Chapter 1
Introduction to Deep Learning

Jingqing Zhang, Hang Yuan, and Hao Dong

Abstract This chapter aims to briefly introduce the fundamentals for deep learning, which is the key component of deep reinforcement learning. We will start with a naive single-layer network and gradually progress to much more complex but powerful architectures such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs). We will end this chapter with a couple of examples that demonstrate how to implement deep learning models in practice.

Keywords Deep learning · Convolutional neural networks · Recurrent neural networks

1.1 Introduction

This chapter introduces the basics of deep learning that will be used in deep reinforcement learning. For those who are already familiar with the fundamentals, please feel free to skip this chapter. This book's content is meant to be self-contained, but one might wish to refer to other books like Bishop (2006) and Goodfellow et al. (2016) to understand some of the topics in depth. Unlike classical reinforcement learning which uses analytical methods for function approximation, deep reinforcement learning relies on deep neural networks such that it can leverage the power of large data volume and increased computing resources. In general, there are two types of models.

J. Zhang
Imperial College London, London, UK
e-mail: jingqing.zhang15@imperial.ac.uk

H. Yuan
Oxford University, Oxford, UK
e-mail: hang.yuan@keble.ox.ac.uk

H. Dong (✉)
Peking University, Beijing, China
e-mail: hao.dong@pku.edu.cn

H. Dong et al. (eds.), *Deep Reinforcement Learning*,
https://doi.org/10.1007/978-981-15-4095-0_1

Discriminative Models study the conditional probability $p(\boldsymbol{y}|\boldsymbol{x})$ with input data $\boldsymbol{x}$ and a target label $\boldsymbol{y}$. In other words, discriminative models predict the label $\boldsymbol{y}$ given the input data $\boldsymbol{x}$. Discriminative models are mostly adopted in tasks such as classification and regression which require discriminative judgement. More specifically, in terms of **classification**, a model is designed to categorize the input data into specific classes from a set of given classes. The binary classification, as the most fundamental classification task, predicts one class from two candidates. For example, in the sentiment analysis (Maas et al. 2011), a piece of text is classified as either positive or negative. In contrast, in multi-label classification, the input data can be assigned with several classes at the same time. In some cases, instead of identifying the class directly, a classification model needs to calculate the probability distribution of classes. For example, the input data has a probability of 80% to be assigned with class A and a probability of 20% to be assigned with class B. This probabilistic representation is mostly needed during training for optimization purposes. Deep learning has achieved great success on classification tasks such as image classification (Krizhevsky et al. 2009) and text classification (Yang et al. 2019). Unlike classifications, which produce discrete class labels, a **regression** studies continuous values. An example of regression is to predict future traffic speed based on historical traffic data (Liao et al. 2018a,b). Regression models remain discriminative models as long as they are learning the conditional probability.

Generative Models are designed to study the joint probability $p(\boldsymbol{x}, \boldsymbol{y})$. Generative models are usually used to generate observed data by learning the distribution of the observed data. For example, the generative adversarial networks (GANs) (Goodfellow et al. 2014) are adopted to generate, reconstruct, and denoise images (Ledig et al. 2017; Yang et al. 2018). Nonetheless, techniques in deep learning like GANs have no explicit relationship with the distribution of the observed data but focus more on the similarity between generated samples and observed data. Meanwhile, generative models are also used for classification purposes like Naive Bayes (Ng and Jordan 2002; Rish et al. 2001). Although both generative models and discriminative models are used for classification, discriminative models only consider which label should be assigned given the observed data, while generative models try to learn the distribution of the observed data. For example, Naive Bayes studies the likelihood $p(\boldsymbol{x}|\boldsymbol{y})$, i.e. the probability of the observed data to be generated assuming a label.

Most deep neural networks that have been explored are discriminative models no matter whether they are initially designed for discriminative or generative problems. This is because many generative problems in practice can be simplified to classification or regression problems. For example, question answering (Devlin et al. 2019) selects which part of the provided context is the answer to the given question; abstractive summarization (Zhang et al. 2019b) selects words from vocabulary to assemble summaries based on the probability of each word. For both cases, they are trying to generate something but one uses a classification approach and the other uses a regression approach.

Concretely, this chapter covers the mechanical components and techniques such as the definitions of neurons, activation functions, and optimizers that can build up deep neural networks and deep learning applications. Fundamental deep neural networks such as multilayer perceptron (MLP), convolutional neural networks (CNNs), and recurrent neural networks (RNNs) are also within the scope of this chapter. Finally, Sect. 1.10 introduces examples of implementing deep neural networks by TensorFlow and TensorLayer. Please refer to Goodfellow et al. (2016) for a more detailed introduction to deep learning.

1.2 Perceptron

1.2.1 One Output

A neuron (node) is the basic unit of deep neural networks. Originally, the neuron was proposed to be an abstract representation of the real neuron in the brain, which receives electrical impulses from its dendrites. When this specific neuron is polarized enough, it will send an action potential spike via its axon to the other adjacent neurons. In a real biological system, these steps do not take place at once but at a more granular scale. Spiking neural networks are better suited in describing the underlying biological processes. At the moment, the deep learning community relies more on deep neural networks (DNNs), also known as artificial neural networks (ANNs). The neurons in deep neural networks are formalized with numerical inputs and outputs. A neuron can have many output neurons in the next layer and a neuron can also have many input neurons in the previous layer. This is a many-to-many relationship. A neuron in one layer aggregates the signals being passed through from its input neurons in the previous layer. This aggregated signal will then be passed through an activation function that will determine the neuronal behavior. Concretely, if the aggregated signal is strong enough, then the activation function will "activate" this neuron and pass forward a high value to the output neurons in the next layer. Otherwise, a low value will be passed forward instead (Fig. 1.1).

$$z = w_1x_1 + w_2x_2 + w_3x_3. \tag{1.1}$$

A neural network can have an arbitrary number of neurons with random connections among themselves, but for the ease of computation, the neurons are organized layer after layer. Typically, a single neuron will have at least two layers, namely the input and output layer as shown in Fig. 1.2. This network can be formalized by Eq. (1.1) and can help with simple decision-making. An example is helping a group of students decide whether or not they can play soccer on a day based on the weather condition. The decision may also rely on some other factors such as the expense of the soccer field and the students' availability. If the weather

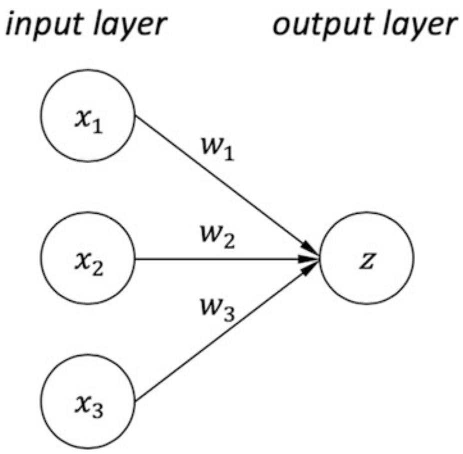


Fig. 1.1 A neural network with three input neurons and one output neuron

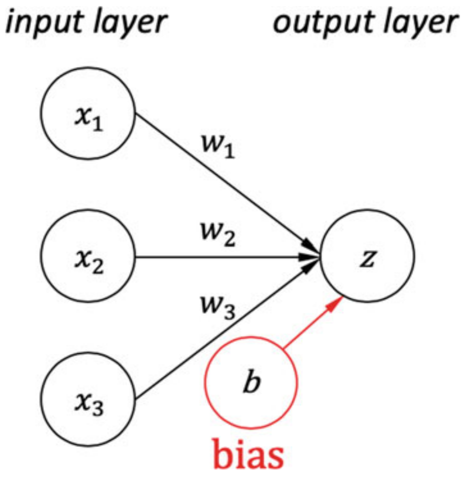


Fig. 1.2 A neural network with bias

condition has a higher impact on the decision, the corresponding weight (w) should have a greater absolute value. In contrast, factors of less importance should have weights with a lower absolute value. If a weight is set as zero, the corresponding input factor is discarded in the decision-making process. This kind of neural network is also called a single-layer neural network or **perceptron**.

1.2.2 Bias and Decision Boundary

A bias is an extra scalar that is added to the neuron to shift the value of the output. For example, Fig. 1.2 shows the single-layer neural network with a bias and it can be formalized as:

$$z = w_1 x_1 + w_2 x_2 + w_3 x_3 + b. \tag{1.2}$$

The bias can help a neural network to fit the data better. For example, let us define a binary classification problem, in which the label y is 1 if the input z is positive and 0 otherwise:

$$y = \begin{cases} 1 \text{ when } z > 0 \\ 0 \text{ otherwise} \end{cases} \tag{1.3}$$

Then the distribution of data samples is shown in Fig. 1.3 and we need to find out a set of weights and bias that can best fit the data. The decision boundary is defined to partition the data samples into the two classes for the binary classification. Formally, the decision boundary is $\{x_1, x_2, x_3 | w_1 x_1 + w_2 x_2 + w_3 x_3 + b = 0\}$.

Let us first simplify this problem by having only two inputs, i.e. $z = w_1 x_1 + w_2 x_2 + b$. As shown in the left-hand side of Fig. 1.3, without the bias component, i.e. $b = 0$, the decision boundary must cross the origin of the Cartesian coordinate as demonstrated by the blue line in the bottom-left corner. However, this apparently cannot fit the data distribution well enough as the data samples for both classes fall on the same side of the boundary. If the bias is non-zero, the decision boundary crosses both axes at $(0, -\frac{b}{w_2})$ and $(-\frac{b}{w_1}, 0)$, respectively, and this decision boundary can fit the data distribution better if the weights and bias are well chosen.

If we come back to the original setting of the problem where the neuron has three inputs, i.e. $z = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$, the decision boundary will become a plane as shown in the right-hand side of Fig. 1.3. In a linear model like the single-layer neural networks defined in Eq. (1.2), the decision boundary is also called **hyperplane**.

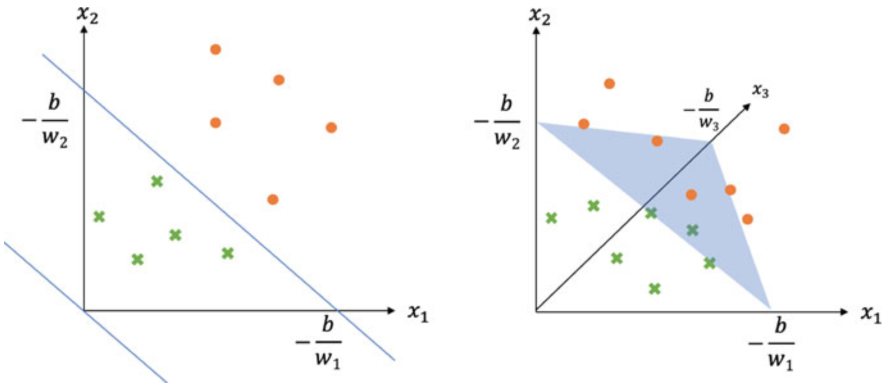


Fig. 1.3 Decision boundary of linear model with two and three inputs. Left: $z = w_1 x_1 + w_2 x_2 + b$, Right: $z = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$

1.2.3 More Than One Output

The single-layer neural network can have multiple neurons. Figure 1.4 shows an example of a single-layer neural network with two outputs, which are computed by Eq. (1.4). Since each output is connected with all of the inputs, the output layer is also called the **dense layer**, or **fully connected (FC) layer**:

$$\begin{aligned} z_1 &= w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1 \\ z_2 &= w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2. \end{aligned} \tag{1.4}$$

In practice, the dense layer can be represented by matrix multiplication:

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \tag{1.5}$$

where $\boldsymbol{W} \in \mathbb{R}^{m\times n}$ is a matrix to represent weights and $\boldsymbol{z} \in \mathbb{R}^m$, $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$ are column vectors to represent outputs, inputs, and biases, respectively. In the example by Eq. (1.4), $m = 2$ and $n = 3$.

