a first-order iterative optimization algorithm. At each iteration, we update the parameters $\mathbf{W}$ by taking a step towards the direction of the negative gradient as follows:

$$\mathbf{W}' = \mathbf{W} - \eta \cdot \nabla_{\mathbf{W}} \mathcal{L}(\mathbf{W}), \tag{3.13}$$

where $\nabla_{\mathbf{W}} \mathcal{L}(\mathbf{W})$ denotes the gradient, and η is the learning rate, which is a positive scalar determining how much we want to go towards this direction. The learning rate η is commonly fixed to a small constant in deep learning.

The loss function is usually a summation of penalty over a set of training samples. Therefore, we write the loss function as follows:

$$\mathcal{L}(\mathbf{W}) = \sum_{i=1}^{N_s} \mathcal{L}_i(\mathbf{W}), \tag{3.14}$$

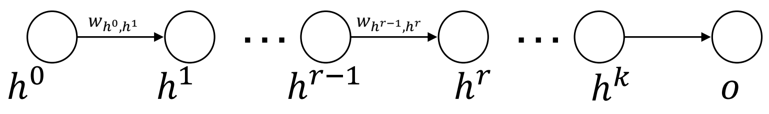


Figure 3.17 A sequence of neurons from consecutive layers

where $\mathcal{L}_i(\mathbf{W})$ is the loss for the i-th sample and N_s denotes the number of samples. In many cases, directly calculating $\nabla_{\mathbf{W}}\mathcal{L}(\mathbf{W})$ over all samples could be both space and time expensive. This where mini-batch gradient descent comes to rescue and is very popular in training deep neural networks. Instead of evaluating the gradient over all training samples, the mini-batch gradient descent method draws a small batch of samples out of the training data and uses them to estimate the gradient. This estimated gradient is then utilized to update the parameters. Specifically, the gradient can be estimated as $\sum\limits_{j\in\mathcal{M}} \nabla_{\mathbf{W}}\mathcal{L}_j(\mathbf{W})$, where $\mathcal{M}$ denotes the set of samples in the minibatch. Other variants of gradient descent have also been developed to train deep neural networks such as Adagrad (Duchi et al., 2011), Adadelta (Zeiler, 2012), and Adam (Kingma and Ba, 2014). They typically have a better convergence than the standard gradient descent methods.

3.6.2 Backpropagation

One crucial step to perform gradient-based optimization is to calculate the gradients with respect to all the parameters. The Backpropagation algorithm provides an efficient way to calculate the gradients using dynamic programming. It consists of two phases: 1) *Forward Phase*: In this phase, the inputs are fed into the deep model and pass through the layers, and the outputs are calculated using the current set of parameters, which are then used to evaluate the value of the loss function; and 2) *Backward Phase*: The goal of this phase is to calculate the gradients of the loss function with respect to the parameters. According to the chain rule, the gradients for all the parameters can be calculated dynamically in a backward direction, starting from the output layer. Next, we detail the backward pass.

Figure 3.17 illustrates a sequence of connected neural units $h^0, h^1, \ldots, h^k, o$ from different layers where h^i denotes a unit from the i-th layer with h^0 from the input layer and o from the output layer. Assuming that this is the only path going through the edge (h^{r-1}, h^r), we can calculate the derivative using the

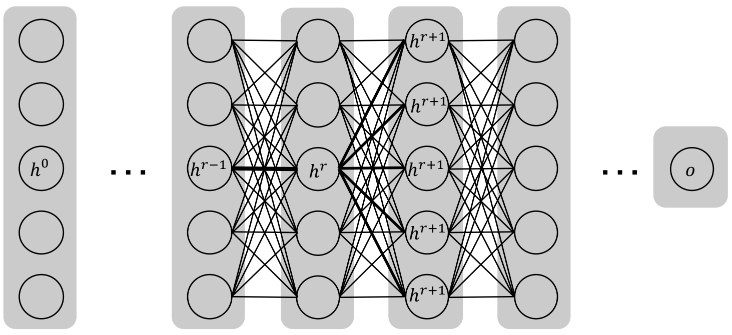


Figure 3.18 Decomposition of paths

chain rule as follows:

$$\frac{\partial \mathcal{L}}{\partial w_{(h^{r-1},h^r)}} = \frac{\partial \mathcal{L}}{\partial o} \cdot \left[\frac{\partial o}{\partial h^k} \prod_{i=r}^{k-1} \frac{\partial h^{i+1}}{\partial h^i} \right] \cdot \frac{\partial h^r}{\partial w_{(h^{r-1},h_r)}} \forall r \in 1 \dots k, \tag{3.15}$$

where $w_{(h^{r-1},h^r)}$ denotes the parameter between the neural units h^{r-1} and h^r.