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{1.6}$$

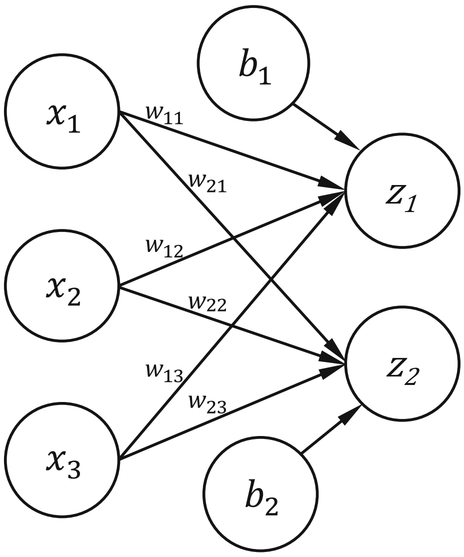


Fig. 1.4 The neural network with three input neurons and two output neurons

1.3 Multilayer Perceptron (MLP)

A multilayer perceptron (MLP) (Rosenblatt 1958; Ruck et al. 1990) stems from a single dense layer to have at least two dense layers. Figure 1.5 presents an MLP consisting of four dense layers. The three layers between the input and output layers are "hidden" because they are typically not accessible from outside the network, and we will refer them as the **hidden layers**. Compared with the network with a single dense layer, MLP can fit more complex data. In other words, MLP has a stronger learning capability than a single-layer neural network. However, more hidden layers in MLP do not necessarily lead to stronger learning capacity. The universal approximation theorem states that a feedforward network with one hidden layer (e.g., MLP with one hidden layer) and any squashing activation function (e.g., sigmoid or tanh) can approximate any Borel measurable function, given that the hidden layer has sufficient hidden units (Samuel 1959; Hornik et al. 1989; Goodfellow et al. 2016). However, in practice, such a network can be inflexible to train or hard to avoid overfitting if the hidden layer is extremely large. Therefore, deep neural networks including MLP typically have several hidden layers.

We start with the logic operations to demonstrate how a network approximates a function. The logic operations including AND, OR, NOR, NAND, XNOR, and XOR take two binary numbers and return either zero or one. For example, AND returns one if and only if the two binary numbers are both one. Simple logic operations can be easily approximated by the perceptron, which can be defined by Eq. (1.7).

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \quad \text{where} \quad z = w_1x_1 + w_2x_2 + b \tag{1.7}$$

Figure 1.6 shows that hyperplanes defined by perceptron can be easily found to separate the points between zero and one for AND, OR, NOR, and NAND. However, it is not possible to do the same for XOR or XNOR.

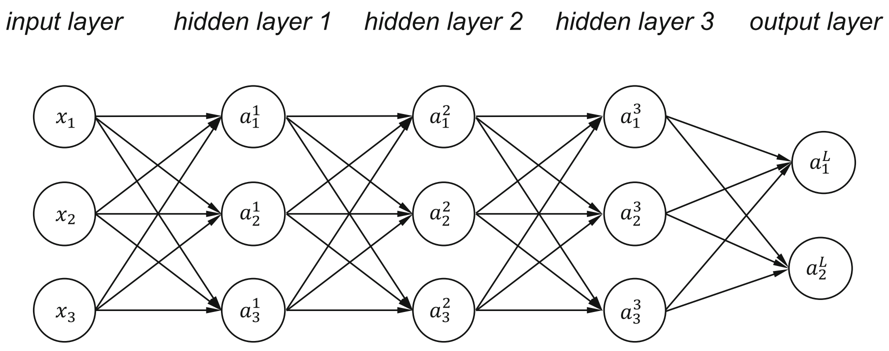
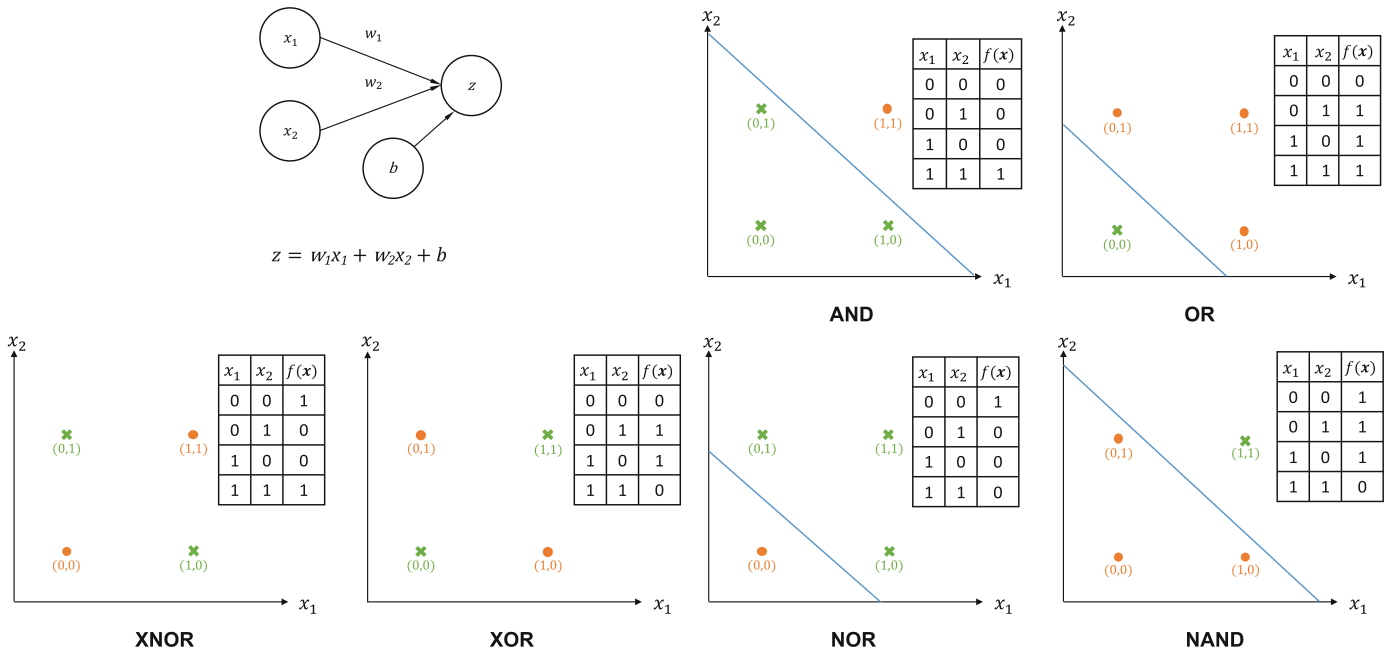


Fig. 1.5 An example of multilayer perceptron (MLP) with three hidden layers and one output layer. The neurons are represented by a^l_i, where l the layer index and i is the output index



AND

x_1	x_2	$f(x)$
0	0	0
0	1	0
1	0	0
1	1	1

OR

x_1	x_2	$f(x)$
0	0	0
0	1	1
1	0	1
1	1	1

XNOR

x_1	x_2	$f(x)$
0	0	1
0	1	0
1	0	0
1	1	1

XOR

x_1	x_2	$f(x)$
0	0	0
0	1	1
1	0	1
1	1	0

NOR

x_1	x_2	$f(x)$
0	0	1
0	1	0
1	0	0
1	1	0

NAND

x_1	x_2	$f(x)$
0	0	1
0	1	1
1	0	1
1	1	0

Fig. 1.6 Top left: The perceptron with two inputs and one output. The rest: Hyperplanes can be found to separate the points between zero (green) and one (orange) for AND, OR, NOR, NAND, but no hyperplane defined by perceptron can be found for XOR, XNOR

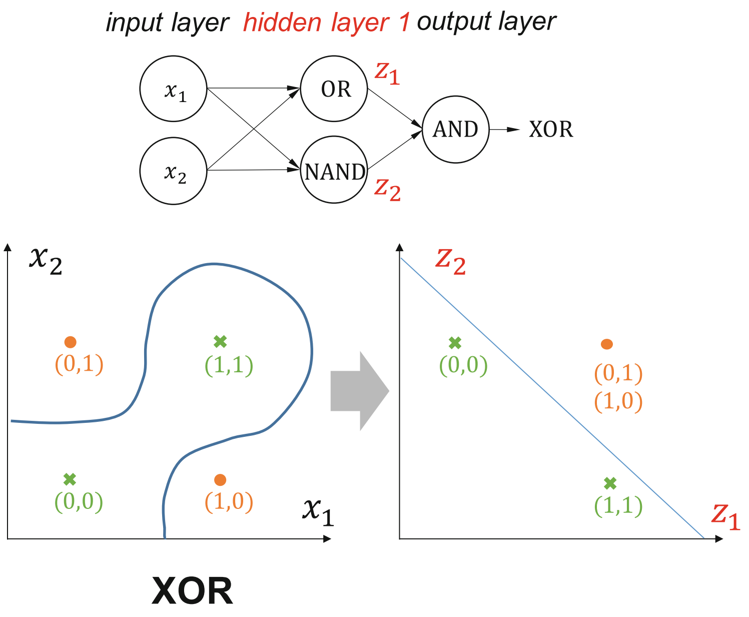


Fig. 1.7 Left: An MLP that approximates XOR. Mid and right: Transformation from the original data space to the feature space, where the data points are linearly separable

The XOR cannot be approximated by a linear model directly working on the original inputs x_1, x_2 like the perceptron, so we need to transform the inputs first. As an example, we use MLP with one hidden layer as shown in Fig. 1.7 to approximate XOR. This MLP first transforms the inputs x_1, x_2 into a new space by approximating the logic operations OR and NAND, and then, in the transformed space, the points are linearly separable by an approximation of AND. The transformed space is also named feature space and this example shows how learning features can improve the learning capacity of a model.

1.4 Activation Functions

Matrix addition and multiplication are both linear operators but the learning capability of a linear model is rather limited. For example, a linear model cannot easily approximate a cosine function. Most real-world problems that deep neural networks are applied to solve cannot be simplified as a linear transformation, so non-linearity is important for deep neural networks. In practice, the non-linearity of deep neural networks is introduced by activation functions, which are typically element-wise operations. In addition, the activation functions are necessary when a model needs to obtain probability vectors instead of vectors with arbitrary values. The choice of activation functions varies in different applications. Even though there

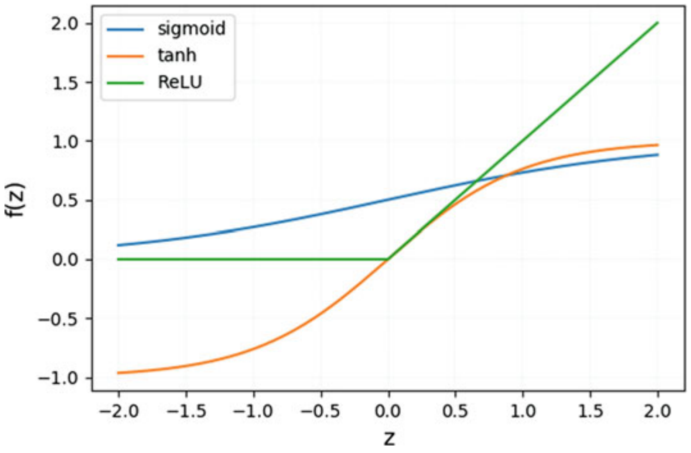


Fig. 1.8 Demonstration of three element-wise activation functions including sigmoid, tanh, and ReLU. The sigmoid constrains values between 0 and 1, while the tanh returns values between -1 and 1. The ReLU returns zero when the input is non-positive but is equivalent to $f(x) = x$ when the input is positive

exist some functions that work well in most deep learning applications, there might be other functions that have better performance on a case by case basis. Therefore, the design of activation functions remains an active research area. This section introduces four commonly used activation functions, namely sigmoid, tanh, ReLU, and softmax (Fig. 1.8).

The logistic **sigmoid** as an activation function has float output ranging between 0 and 1 as defined by Eq. (1.8). The sigmoid function can be used at the output layer for classification purpose. For example, a binary classifier with one output neuron uses sigmoid to constrain the output value between 0 and 1 and then converts it to a discrete class label (either 0 or 1) by using a threshold like 0.5.

$$f(z) = \frac{1}{1 + e^{-z}}. \tag{1.8}$$

Similar to the sigmoid function, the **hyperbolic tangent (tanh)** constrains output values to a limited range between -1 and 1 as defined by Eq. (1.9). The tanh function can be used in the hidden layers (Glorot et al. 2011) to provide non-linearity. It can also be used in the output layer, e.g. in the generation of images whose pixel values range between -1 and 1.

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{1.9}$$

The **rectified linear unit (ReLU)**, also known as the rectifier, is defined by Eq. (1.10). The study by Glorot et al. (2011) shows that ReLU is more promising than sigmoid and tanh, and ReLU has also been widely adopted in recent works (He et al. 2016; Cao et al. 2017; Noh et al. 2015). The empirical advantages of ReLU are:

- Easier to implement and compute: in the implementation of ReLU, a simple comparison with zero is conducted first and then the activation is set to zero or z accordingly. Whereas in the sigmoid and tanh, the exponential function is harder to compute especially in the case of large networks.
- Easier for a network to optimize: ReLU function is close to being linear, consisting of two linear functions. This property makes the gradient large and consistent. The gradient of an active neuron by ReLU is always one, but the gradient of a neuron by sigmoid or tanh suffers from vanishing when the activated value approaches the limits (i.e., -1, 0, or 1).

$$f(z) = \begin{cases} 0 & \text{when } z <= 0 \\ z & \text{when } z > 0 \end{cases} \tag{1.10}$$

However, merely setting negative values to zero in ReLU can lead to information loss. Imagine, if a neuron constantly outputs zero, it will always output zero in the future and is unlikely to recover. This can happen because of an inappropriate learning rate or a negative bias. The work by Xu et al. (2015) proposes a solution to this with another activation function called leaky ReLU, which is defined in Eq. (1.11). The scalar α in this equation is a small positive value to control the slope (e.g., 0.01 or 0.02) so that a little information from the negative scope can be retained.

$$f(z) = \begin{cases} \alpha z & \text{when } z <= 0 \\ z & \text{when } z > 0 \end{cases} \tag{1.11}$$

The parametric ReLU (PReLU) (He et al. 2015) is similar to the leaky ReLU except that it considers α as a trainable parameter. There is no clear evidence to show which one of ReLU, leaky ReLU or PReLU is significantly better than the others since the choice varies in different scenarios.

Unlike the activation functions mentioned above, the **softmax** function, defined by Eq. (1.12), provides normalization based on all values from previous layer's outputs. The softmax function first computes the exponential function e^z and then normalizes each entry by dividing it.

$$f(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \tag{1.12}$$

In practice, the softmax function is typically only used in the output layer to normalize the output vector z into a probability vector, where each entry is non-negative and the entries are added to one. Therefore, the softmax function is widely used for classification.

1.5 Loss Functions

In deep learning, loss functions are defined to quantify an error, also known as the loss value or cost, between the prediction and target (i.e., ground truth, gold standard). The loss value is normally used as the objective to optimize the parameters of neural networks, such as the weights and biases. This section introduces some commonly used loss functions and Sect. 1.6 will introduce how to optimize the parameters based on the loss values.

1.5.1 Cross-Entropy Loss

Before we introduce the cross-entropy loss, we start with a similar concept named Kullback–Leibler (KL) divergence. The KL divergence measures the similarity between two distribution $P(x)$ and $Q(x)$:

$$D_{\mathrm{KL}}(P\|Q) = \mathbb{E}_{x\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{x\sim P}[\log P(x) - \log Q(x)] \tag{1.13}$$

The KL divergence is non-negative and equals to 0 if and only if P and Q have the same distribution. Since the first term in KL divergence has no relation with Q, we introduce cross-entropy which can remove the first term.

$$H(P, Q) = -\mathbb{E}_{x\sim P} \log Q(x) \tag{1.14}$$

Therefore, minimizing the cross-entropy with respect to Q is equivalent to minimizing the KL divergence. As mentioned before, in some deep learning applications, e.g. classification, deep neural networks calculate a probability distribution of classes in practice instead of identifying the target class directly. Therefore, we can use the cross-entropy to measure how well the predicted distribution is and then update the network accordingly.

We start with binary classification as an example. In binary classification, for each input data sample x_i with target y_i (i.e., 0 or 1), a model needs to predict the probability of each candidate class $\hat{y}_{i,1}$, $\hat{y}_{i,2}$. Since $\hat{y}_{i,1} + \hat{y}_{i,2} = 1$, we can rewrite the prediction as $\hat{y}_i$ which represents the probability of one class, so the probability of the other class is $1 - \hat{y}_i$. Therefore, a neural network for binary classification typically has only one output neuron (with sigmoid) and following the definition of cross-entropy, we have:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\Big(y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)\Big), \tag{1.15}$$

where N represents the total number of data samples. Since y_i is either 0 or 1, only one of $y_i \log \hat{y}_i$ and $(1 - y_i)\log(1 - \hat{y}_i)$ is retained for each data sample. If $\forall i,\ y_i = \hat{y}_i$, the cross-entropy loss is zero.

In multinomial classification, where each data sample x_i is classified into one out of three or more candidate classes, a model predicts the probability of each class $\{\hat{y}_{i,1}, \hat{y}_{i,2}, \ldots, \hat{y}_{i,M}\}$, where $M \geq 3$ and $\sum_{j=1}^{M} \hat{y}_{i,j} = 1$. The target of each data sample is referred to as c_i, which is an integer between $[1, M]$, and it can be converted to a one-hot vector $\boldsymbol{y}_i = [y_{i,1}, y_{i,2}, \ldots, y_{i,M}]$, where only $y_{i,c_i} = 1$ and others are zero. Then, we can write the cross-entropy loss for the multinomial classification as below:

$$\begin{aligned}\mathcal{L} &= -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j} \log \hat{y}_j = -\frac{1}{N}\sum_{i=1}^{N}(0 + \cdots + y_{i,c_i} \log \hat{y}_{c_i} + \cdots + 0) \\ &= -\frac{1}{N}\sum_{i=1}^{N} \log \hat{y}_{c_i}. \end{aligned} \tag{1.16}$$

1.5.2 $\mathcal{L}_p$ Norm

Given a vector $\boldsymbol{x}$, p-norm measures its scale such that a vector with larger values has a larger scale, and it is defined as follows, where p is an integer greater or equal to 1.

$$\begin{aligned} \|\boldsymbol{x}\|_p &= \left(\sum_{i=1}^{N} |x_i|^p\right)^{1/p} \\ \text{i.e.,} \|\boldsymbol{x}\|_p^p &= \sum_{i=1}^{N} |x_i|^p \end{aligned} \tag{1.17}$$

In deep learning, a p-norm can be used to measure the difference between two vectors written as $\mathcal{L}_p$, as in Eq. (1.18), where $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ are the target and prediction, respectively.

$$\mathcal{L}_p = \|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_p^p = \sum_{i=1}^{N} |y_i - \hat{y}_i|^p. \tag{1.18}$$

1.5.3 Mean Squared Error

The mean squared error (MSE) is the averaged $\mathcal{L}_2$ norm as defined by Eq. (1.19). The MSE can be used for regression problems in which the outputs of a neural network are continuous values. For example, the difference between two images can be measured by MSE between pixels of the two images.

$$\mathcal{L} = \frac{1}{N}\|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_2^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2, \tag{1.19}$$

where N is the number of data samples, and $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ are the target and prediction, respectively.

1.5.4 Mean Absolute Error

Similar to MSE, the mean absolute error (MAE) can also be used for regression problems and is defined as the averaged $\mathcal{L}_1$ norm.

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{1.20}$$

Both MSE and MAE minimize the difference between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$. MSE offers a better mathematical property making it easier to compute the partial derivative which is required by gradient descent. In contrast, since the absolute term is not differentiable when $y_i = \hat{y}_i$, the partial derivative of MAE needs to walk around this case. In addition, when the difference between y_i and $\hat{y}_i$ is greater than 1, MSE has a larger error value compared to MAE (i.e., $(y_i - \hat{y}_i)^2$ vs $|y_i - \hat{y}_i|$) which can lead to a quicker optimization of a network.

1.6 Optimization

In this section we describe the optimization of deep neural networks, or in other words, how the parameters of deep neural networks are trained. This section covers back-propagation, gradient descent, stochastic gradient descent, and the selection of hyper-parameters.

1.6.1 Gradient Descent and Error Back-Propagation

Given a neural network and a loss function, the training of the neural network is formalized to learning its parameters $\boldsymbol{\theta}$ so that the loss $\mathcal{L}$ is minimized. Finding the minimum by searching $\boldsymbol{\theta}\ \ s.t.\ \ \nabla_{\boldsymbol{\theta}}\mathcal{L} = 0$ in a brute-force fashion is infeasible in practice, especially when the formula is as complex as that of a deep neural network. Therefore, we consider a process to approach the minimum by small steps and this technique is called **gradient descent**.

Figure 1.9 illustrates two examples of gradient descent. The learning process of gradient descent starts from a randomly picked point and the loss $\mathcal{L}$ decreases along with the update of parameters as denoted by the red dotted path. Similarly, in a neural network, its parameters are first randomly initialized and updated each step based on the partial derivative $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$. More specifically, the parameters are updated iteratively by $\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$, where α the learning rate of each step and $\boldsymbol{\theta}$ mostly consists of weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$ of each layer.

Back-Propagation (Rumelhart et al. 1986; LeCun et al. 2015) is a technique to compute the partial derivative $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ in the network. To make the computation of $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ clearer, we introduce an intermediate value $\boldsymbol{\delta} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}$, which is the partial derivative of the loss $\mathcal{L}$ with respect to the layer's output $\boldsymbol{z}$. Then, the partial derivatives of the loss $\mathcal{L}$ with respect to each parameter, which assemble $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$, are computed based on the intermediate value $\boldsymbol{\delta}$.

The layers are indexed as $l = 1, 2, \ldots L$, where L is the index of the output layer, each layer has an output $\boldsymbol{z}^l$, an intermediate value $\boldsymbol{\delta}^l = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l}$, and an activation output $\boldsymbol{a}^l = f(\boldsymbol{z}^l)$ (where f is the activation function). We use an MLP with MSE loss and a sigmoid activation function as an example. Given $\boldsymbol{z}^l = \boldsymbol{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$, $\boldsymbol{a}^l = f(\boldsymbol{z}^l) = \frac{1}{1+e^{-\boldsymbol{z}^l}}$, and $\mathcal{L} = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{a}^L\|_2^2$, we represent the partial derivative of

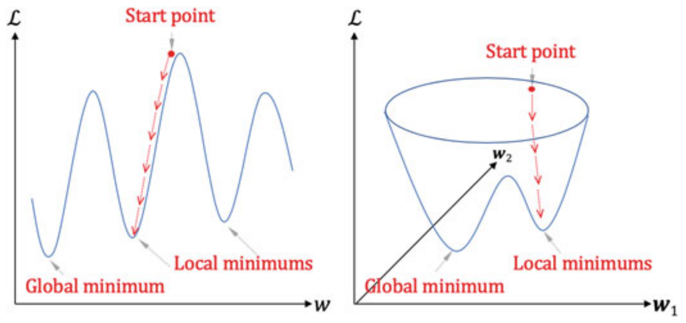


Fig. 1.9 Examples of gradient descent with a trainable parameter (Left) and two trainable parameters (Right). In gradient descent, the learning process starts from a ranomly picked point. With the parameters updates shown by the red arrows, the loss $\mathcal{L}$ gradually reachs a saddle point. Note that there is no guarantee the gradient descent can find the global minimum but in most cases a local minimum is approached

the activation output with respect to its original output as $\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} = f'(\boldsymbol{z}^l) = f(\boldsymbol{z}^l)(1 - f(\boldsymbol{z}^l)) = \boldsymbol{a}^l(1-\boldsymbol{a}^l)$ and the partial derivative of the loss with respect to the activation output as $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^L} = (\boldsymbol{a}^L - \boldsymbol{y})$. To compute the partial derivative of the loss with respect to the output layer, we apply the chain rule as follows:

- $\boldsymbol{\delta}^L = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^L}\frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} = \left(\boldsymbol{a}^L - \boldsymbol{y}\right) \odot \left(\boldsymbol{a}^L\left(1 - \boldsymbol{a}^L\right)\right)$

Then, the partial derivative of the loss with respect to all the other layers' outputs can be computed recursively as follows, where $l = 1, 2, \ldots, L-1$.