In multi-layer neural networks, we often have several paths going through the edge (h^{r-1}, h^r). Hence, we need to sum up the gradients calculated through different paths as follows:

$$\frac{\partial \mathcal{L}}{\partial w_{(h^{r-1},h^r)}} = \underbrace{\frac{\partial \mathcal{L}}{\partial o} \cdot \left[\sum_{[h^r,h^{r+1},\dots,h^k,o]\in\mathcal{P}} \frac{\partial o}{\partial h^k} \prod_{i=r}^{k-1} \frac{\partial h^{i+1}}{\partial h^i} \right]}_{\text{Backpropagation computes } \Delta(h^r,o)=\frac{\partial \mathcal{L}}{\partial h^r}} \frac{\partial h^r}{\partial w_{(h^{r-1},h^r)}}, \tag{3.16}$$

where $\mathcal{P}$ denotes the set of paths starting from h^r to o, which can be extended to pass the edge (h^{r-1}, h^r). There are two parts on the right hand side of Eq. (3.16), where the second part is trouble-free (will be discussed later) to calculate while the first part (annotated as $\Delta(h^r, o)$) can be calculated recursively. Next, we discuss how to recursively evaluate the first term. Specifically, we have

$$\begin{aligned} \Delta(h^r, o) &= \frac{\partial \mathcal{L}}{\partial o} \cdot \left[\sum_{[h^r,h^{r+1},\dots,h^k,o]\in\mathcal{P}} \frac{\partial o}{\partial h^k} \prod_{i=r}^{k-1} \frac{\partial h^{i+1}}{\partial h^i} \right] \\ &= \frac{\partial \mathcal{L}}{\partial o} \cdot \left[\sum_{[h^r,h^{r+1},\dots,h^k,o]\in\mathcal{P}} \frac{\partial o}{\partial h^k} \prod_{i=r+1}^{k-1} \frac{\partial h^{i+1}}{\partial h^i} \cdot \frac{\partial h^{r+1}}{\partial h^r} \right]. \end{aligned} \tag{3.17}$$

As shown in Figure 3.18, we can decompose any path $P \in \mathcal{P}$ into two parts –

the edge (h^r, h^{r+1}) and the remaining path from h^{r+1} to o. Then, we can categorize the paths in $\mathcal{P}$ using the edge (h^r, h^{r+1}). Specifically, we denote the set of paths in $\mathcal{P}$ that share the same edge (h^r, h^{r+1}) as $\mathcal{P}_{r+1}$. As all paths in $\mathcal{P}_{r+1}$ share the same first edge (h^r, h^{r+1}), any path in $\mathcal{P}_{r+1}$ can be characterized by the remaining path (i.e., the path from h^{r+1} to o) besides the first edge. We denote the set of the remaining paths as $\mathcal{P}'_{r+1}$. Then, we can continue to simplify Eq.(3.17) as follows:

$$\begin{aligned}
\Delta(h^r, o) &= \frac{\partial \mathcal{L}}{\partial o} \cdot \left[\sum_{(h^r, h^{r+1}) \in \mathcal{E}} \frac{\partial h^{r+1}}{\partial h^r} \cdot \left[\sum_{[h^{r+1}, \ldots, h_k, o] \in \mathcal{P}'_{r+1}} \frac{\partial o}{\partial h_k} \prod_{i=r+1}^{k-1} \frac{\partial h^{i+1}}{\partial h^i} \right] \right] \\
&= \sum_{(h^r, h^{r+1}) \in \mathcal{E}} \frac{\partial h^{r+1}}{\partial h^r} \cdot \frac{\partial \mathcal{L}}{\partial o} \cdot \left[\sum_{[h^{r+1}, \ldots, h_k, o] \in \mathcal{P}'_{r+1}} \frac{\partial o}{\partial h_k} \prod_{i=r+1}^{k-1} \frac{\partial h^{i+1}}{\partial h^i} \right] \\
&= \sum_{(h^r, h^{r+1}) \in \mathcal{E}} \frac{\partial h^{r+1}}{\partial h^r} \cdot \Delta(h^{r+1}, o), \qquad (3.18)
\end{aligned}$$