- Given $\boldsymbol{z}^{l+1} = \boldsymbol{W}^{l+1}\boldsymbol{a}^l + \boldsymbol{b}^{l+1}$
- Then $\boldsymbol{\delta}^l = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{l+1}}\frac{\partial \boldsymbol{z}^{l+1}}{\partial \boldsymbol{a}^l}\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} = \left(\boldsymbol{W}^{l+1}\right)^T \boldsymbol{\delta}^{l+1} \odot \left(\boldsymbol{a}^l\left(1 - \boldsymbol{a}^l\right)\right)$

The second step of the back-propagation is to compute the partial derivative of the loss with respect to the parameters $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l}$ and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l}$ of each layer based on the intermediate value $\boldsymbol{\delta}^l$.

- Given $\boldsymbol{z}^l = \boldsymbol{W}^l\boldsymbol{a}^{l-1} + \boldsymbol{b}^l$, we have $\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{W}^l} = \boldsymbol{a}^{l-1}$ and $\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{b}^l} = 1$
- Then $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l}\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{W}^l} = \boldsymbol{\delta}^l\left(\boldsymbol{a}^{l-1}\right)^T$, $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^l}\frac{\partial \boldsymbol{z}^l}{\partial \boldsymbol{b}^l} = \boldsymbol{\delta}^l$

Finally, we use the $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l}$ and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l}$ to update the parameters $\boldsymbol{W}^l$ and $\boldsymbol{b}^l$ as follows:

- $\boldsymbol{W}^l := \boldsymbol{W}^l - \alpha\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^l}$
- $\boldsymbol{b}^l := \boldsymbol{b}^l - \alpha\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^l}$

With the partial derivative $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$, gradient descent updates the parameter iteratively and converges to a minimum point of the loss function as in Fig. 1.9. In practice, the converged point is typically a local minimum rather than the global one. However, as deep neural networks offer a good representation capacity, the local minimums tend to be close to the global minimum (Goodfellow et al. 2016).

In gradient descent, the computation of the loss value $\mathcal{L}$ in each iteration can be expensive if the size of dataset (i.e., total number of data samples) N is large. Given the MSE in the example above, we can expand the MSE to:

$$\mathcal{L} = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{a}^L\|_2^2 = \frac{1}{2}\sum_{i=1}^{N}\left(y_i - a_i^L\right)^2 \tag{1.21}$$

In practice, the size of dataset can be more than tens of thousands so the gradient descent suffers from inefficiency due to the computation of $\mathcal{L}$. To tackle this problem, we introduce stochastic gradient descent which computes $\mathcal{L}$ of a small batch of data samples.

1.6.2 *Stochastic Gradient Descent and Adaptive Learning Rate*

Instead of computing the loss $\mathcal{L}$ of all training data in each iteration, the stochastic gradient descent (SGD) (Bottou and Bousquet 2007) randomly selects a small number of data samples from the training set. These small number of data samples are named as a **mini-batch**, and the quantity of data samples in the mini-batch is referred to as **batch size**. We can rewrite the Eq. (1.21) with batch size B and $B \ll N$ so that the computation of $\mathcal{L}$ is much more efficient.

$$\mathcal{L} = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{a}^L\|_2^2 = \frac{1}{2}\sum_{i=1}^{B}\left(y_i - a_i^L\right)^2 \tag{1.22}$$

The training process of stochastic gradient descent is outlined in Algorithm 1. If the parameters are updated with sufficient times (i.e., sufficient training steps/iterations), the mini-batches can cover the entire training set.

Algorithm 1 The training process of stochastic gradient descent (SGD)

```
Input: Parameters $\boldsymbol{\theta}$, learning rate $\alpha$, number of training steps/iterations $S$
1: for $i = 0$ to $S$ do
2:     Compute $\mathcal{L}$ of a mini-batch;
3:     Compute $\frac{\partial \mathcal{L}}{\partial \theta}$ by back-propagation;
4:     $\nabla\boldsymbol{\theta} \leftarrow -\alpha * \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$;
5:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla\boldsymbol{\theta}$; update the parameters
6: end for
7: return $\boldsymbol{\theta}$; return the trained parameters;
```

The learning rate controls the step size of each update in SGD. If the learning rate is too large, the SGD may fail to find the minimum as shown in Fig. 1.10. If the learning rate is too small, the SGD can be slow to converge (Fig 1.10) or become stuck in a local minimum which has high error (Fig 1.9). Therefore, it is difficult to determine a proper fixed learning rate. Recent studies proposed adaptive learning rates, such as Adam (Kingma and Ba 2014), RMSProp (Tieleman and Hinton 2017), and Adagrad (Duchi et al. 2011), which speed up the training process by automatically adapting the learning rate. Adam is one of the most frequently used algorithm. Instead of using the gradients to update the parameters directly, Adam computes the running average of the gradients and the second moment of the gradients to update the parameters as shown in Algorithm 2. The β_1 and β_2 terms are the forgetting factors, also known as momentum, for the gradients and the second moment of the gradients, respectively. By default, β_1 is 0.9 and β_2 is 0.999 (Kingma and Ba 2014).

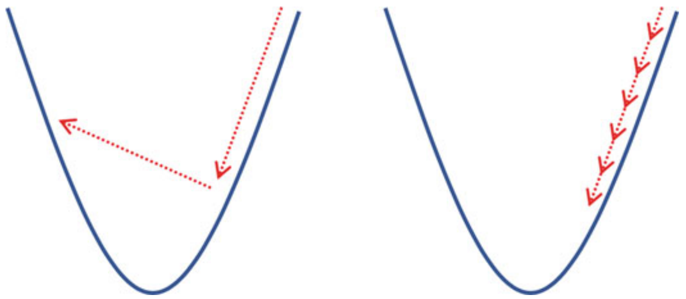


Fig. 1.10 A large learning rate may accelerate the training process but can also make it hard to train a model with ideal parameters. As shown in the left figure, which has a larger learning rate than the right figure, the loss value may increase after parameters update and it can be hard to approach the minimum. In contrast, in the right figure, which has a lower learning rate, the loss value decreases consistently but in a slower manner

Algorithm 2 The training process of Adam optimization

```
Input: parameters $\boldsymbol{\theta}$, learning rate $\alpha$, number of training steps/iterations $S$, $\beta_1 = 0.9$, $\beta_2 = 0.999$,
    $\epsilon = 10^{-8}$
 1: $\boldsymbol{m}_0 \leftarrow 0$; initialize the first moment vector
 2: $\boldsymbol{v}_0 \leftarrow 0$; initialize the second moment vector
 3: for $t = 1$ to $S$ do
 4:     $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$; compute the gradient using a random mini-batch
 5:     $\boldsymbol{m}_t \leftarrow \beta_1 * \boldsymbol{m}_{t-1} + (1 - \beta_1) * \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$; update the first moment
 6:     $\boldsymbol{v}_t \leftarrow \beta_2 * \boldsymbol{v}_{t-1} + (1 - \beta_2) * (\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}})^2$; update the second moment
 7:     $\hat{\boldsymbol{m}}_t \leftarrow \frac{\boldsymbol{m}_t}{1-\beta_1^t}$; compute the running average of the first moment
 8:     $\hat{\boldsymbol{v}}_t \leftarrow \frac{\boldsymbol{v}_t}{1-\beta_2^t}$; compute the running average of the second moment
 9:     $\nabla\boldsymbol{\theta} \leftarrow -\alpha * \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t + \epsilon}}$;
10:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla\boldsymbol{\theta}$; update parameters
11: end for
12: return $\boldsymbol{\theta}$; return the trained parameters
```

1.6.3 Hyper-Parameter Selection

In deep learning, hyper-parameters refer to the settings of a model, such as the number of layers, and the settings of the training process, such as the number of steps, batch size, and learning rate. These settings can significantly affect the performance of a model, so selecting these hyper-parameters appropriately is essential to obtain an ideal model.

To evaluate the performance of different hyper-parameters, the data is usually split into training, validation, and testing sets. Then, multiple hyper-parameter settings are applied to the training set and evaluated on the validation set. Finally, the model with the best hyper-parameters that performs best on the validation set is selected for a final evaluation on the testing set.

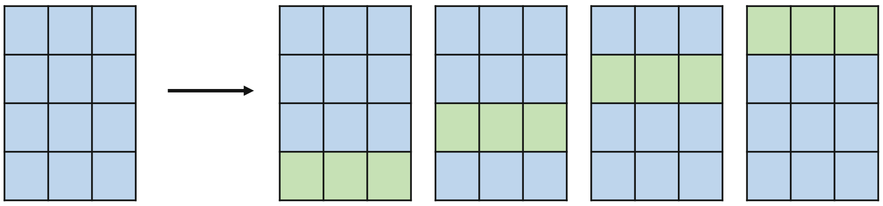


Fig. 1.11 An example of four-fold cross-validation. The dataset is split into four subsets (each row is a subset for demonstration purpose). In each trial, the blue subsets are the training set and the green subset is the testing set. The final evaluation result is the average of the four trials

Cross-Validation

For a small dataset, splitting the data into training and testing sets may be problematic. If the size of the training set is too small, the performance of a model can be harmed since there is no sufficient training data. On the other hand, if the testing set is too small, a model cannot be adequately evaluated. To tackle this problem, cross-validation is introduced.

In a k-fold cross-validation, a dataset is split into k non-overlapping subsets and each subset has the same size. The training/testing process is repeated for k times and, in each time, one of the subsets is selected for testing and the remainders for training. The final evaluation result is then averaged by the result across the k trials. Figure 1.11 illustrates an example of four-fold cross-validation.

1.7 Regularization

Regularization refers to a collection of methods which are designed to make sure a model not only works well on the training set but also on the testing data and new dataset. This section introduces the concept of overfitting and some regularization methods including weight decay, dropout, and batch normalization.

1.7.1 Overfitting

A machine learning model is optimized to minimize the training error (i.e., loss) but this cannot guarantee that the model can also perform well on the testing data. If the model is optimized "overly," the model may even have a significantly large testing error. This case is called overfitting. For example, in Fig. 1.12, the polynomial model represented by the dashed curve suffers from overfitting. This model fits the training data accurately but it fails to fit the testing data. Such a model with overfitting can be unreliable in real-world applications where there is always new data. In contrast,

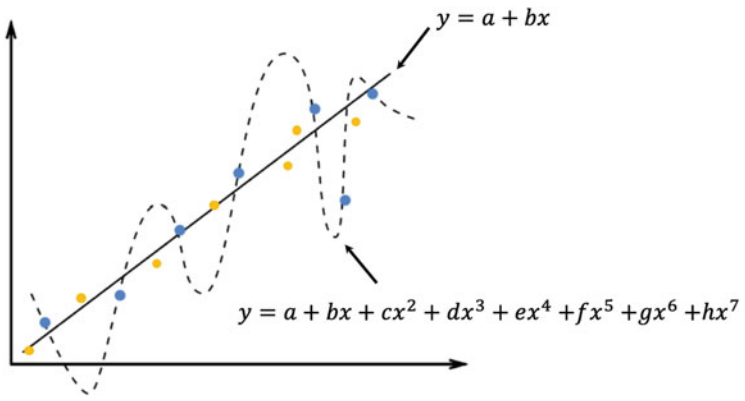


Fig. 1.12 A demonstration of overfitting. The blue dots represent training data, and the orange dots are testing data. Though the linear model represented by the solid line has a larger training error, it has much smaller testing error than the polynomial model represented by the dashed curve. Thus, we can say the polynomial model suffers from overfitting

the linear model represented by the solid straight line has fewer parameters while offering a better fit for the testing data.

Underfitting is opposite to overfitting, where the model cannot fit the training data, resulting in large error for both training and testing data. However, in practice, underfitting can be solved by using a larger model (more layers, more parameters, etc.), but solving overfitting is more challenging. The simplest way to alleviate overfitting is to use more training data which is not always possible since data acquisition and data labeling can be expensive.

1.7.2 Weight Decay

Weight decay is a simple but yet effective regularization method targeting the overfitting problem. It introduces a regularization term as a penalty to encourage $\boldsymbol{\theta}$ with smaller absolute values. For example, as Fig. 1.12 shows, if the parameters from c to h of the polynomial model have smaller absolute values, the model will have a lower swing range so that it can better fit the data. The loss function with the parameter norm penalty is defined as follows:

$$\mathcal{L}_{\text{total}} = \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) + \lambda\Omega(\theta), \tag{1.23}$$

where $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ is the original loss function computed from the target $\boldsymbol{y}$ and prediction $\hat{\boldsymbol{y}}$, Ω is the parameter norm penalty function and λ is a small value that controls the strength of the regularization. Two of the most commonly used parameter norm penalty functions are $\mathcal{L}_1 = \|\boldsymbol{W}\|$ and $\mathcal{L}_2 = \|\boldsymbol{W}\|_2^2$. The parameters

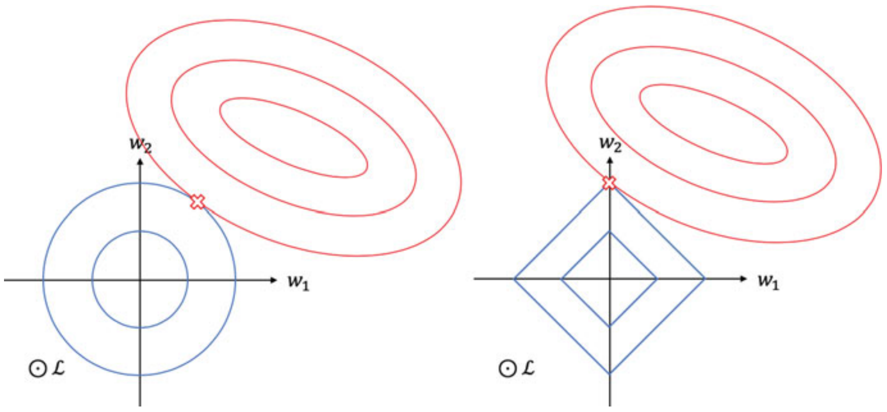


Fig. 1.13 Left: A demonstration of contour lines of the original loss (red) and $\mathcal{L}_2$ (blue). Right: A demonstration of contour lines of the original loss (red) and $\mathcal{L}_1$ (blue). The interaction points (red crosses) of the two contour lines in each sub-figure indicate that $\mathcal{L}_1$ may tend to produce parameters valued zero and $\mathcal{L}_1$ may produce parameters with similar absolute values

of deep neural networks often have absolute values smaller than 1, so $\mathcal{L}_1$ can lead to a large penalty than $\mathcal{L}_2$ since $|w| > w^2$ when $|w| < 1$. Therefore, the loss function with $\mathcal{L}_1$ has the property which encourages the parameters of a network to have rather small values, or even zeros. This enables the network to implicitly perform feature selection, i.e. discarding some input features by setting the corresponding parameters to zero or some small values. As Fig. 1.13 shows, given two parameters w_1, w_2, in the coordinate system, ${w_1}^2 + {w_2}^2 = r^2$ is a circle with radius of r and $|w_1| + |w_2| = r$ is a square with diagonal length of $2r$, both of which are demonstrated by the blue contour lines. The red contour lines indicate the original loss $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$. The intersection points, represented by the red crosses, of the parameter norm penalties and the original loss, indicate that $\mathcal{L}_1$ is more likely to produce parameters valued zero than $\mathcal{L}_2$, while $\mathcal{L}_2$ may produce parameters with similar absolute values.

1.7.3 Dropout

Deep neural networks with large numbers of neurons can suffer from the co-adaptation of neurons which can result in overfitting. The co-adaptation of neurons means that the neurons are dependent on each other. If one of the neurons fails, all dependent neurons may also fail and this can lead to the failure of the entire neural network. Dropout (Hinton et al. 2012; Srivastava et al. 2014) is a popular technique to address this problem by preventing the co-adaptation of neurons (i.e., parameters). To prevent the co-adaptation of parameters, during training, the hidden outputs are randomly set to zero, which resembles a random disconnection

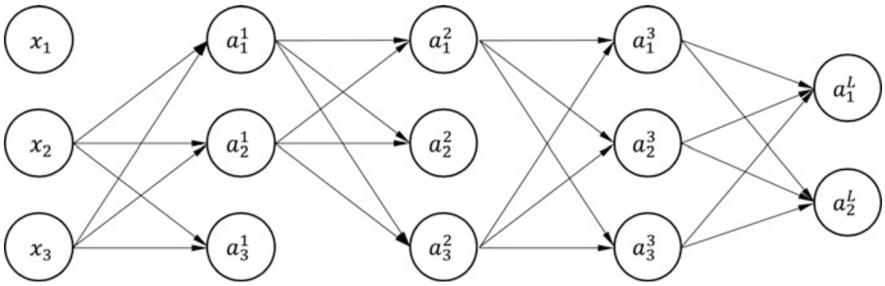


Fig. 1.14 Applying dropout to MLP where some neurons are randomly deactivated

of neurons from one layer to the next, as illustrated in Fig. 1.14. During back-propagation, with a zero-valued output $\boldsymbol{a}$, the corresponding partial derivative of the loss with respect to the layer output $\boldsymbol{\delta}$ will be zero. In other words, only the remaining connected neurons are updated. Therefore, the dropout method can train different sub-networks while allowing all of them to share the same parameters (Hinton et al. 2012). During testing, dropout is disabled, and no outputs are set to zero. This means that all sub-networks work together to predict the final result (i.e., ensemble learning (Hara et al. 2016)). The theoretical proof of dropout was not presented in the original work by Hinton et al. (2012), but more recent studies proved its effectiveness in ensemble learning (Hara et al. 2016) and Bayesian approximation (Gal and Ghahramani 2016).

1.7.4 Batch Normalization

Batch normalization (Ioffe and Szegedy 2015) normalizes the inputs of a layer to have a mean of 0 and a variance of 1 and can improve the performance of a neural network and its training stability. Specifically, during training, the batch normalization layer estimates the mean and variance of the batch inputs using a moving average. Then, the moving mean and variance are updated to normalize the batch inputs. During testing, the moving mean and variance are fixed and applied to normalize the inputs.

Besides improving the performance and stability, batch normalization provides regularization. Similar to the dropout process that adds a random factor to the hidden values, the moving mean and variance of batch normalization introduce randomness as they are updated in each iteration according to the random mini-batch. Therefore, a neural network is encouraged during training to be robust enough to deal with the variation (Fig. 1.15).

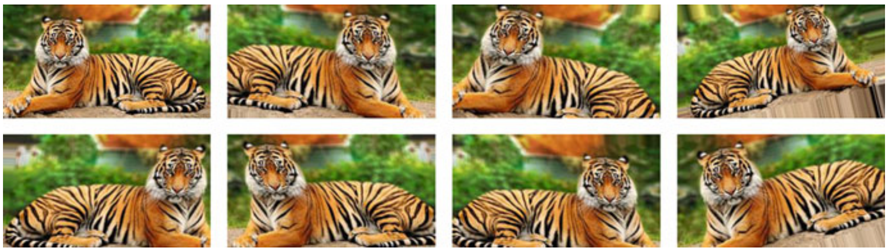


Fig. 1.15 An example of image data augmentation. The top-left image is the original image and the others are obtained by random flip, rotate, shear, shift, and zoom on the original image

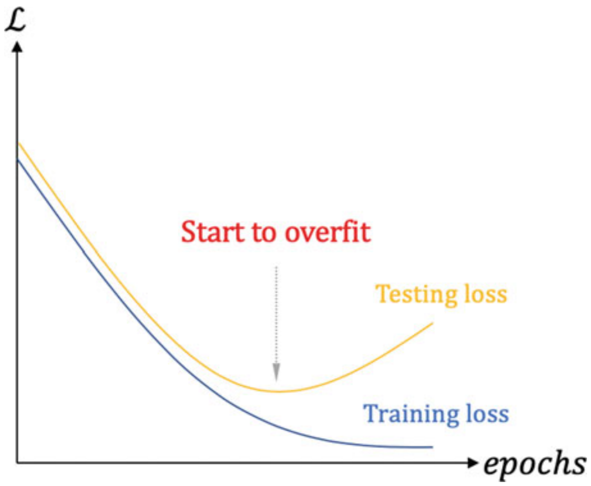


Fig. 1.16 A demonstration of where the overfitting starts. The early stopping can be applied so that the training process is terminated before the overfitting starts

1.7.5 *Other Methods for Alleviating Overfitting*

There are many other methods designed to prevent overfitting, such as early stopping and data augmentation. Early stopping allows early termination of the training process once it matches an empirical criterion, such as a threshold of accuracy on the validation set. Figure 1.16 shows that the testing loss may start to increase during training (i.e., the overfitting starts) and early stopping can be applied so that the training process is terminated before the overfitting starts. Data augmentation increases the size of training data by augmenting the existing training data. For example, image data can be augmented by simply flipping, rotating, shifting, and zooming. Data augmentation methods that generate arbitrary but reasonable data can reduce overfitting and improve the performance of a model (Simonyan and

Zisserman 2015; He et al. 2016; Howard et al. 2017; Dong et al. 2017b). As with an image, the audio can be augmented by adding noise or perturbation. A recent study by Ko et al. (2015) showed that audio data augmentation with speed perturbation can improve the performance of speech recognition algorithms.

However, it is not applicable to use similar augmenting transformations on textual data since the order of words provides specific meaning. For example, "people like dogs" is not semantically equivalent to "dogs like people." A practical way to augment textual data can be rephrasing sentences by replacing words with pre-defined synonyms (Zhang et al. 2015). Moreover, instead of augmenting the raw textual data, another study (Reed et al. 2016) interpolates the text embeddings of two random sentences so that the model is aware of the gaps in the text latent space.

1.8 Convolutional Neural Networks

Convolutional neural networks (CNNs) (LeCun et al. 1989) are a variant of MLP and are particularly useful in computer vision (Krizhevsky et al. 2012; Simonyan and Zisserman 2015; He et al. 2016), time series prediction (van den Oord et al. 2016), natural language processing (Zhang et al. 2019a; Yin et al. 2017), and also reinforcement learning (Rusu et al. 2016; James et al. 2019). Many of the deployed real-world machine learning systems are built on CNNs, which often demonstrate far superior performances when being compared against those with conventional methods. In this section, we introduce two kinds of layers, namely convolutional layer and pooling layer, which are commonly used to construct CNNs.

Convolutional Layer The convolutional layer has the most distinguishable feature of CNNs. The idea of its design stems from the study of the human brain again where we have an array of nearby neurons processing a subset of the visual input. Concretely, as Fig. 1.17 has shown, the convolution volume uses four different neurons to process the same region from the input image. Different neurons could be responsible for different tasks such as edge, color, or angle detection. The neuron in the convolution input is locally connected rather than being connected to all units from the previous layer. Convolutional layers can also be stacked one by one, which means a convolutional layer can be applied to the output from another convolutional layer. The benefit of a convolutional layer is that it has far fewer connections to the previous layer than a dense layer so the convolutional layer typically can be trained more quickly. Figure 1.17 also shows that each neuron in a convolutional layer contains all the information of a small region and across all channels. For example, if the input layer is the RGB image input layer, then a neuron in the convolutional layer has the information after the filter is applied to a small region of the image across all the RGB image channels.