where $\mathcal{E}$ denotes the set containing all existing edges pointing from the unit h^r to a unit h^{r+1} from the $(r+1)$-th layer. Note that, as shown in Figure 3.18, any unit in the $(r+1)$-th layer is connected to h^r, hence all units from the $(r+1)$-th layer are involved in the first summation in Eq. (3.18). Since each h^{r+1} is from the later layer than h^r, $\Delta(h^{r+1}, o)$ has been evaluated during the previous backpropagation process and can be directly used. We still need to compute $\frac{\partial h^{r+1}}{\partial h^r}$ to complete evaluating Eq. (3.18). To evaluate $\frac{\partial h^{r+1}}{\partial h^r}$, we need to take the activation function into consideration. Let a^{r+1} denote the values of unit h^{r+1} right before the activation function $\alpha()$, that is $h^{r+1} = \alpha(a^{r+1})$. Then, we can use the chain rule to evaluate $\frac{\partial h^{r+1}}{\partial h^r}$ as follows:

$$\frac{\partial h^{r+1}}{\partial h^r} = \frac{\partial \alpha(a^{r+1})}{\partial h^r} = \frac{\partial \alpha(a^{r+1})}{\partial a^{r+1}} \cdot \frac{\partial a^{r+1}}{\partial h^r} = \alpha'(a^{r+1}) \cdot w_{(h^r, h^{r+1})}, \qquad (3.19)$$

where $w_{(h^r, h^{r+1})}$ is the parameter between the two units h^r and h^{r+1}. Then, we can rewrite $\Delta(h^r, o)$ as follows:

$$\Delta(h^r, o) = \sum_{(h^r, h^{r+1}) \in \mathcal{E}} \alpha'(a^{r+1}) \cdot w_{(h^r, h^{r+1})} \cdot \Delta(h^{r+1}, o). \qquad (3.20)$$

Now, we return to evaluate the second part of Eq. (3.17) as follows:

$$\frac{\partial h^r}{\partial w_{(h^{r-1}, h^r)}} = \alpha'(a^r) \cdot h^{r-1}. \qquad (3.21)$$

With Eq. (3.20) and Eq. (3.21), we can now efficiently evaluate Eq. (3.16) recursively.

3.6.3 Preventing Overfitting

Deep neural networks can easily overfit to the training data due to its extremely high model capacity. In this section, we introduce some practical techniques to prevent neural networks from overfitting.

Weight Regularization

A common technique to prevent models from overfitting in machine learning is to include a regularization term on model parameters into the loss function. The regularization term constrains the model parameters to be relatively small that generally enables the model to generalize better. Two commonly adopted regularizers are the L_1 and L_2 norm of the model parameters.

Dropout

Dropout is an effective technique to prevent overfitting (Srivastava et al., 2014). The idea of dropout is to randomly ignore some units in the networks during each batch of the training procedure. There is a hyper-parameter called *dropout rate p* controlling the probability of neglecting each unit. Then, in each iteration, we randomly determine which neurons in the network to drop according to the probability p. Instead of using the entire network, the remaining neurons and network structure are then used to perform the calculation and prediction for this iteration. Note that the dropout technique is usually only utilized in the training procedure; in other words, the full network is always used to perform predictions during the inference procedure.

Batch Normalization

Batch normalization (Ioffe and Szegedy, 2015) was initially introduced to solve the problem of the internal covariate shift. It can also help mitigate overfitting. Batch normalization is to normalize the activation from the previous layer before feeding them into the next layer. Specifically, during the training procedure, if a mini-batch training procedure is adopted, this normalization is conducted by subtracting the batch mean and dividing the batch standard deviation. During the inference stage, we use the population statistics to perform the normalization.

3.7 Conclusion

In this chapter, we introduced a variety of basic deep architectures, including feedforward networks, convolutional neural networks, recurrent neural networks, and autoencoders. We then discussed gradient-based methods and the backpropagation algorithm for training deep modes. Finally, we reviewed some practical techniques to prevent overfitting during the training procedure of these architectures.

3.8 Further Reading

To better understand deep learning and neural networks, proper knowledge on linear algebra, probability and optimization is necessary. There are quite a few high quality books on these topics such as *Linear algebra* (Hoffman and Kunze, n.d.), *An Introduction to Probability Theory and Its Applications* (Feller, 1957), *Convex Optimization* (Boyd et al., 2004) and *Linear Algebra and Optimization for Machine Learning* (Aggarwal, 2018). These topics are also usually briefly introduced in machine learning books such as *Pattern Recognition and Machine Learning* (Bishop, 2006). There are dedicated books providing more detailed knowledge and content on deep neural networks such as *Deep Learning* (Goodfellow et al., 2016) and *Neural Networks and Deep Learning: A Textbook* (Aggarwal, 2018). In addition, various deep neural network models can be easily constructed with libraries and platforms such as Tensorflow (Abadi et al., 2015) and Pytorch (Paszke et al., 2017).