Regarding the convolution operation inside the convolutional layer, it uses filters to extract various important features. A layer has an input of height/width W. When

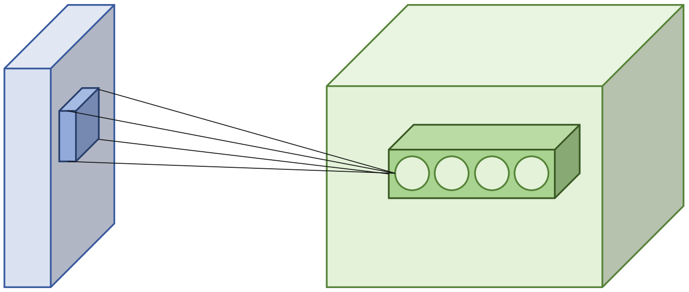


Fig. 1.17 Computation of the convolution volume from a sample image. There are four neurons applied to the same region in this example

we convolve an input with a filter of size F, we simply compute a dot product between the input and the filter values in a sliding window fashion. Then we move to apply the filter to the next block. The stride S describes how far each input block is away from each other. For instance, with the stride of two ($S = 2$), the filter is applied to the block that is one element away, skipping one row/column essentially. Lastly, sometimes in order to ensure that boundary values are well-considered, we have to add zeros on the edge, namely padding. We let the padding size be P. The output volume size of a convolutional layer can be computed by

$$\left\lfloor \frac{W - F + 2P}{S} + 1 \right\rfloor \tag{1.24}$$

The output volume has the same depth (number of output channels) as the number of filters. Figure 1.18 shows a concrete example of the convolution operation. In this example, there is an image of size 4×4 (height $\times$ width) with 3 input channels (RGB), and 1 filter sized $3 \times 3 \times 3$ (filter height $\times$ filter width $\times$ input channels) with a stride $S = 1$ and a padding $P = 0$. According to Eq. (1.24), the output height/width is $(4 - 3 + 0)/1 + 1 = 2$. The depth of the output (number of output channels) is 1 since there is 1 filter. To compute the top-left value in each channel, we first compute the dot products between the input image and the filter, which generate three values, and then sum up the three values to produce the top-left value. The convolution operation is a special case of $\sum_i w_i x_i$, where w_i is non-zero in a much smaller set. The output can then be passed through an activation function which introduces non-linearity.

Pooling Layer Pooling takes advantage of the fact that, for images, neighboring pixels are similar. So it is assumed that proper down-sampling, such as only retaining the maximum or the average of a small region, is beneficial for modeling. There are typically two types of pooling layers to reduce the dimensions, namely max-pooling and average-pooling. In Fig. 1.19, we are showing examples of max-pooling and average-pooling on a 4×4 input with a stride of 2. The pooling layer

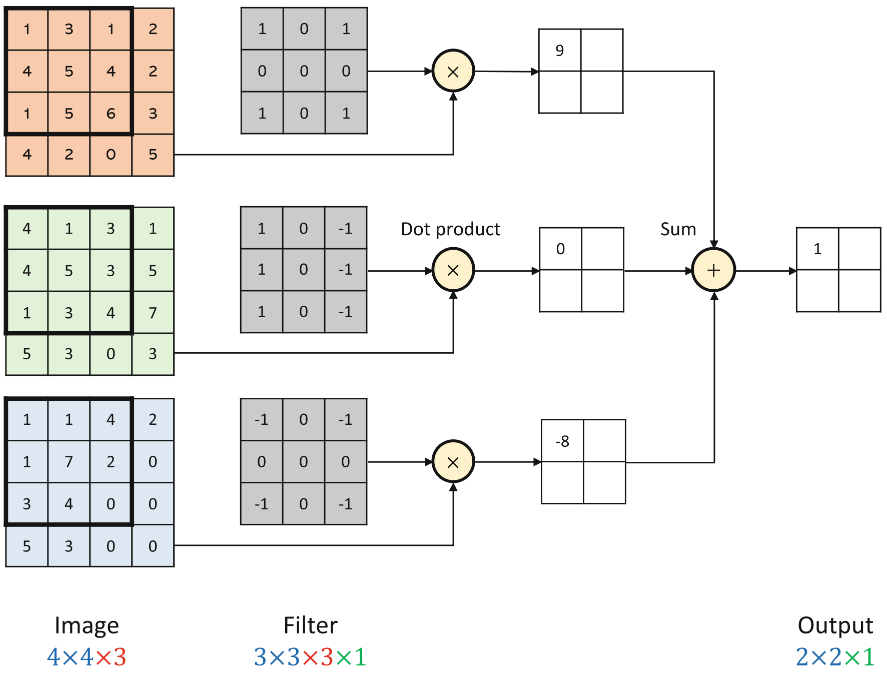


Fig. 1.18 Illustration of the convolution operation. In this example, 1 filter with size 3×3×3 (filter height × filter width × input channels) is applied on an image sized 4 × 4 (height × width) with 3 input channels (RGB). The dot products between the image and the filter are computed across the channels. The values obtained from the dot products are summed up to produce the top-left value of the output

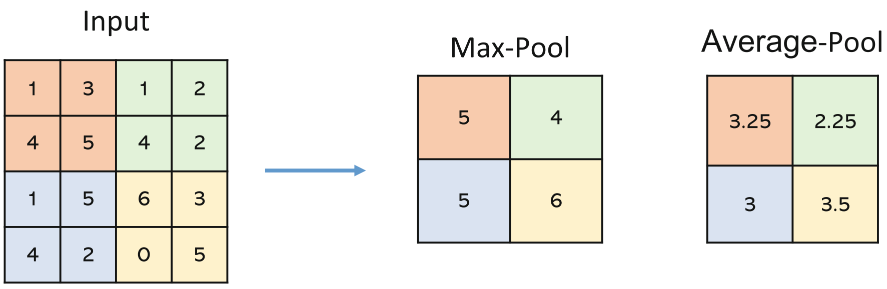


Fig. 1.19 2 × 2 max-pooling and average-pooling examples with a stride of 2 on a 4 × 4 input

reduces the dimensions of the output significantly, which makes computation in the following layers more efficient. For example, there can be hundreds of channels after a convolutional layer. Before the output is passed to a dense layer, reducing the dimensions of the output by pooling is preferred so that the successive dense layer has less computation workload.

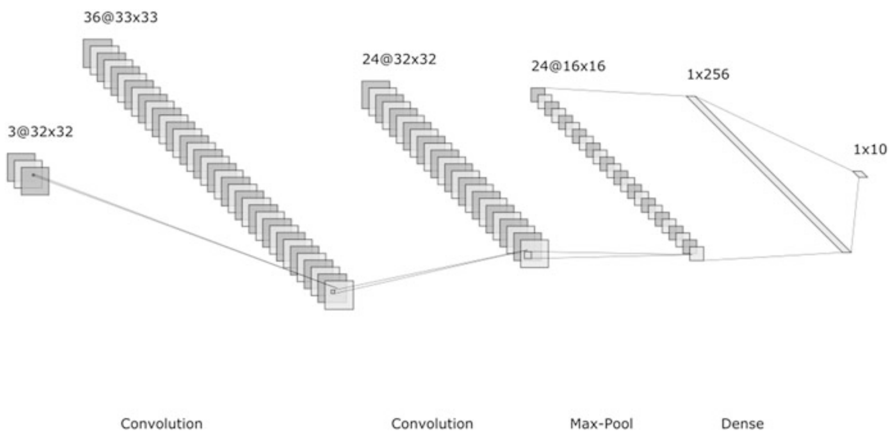


Fig. 1.20 A example of CNN with two convolutional layers, a max-pooling layer, and a dense layer. Figure created by NN-SVG[1]

Overall, the convolutional layer and pooling layer together with the dense layer are the basic components to construct CNNs. Figure 1.20 demonstrates a CNN with two convolutional layers, a max-pooling layer, and a dense layer. Note that activation functions can be applied to the output of the convolutional layers in the same way as the dense layer.

CNNs adopt the idea of **parameter sharing** which is different from MLP. The parameter sharing across different parts of a model makes the model more efficient (fewer parameters and less memory) and possible to handle variable data forms (different lengths and sizes). Recall that, in a dense layer, there is a weight matrix whose element w_{ij} denotes the connectivity between the i-th neuron in the previous layer and the j-th neuron in the current layer. However, in a convolutional layer, the filters are essentially weights, which are used repeatedly when the output values are being computed. The repeated usage of filters reduces the number of parameters needed in a convolutional layer and this is why a convolutional layer typically has far fewer parameters than a dense layer given similar sizes of input and output.

Batch normalization (batch-norm layers) (Ioffe and Szegedy 2015) can be integrated with CNNs to accelerate the training due to the internal covariate shift. As mentioned above, the input of a batch-norm layer is normalized by a mean and a variance, which are independent of other layers. Therefore, intuitively, the batch normalization simplifies the interactions between layers in the gradient update and allows larger learning rates which accelerate the training.

LeNet (LeCun et al. 1998), AlexNet (Krizhevsky et al. 2012), and VGGnet (Simonyan and Zisserman 2015) are some popular CNNs. How to design the architecture of CNNs for a specific task or a general scenario is still an on-going

[1] http://alexlenail.me/NN-SVG/LeNet.html.

research topic. The design can be an empirical driven exercise and requires lots of trials. However, recent works in neural architecture search seem to have provided more insights (Zoph and Le 2016; Zoph et al. 2018).

1.9 Recurrent Neural Networks

Recurrent neural networks (RNNs) (Rumelhart et al. 1986) is another class of deep learning architectures and it is designed to process sequential data. Unlike the images which can be represented by a grid of values, the sequential data refers to a sequence of values $\{x_1, x_2, \ldots, x_n\}$, which is also a common data format. For example, a document is composed of a sequence of words, and the values of a stock can be represented by a sequence of stock prices.

An important feature of the sequential data is the interaction among elements within the sequence. For example, provided with a snippet of text, a human reader may easily infer the content that would come next by only reading the beginning. However, the modeling of such interaction within the sequence can be more challenging if the sequence is longer. Therefore, RNNs should be able to effectively accumulate information provided by the sequential data and adequately consider the impact of earlier values on later ones in the sequence.

The design of RNNs, like that of CNNs, also adopts parameter sharing. The use of parameter sharing allows the same weight to be utilized repeatedly across multiple locations in the input sequential data. For example, RNNs should be able to learn that the sentences “Deep learning has been popular since the 2010s.” and “Since the 2010s, deep learning has been popular.” express the same meaning even though the positions of words are different. Similarly, when the CNNs are used to classify an image of a cat, the position of the cat in the image should not change the decision made by the CNNs (Fig. 1.21).

Simple Cell Similar to the CNNs which can process images of variable sizes, the RNNs can also easily be adjusted to process sequences with variable lengths. The

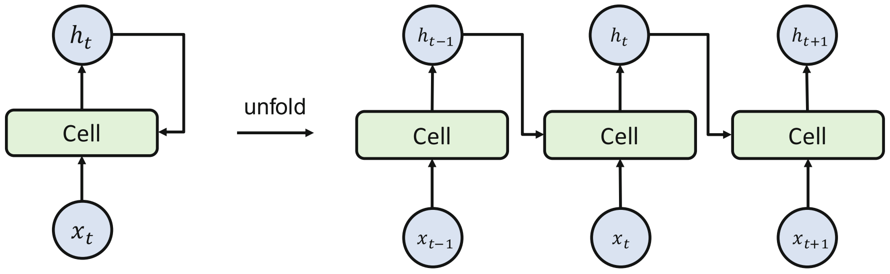


Fig. 1.21 An illustration of RNN architecture. The cell ingests the value $\boldsymbol{x}_t$ and the previous hidden state $\boldsymbol{h}_{t-1}$, and then outputs the new hidden state $\boldsymbol{h}_t$

idea of RNNs is to define a computation unit, referred to as a cell, and the cell is repeatedly computed given each value in the sequence one by one. The cell has a state which accumulates the information so far. When the cell is computed, it takes a value from the sequence and the previous state of the cell as inputs, and then generates a new state, which will be used in the next computation round. The simplest RNN cell applies a linear transformation which can be defined as follows:

$$\boldsymbol{h}_t = \boldsymbol{W}[\boldsymbol{x}_t; \boldsymbol{h}_{t-1}] + \boldsymbol{b} \tag{1.25}$$

In this equation, the previous state of the cell $\boldsymbol{h}_{t-1}$ is concatenated with the value $\boldsymbol{x}_t$ and then multiplied by the linear kernel $\boldsymbol{W}$. A bias $\boldsymbol{b}$ can also be added to the state. An RNN constructs a deep computational graph as the linear kernel is repeatedly multiplied. Such a deep computational graph may cause the exploding of gradients if the eigenvalues of $\boldsymbol{W}$ are greater than 1 in magnitude or vanishing of gradients if the eigenvalues are less than 1 in magnitude. The exploding of gradients can make the learning process volatile while the vanishing of gradients can make the optimization of objectives (cost or loss) less effective. The RNNs with the simple cell may suffer from either problem if the input sequence is lengthy.

LSTM The long short-term memory networks or LSTMs (Hochreiter et al. 1997) are more sophisticated RNNs to handle the long-term dependencies in long sequences, and the LSTM computation can serve as a cell in RNNs.

Unlike the simple cell, the LSTM cell has two states: cell state $\boldsymbol{C}_t$ and hidden state $\boldsymbol{h}_t$. The update process of the cell state forms an information highway (the orange line in Fig. 1.22) which runs across the entire sequence with simple computations. This feature allows an easier flow of information throughout the sequence so that the dependency between two values that are located far away from each other in the sequence (i.e., long-term dependency) can be properly considered. Meanwhile, the hidden state is involved with gated computations. The gate controls

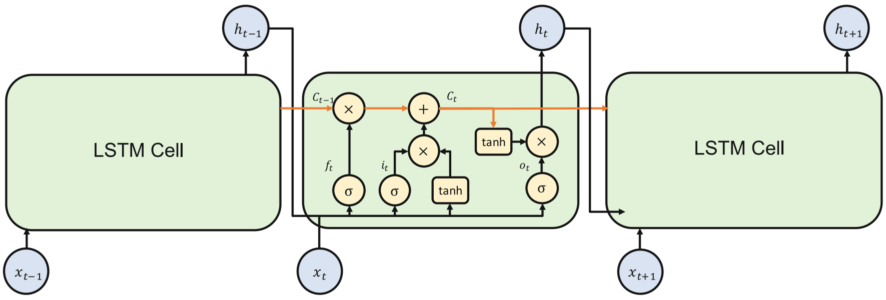


Fig. 1.22 An illustration of RNN with the LSTM cell. There are two states in the LSTM which are the cell state $\boldsymbol{C}_t$ and the hidden state $\boldsymbol{h}_t$. In addition, the three gates control whether information should be removed or added. Figure reproduced based on Olah (2015)

whether to forget or add information to the flow and is implemented by the sigmoid function. The output of the sigmoid function is restricted between 0 and 1. In other words, when the sigmoid function outputs 1, the corresponding information should be totally kept. In contrast, the corresponding information should be totally forgotten if the sigmoid function outputs 0.

There are three kinds of gates in an LSTM cell: the forget gate, input gate, and output gate. The forget state first determines whether certain information should be removed from the cell state based on the new input. In addition, the input gate controls whether the new input should be added into the cell state for longer storage and also for a replacement to any information which has been forgotten. Then finally, the output state decides what the cell should output based on the new cell state. The three gates and the computation within the LSTM cell can be formally defined as follows. Note that σ represents the sigmoid function.

$$\begin{aligned}
\text{Forget gate:} \quad \boldsymbol{f}_t &= \sigma(\boldsymbol{W}_f[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t] + \boldsymbol{b}_f) \\
\text{Input gate:} \quad \boldsymbol{i}_t &= \sigma(\boldsymbol{W}_i[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t] + \boldsymbol{b}_i) \\
\text{Output gate:} \quad \boldsymbol{o}_t &= \sigma(\boldsymbol{W}_o[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t] + \boldsymbol{b}_o) \\
\text{Update cell state:} \quad \boldsymbol{C}_t &= \boldsymbol{f}_t \times \boldsymbol{C}_{t-1} + \boldsymbol{i}_t \times \tanh(\boldsymbol{W}_C[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t] + \boldsymbol{b}_C) \\
\text{Update hidden state:} \quad \boldsymbol{h}_t &= \boldsymbol{o}_t \times \tanh(\boldsymbol{C}_t)
\end{aligned} \tag{1.26}$$

There is a family of gated RNNs that uses gated recurrent units (or GRUs) and the LSTM is a member of this family. Recent works have investigated different RNN architectures but it is still unclear which one is clearly better than others (Cho et al. 2014; Jozefowicz et al. 2015).

RNNs are widely adopted in deep learning to process sequential data like natural language and time series (Liao et al. 2018b; Chung et al. 2014; Mikolov et al. 2010) and also applied to solve reinforcement learning problems (Peng et al. 2018; Wierstra et al. 2010). Based on the relations between inputs and outputs, the architecture of RNNs can be modified in different scenarios. For example, a typical example of sequence input and single output is text classification (Zhang et al. 2019a; Lee and Dernoncourt 2016) where the input is a sequence of words (a sentence or a document) and the output is a single label to represent the predicted class. More challenging tasks such as machine translation (Sutskever et al. 2014; Luong et al. 2015; Bahdanau et al. 2015) and text summarization (Nallapati et al. 2017) have a sequence input and a sequence output.

1.10 Deep Learning Examples

This section introduces examples of how to implement deep learning models in TensorFlow[2] and TensorLayer.[3] TensorFlow (Abadi et al. 2016) by Google is an open-source library that enables researchers and engineers to develop deep learning models, while TensorLayer (Dong et al. 2017a) provides a moderate abstraction over TensorFlow to make such development easier and more flexible. The content of this section is validated on Python 3, TensorFlow 2.0, and TensorLayer 2.0 or later. In the future, TensorLayer will support different computational backend not only TensorFlow.

1.10.1 Tensor and Gradients

The tensor is the most fundamental computation unit in TensorFlow and it is used to represent outputs of an operation. A tensor can be created by operations such as `tf.constant`, `tf.matmul`, etc. Tensor does not store the values of the operation's outputs but provides access to the computation of those values in a TensorFlow session. In TensorFlow 2.0, there is no need to run a session manually, as in eager execution, graphs and sessions are designed to stay in the backend. For examples, in the matrix multiplication as shown below, matrices can be created by `tf.constant` and the multiplication is computed by `tf.matmul` whose output is another matrix.

Matrix multiplication in TensorFlow by Tensor.

```
>>> import tensorflow as tf
>>> a = tf.constant([[1, 2], [1, 2]])
# tf.Tensor(
# [[1 2]
#  [1 2]], shape=(2, 2), dtype=int32)
>>> b = tf.constant([[1], [2]])
# tf.Tensor(
# [[1]
#  [2]], shape=(2, 1), dtype=int32)
>>> c = tf.matmul(a, b)
# tf.Tensor(
# [[5]
#  [5]], shape=(2, 1), dtype=int32)
```

In the forward propagation of deep neural networks, the tensors are automatically connected by each other as a graph. Based on the graph and the automatic

[2] https://github.com/tensorflow/tensorflow.

[3] https://github.com/tensorlayer/tensorlayer.

differentiation technique provided by TensorFlow, gradients can be computed in the back-propagation. TensorFlow 2.0 provides `tf.GradientTape` to compute gradients of recorded operations with respect to its input variables. For example, the code below shows an example of computing gradients in back-propagation. The forward propagation and the computation of loss are within the scope of `tf.GradientTape`, while the back-propagation and the update of weights are outside the scope. The `tf.GradientTape` records all operations that are executed within the scope onto a `tape`. Then the gradients associated with each recorded operation and its input variables are computed by reverse-mode automatic differentiation. Once the function `tape.gradient()` is called, the resources held by `tf.GradientTape` are released.

Gradients computation in TensorFlow and TensorLayer.

```python
import tensorflow as tf
import tensorlayer as tl
def train(model, dataset, optimizer):
   # given a model which is an instance of Model by TensorLayer
   # traverse the dataset where x is input and y is target output
   for x, y in dataset:
      # create the scope of gradient tape
      with tf.GradientTape() as tape:
         prediction = model(x) # forward propagation
         loss = loss_fn(prediction, y) # loss function
      # back-propagation and computing gradients, then the
          resources held by the GradientTape are released
      gradients = tape.gradient(loss, model.trainable_weights)
      # apply the gradients to weights and update the weights by
          the optimizer
      optimizer.apply_gradients(zip(gradients,
          model.trainable_weights))
```

1.10.2 Define a Model

In TensorLayer 2.0, `Model` is an entity that consists of multiple `Layers` and defines the propagation between the `Layers`. TensorLayer 2.0 provides two sets of APIs to define a model. Static model APIs allow users to build a model fluently and dynamic model APIs provide more flexibility in the forward propagation. A static model requires users to manually construct a graph and compile it. Once the model is compiled, the forward propagation cannot be changed. Unlike the static model, the dynamic model can be executed eagerly like Python normally does and the forward propagation is mutable.

In the implementation of models, as shown in the examples below, the difference between a static model and a dynamic model can be summarized in two aspects. First, when layers in a static model are declared, the connection between layers (i.e., the forward propagation) is defined explicitly at the same time. Based on the connection, for each layer, TensorLayer can automatically infer the size of input variables from previous layers and then construct weights. When the `Model` is finally instanced, only inputs and outputs need to be specified and TensorLayer automatically builds a graph based on the connection. However, when a dynamic model is initialized, the forward propagation is still unknown until it is defined in the function `forward` later. Thus, the size of input variables cannot be automatically inferred and it has to been manually provided via the argument `in_channels`.

Second, the forward propagation of a static model is fixed once the model is constructed, so it is easier to accelerate the computation of a static model. TensorFlow 2.0 provides a new feature called `tf.function` which can be used as a decorator and accelerate the computation. Unlike the static model, the forward propagation in a dynamic model can be more flexible. For example, the forward flow can be controlled based on input values or arguments specified by users. Users are also allowed to use or abandon any layer in the forward propagation of a dynamic model.

An example of a static model: multilayer perceptron (MLP)

```
import tensorflow as tf
from tensorlayer.layers import Input, Dense
from tensorlayer.models import Model

# a multilayer perceptron (MLP) model with three dense layers
def get_mlp_model(inputs_shape):
   ni = Input(inputs_shape)
   # since the connection between layers is explicitly defined
   # in_channels of each layer is automatically inferred
   nn = Dense(n_units=800, act=tf.nn.relu)(ni)
   nn = Dense(n_units=800, act=tf.nn.relu)(nn)
   nn = Dense(n_units=10, act=tf.nn.relu)(nn)
   # automatic build a model based on the connection between
       layers
   M = Model(inputs=ni, outputs=nn)
   return M

MLP = get_mlp_model([None, 784])
# switch to evaluation mode
MLP.eval()
# ingest data into the model
# the computation can be accelerated by using @tf.function in
    TensorFlow 2.0
outputs = MLP(data)
```

An example of a dynamic model: multilayer perceptron (MLP)

```
import tensorflow as tf
from tensorlayer.layers import Input, Dense
from tensorlayer.models import Model
```

```
class MLPModel(Model):
   def __init__(self):
      super(MLPModel, self).__init__()
      # since the connection between layers is unknown so far,
          in_channels has to be manually provided
      # assume the input data is size 784
      self.dense1 = Dense(n_units=800, act=tf.nn.relu,
          in_channels=784)
      self.dense2 = Dense(n_units=800, act=tf.nn.relu,
          in_channels=800)
      self.dense3 = Dense(n_units=10, act=tf.nn.relu,
          in_channels=800)

   def forward(self, x, foo=False):
      # define the forward propagation
      z = self.dense1(z)
      z = self.dense2(z)
      out = self.dense3(z)
      # control the forward flow in a dynamic model
      if foo:
         out = tf.nn.softmax(out)
      return out

MLP = MLPModel()
# switch to evaluation mode
MLP.eval()
# ingest data into the model
# the argument foo controls the forward flow
outputs_1 = MLP(data, foo=True) # with softmax
outputs_2 = MLP(data, foo=False) # without softmax
```

1.10.3 Customized Layers

TensorLayer 2.0 provides more than a hundred layers for users, and at the same time, TensorLayer 2.0 also supports `Lambda Layer` so that users can easily customize layers. The simplest example is to pass a lambda function into a `Lambda Layer` as shown below. Users may also define a customized function with arguments and the arguments can be passed by `fn_args` when the `Lambda Layer` is initialized or called.

```
import tensorlayer as tl
x = tl.layers.Input([8, 3], name='input')
y = tl.layers.Lambda(lambda x: 2*x)(x) # this layer has no
    trainable weights.

def customize_fn(input, foo): # arguments can be set by fn_args
    in Lambda Layer.
   return foo * input
```

```
z = tl.layers.Lambda(customize_fn, fn_args={'foo': 42})(x) #
    this layer has no weights.
```

The `Lambda Layer` can also have trainable weights. The example below shows that the weight is defined outside the customized function and it should be passed into the `Lambda Layer` by `fn_weights`.

```
import tensorflow as tf
import tensorlayer as tl
a = tf.Variable(1.0) # weight which is defined outside the scope
    of the customized function.
def customize_fn(x):
   return x + a
x = tl.layers.Input([8, 3], name='input')
y = tl.layers.Lambda(customize_fn, fn_weights=[a])(x) # weights
    are passed by fn_weights, which should be a list.
```

Moreover, the `Lambda Layer` enables the compatibility of Keras in TensorLayer. Users may define a Keras model and pass the model into a `Lambda Layer` as a function since the Keras model is callable. The trainable weights of the Keras model need to be fetched and then passed into the `Lambda Layer` so that the Keras model can be updated together with the customized model.

```
import tensorflow as tf
import tensorlayer as tl
# define a Keras model
layers = [
   tf.keras.layers.Dense(10, activation=tf.nn.relu),
   tf.keras.layers.Dense(5, activation=tf.nn.sigmoid),
   tf.keras.layers.Dense(1, activation=tf.identity)
]
perceptron = tf.keras.Sequential(layers)
# in order to get trainable_variables of keras
_ = perceptron(np.random.random([100, 5]).astype(np.float32))

class CustomizeModel(tl.models.Model):
   def __init__(self):
      super(CustomizeModel, self).__init__()
      self.dense = tl.layers.Dense(in_channels=1, n_units=5)
      self.lambdalayer = tl.layers.Lambda(perceptron,
          perceptron.trainable_variables) # pass the trainable
          weights of the model into the Lambda layer.

   def forward(self, x):
      z = self.dense(x)
      z = self.lambdalayer(z)
      return z
```

1.10.4 MLP: Image Classification on MNIST

With the `Models`, `Layers`, and other supportive APIs provided by TensorLayer 2.0, users can design and implement their own deep learning models in a straightforward and flexible manner. To help readers have a better understanding of how to write a deep learning model by TensorLayer, let us start from an MLP to classify images on the MNIST dataset (LeCun et al. 1998), which collects 70,000 images of handwritten digits. The implementation of a deep learning example typically has five steps including data loading, building a model, training, testing, and saving the model.

TensorLayer provides APIs in the submodule `tl.files` to load various popular datasets including MNIST, CIFAR10, PTB, CelebA, etc. For example, the MNIST dataset can be loaded by `tl.files.load_mnist_dataset` with a specific shape. The datasets are typically split into three subsets: the training set, validation set, and testing test.

```
# Loading the MNIST dataset by TensorLayer
X_train, y_train, X_val, y_val, X_test, y_test =
    tl.files.load_mnist_dataset(shape=(-1, 784)) # each image in
    MNIST is originally sized 28x28, i.e. has 784 pixels.
```

As introduced in the Sect. 1.10.2, an MLP model can be implemented as a either static or dynamic model in TensorLayer 2.0. In this example, the MLP model is designed to have three `Dense` layers and is implemented as a static model. But unlike a conventional MLP, the MLP model in this example also has three `Dropout` layers, which are used to prevent overfitting.

```
# build the model
ni = tl.layers.Input([None, 784]) # the input is aligned with
    the shape of data
# the layers of the MLP is connected one by one
nn = tl.layers.Dropout(keep=0.8)(ni)
nn = tl.layers.Dense(n_units=800, act=tf.nn.relu)(nn)
nn = tl.layers.Dropout(keep=0.5)(nn)
nn = tl.layers.Dense(n_units=800, act=tf.nn.relu)(nn)
nn = tl.layers.Dropout(keep=0.5)(nn)
nn = tl.layers.Dense(n_units=10, act=None)(nn)
# create the model with specified inputs and outputs
network = tl.models.Model(inputs=ni, outputs=nn, name="mlp")
```

The training of the MLP model on the MNIST dataset is to learn the weights of the model. Users can trigger the training process by simply calling the function `tl.utils.fit`. In addition, the testing step is to validate if the model has properly learned from the data and can be triggered by `tl.utils.test`.

```
# Define a metric to evaluate the accuracy of the model.
# Different from the loss function, the metric is NOT used to
    backpropagate or update the model.
```

```
def acc(_logits, y_batch):
   return tf.reduce_mean(
      tf.cast(
         tf.equal(
            tf.argmax(_logits, 1),
            tf.convert_to_tensor(y_batch, tf.int64)),
         tf.float32),
      name='accuracy'
   )

# Training
tl.utils.fit(
   network, # the model
   train_op=tf.optimizers.Adam(learning_rate=0.0001), # the
       optimizer
   cost=tl.cost.cross_entropy, # the loss function
   X_train=X_train, y_train=y_train, # training set
   acc=acc, # the metrics to evaluate the accuracy of a model
   batch_size=256, # the size of mini-batch
   n_epoch=20, # number of epoch to train
   X_val=X_val, y_val=y_val, eval_train=True, # validation set
)

# Testing
tl.utils.test(
   network, # the model just trained
   acc=acc, # the metrics to evaluate the accuracy of a model
   X_test=X_test, y_test=y_test, # testing set
   batch_size=None, # the size of mini-batch. If None, the whole
       testing set is fed into the network together, so only set
       it None when the testing set is small.
   cost=tl.cost.cross_entropy # the loss function
)
```

Finally, the weights of the trained MLP model can be saved to a local file so that the model can be restored later for inference.[4]

```
# save network weights to a file
network.save_weights('model.h5')
```

1.10.5 CNN: Image Classification on CIFAR10

The CIFAR-10 dataset (Krizhevsky et al. 2009) was a challenging and popular benchmark for image classification. It collects images from 10 classes and each

[4] The full code of the MLP example is available at https://github.com/tensorlayer/tensorlayer/tree/master/examples/basic_tutorials.

class has 6000 images. The images are sized 32 × 32 with RGB color and each image exclusively focuses on one single object (class) such as a dog, airplane, ship, etc. In TensorLayer 2.0, CIFAR-10 can be easily loaded and augmented by using `Dataset` and `Dataloader` APIs.

```
# pre-defined data augmentation
def _fn_train(img, target):
   # 1. Randomly crop a [height, width] section of the image.
   img = tl.prepro.crop(img, 24, 24, False)
   # 2. Randomly flip the image horizontally.
   img = tl.prepro.flip_axis(img, is_random=True)
   # 3. Subtract off the mean and divide by the variance of the
       pixels.
   img = tl.prepro.samplewise_norm(img)
   target = np.reshape(target, ())
   return img, target

# loading the training set
train_ds = tl.data.CIFAR10(train_or_test='train', shape=(-1, 32,
    32, 3))
# feed the dataset into a dataloader, which integrates data
    augmentation
train_dl = tl.data.Dataloader(train_ds, transforms=[_fn_train],
    shuffle=True, batch_size=batch_size,
    output_types=(np.float32, np.int32))

# loading the testing set
test_ds = tl.data.CIFAR10(train_or_test='test', shape=(-1, 32,
    32, 3))
# feed the dataset into a dataloader
test_dl = tl.data.Dataloader(test_ds, batch_size=batch_size)

# the images can be accessed by iteration
for X_batch, y_batch in train_dl:
   # code to train/test a model
```

In this example, a CNN model with batch normalization (Ioffe and Szegedy 2015) is trained to classify the images from CIFAR-10. The model has two convolution blocks, each of which contains a batch normalization layer, and the blocks are followed by three dense layers.[5]

```
# a static CNN model with BatchNorm
def get_model_batchnorm(inputs_shape):
   # customized initialization
   W_init = tl.initializers.truncated_normal(stddev=5e-2)
   W_init2 = tl.initializers.truncated_normal(stddev=0.04)
   b_init2 = tl.initializers.constant(value=0.1)
```

[5]The full source code of the CNN example is available at https://github.com/tensorlayer/tensorlayer/tree/master/examples/basic_tutorials.

```
# start from a input layer
ni = Input(inputs_shape)

# the first convolution block with a Conv2d, a BatchNorm and
    a MaxPool.
nn = Conv2d(64, (5, 5), (1, 1), padding='SAME',
    W_init=W_init, b_init=None)(ni)
nn = BatchNorm2d(decay=0.99, act=tf.nn.relu)(nn)
nn = MaxPool2d((3, 3), (2, 2), padding='SAME')(nn)

# the second convolution block with a Conv2d, a BatchNorm and
    a MaxPool.
nn = Conv2d(64, (5, 5), (1, 1), padding='SAME',
    W_init=W_init, b_init=None)(nn)
nn = BatchNorm2d(decay=0.99, act=tf.nn.relu)(nn)
nn = MaxPool2d((3, 3), (2, 2), padding='SAME')(nn)

# the outputs of the convolution blocks are finally fed into
    three Dense layers
nn = Flatten()(nn) # reshape the tensor
nn = Dense(384, act=tf.nn.relu, W_init=W_init2,
    b_init=b_init2)(nn)
nn = Dense(192, act=tf.nn.relu, W_init=W_init2,
    b_init=b_init2)(nn)
nn = Dense(10, act=None, W_init=W_init2)(nn)

# create the model given the inputs and outputs
M = Model(inputs=ni, outputs=nn, name='cnn')
return M
```

1.10.6 RNN and Seq2seq: Chatbot

Chatbots are designed to conduct conversation by audio and text in general. In this example, we simplify the chatbot which takes text as inputs and responses in text. In this sense, the seq2seq by (Sutskever et al. 2014) can be a good fit for the chatbot. The seq2seq model has a sequence input and a sequence output. For example, both the input and output can be a sentence, which is a sequence of words. In chatbot, the seq2seq model takes a sentence as input and is trained to respond properly with another sentence. The seq2seq was originally proposed for machine translation but has potentials on many other sequence-to-sequence scenarios such as traffic prediction (Liao et al. 2018b) and text summarization (Liu et al. 2018). In practice, the seq2seq model consists of two RNNs: one encoder and one decoder. The encoder RNN learns the representation of the input sequence and the decoder RNN generates the response against the input. TensorLayer provides APIs to build a seq2seq model with one line of code.

```
# Seq2seq model
model_ = Seq2seq(
   decoder_seq_length=decoder_seq_length, # the upper limit of
       the sequence length in the decoding
   cell_enc=tf.keras.layers.GRUCell, # the cell for the encoder
       (RNN)
   cell_dec=tf.keras.layers.GRUCell, # the cell for the decoder
       (RNN)
   n_layer=3, # number of RNN layers for the encoder and decoder
   n_units=256, # number of hidden units in RNN layers
   embedding_layer=tl.layers.Embedding(vocabulary_size=vocabulary
       _size, embedding_size=emb_dim), # the embedding layer of
       the encoder
)
```

An example output of the seq2seq based chatbot model[6] is demonstrated below. The model ingests the input query which is a sentence and outputs several candidate responses.

```
Query > happy birthday have a nice day
 > thank you so much
 > thank babe
 > thank bro
 > thanks so much
 > thank babe i appreciate it
```

References

Abadi M, Barham P, Chen J, Davis A, Dean J, Devin M, Geoffrey S, Irving G, Devin M, Kudlur M, Manjunath J, Monga R, Moore S, Murray DG, Derek B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng X (2016) TensorFlow: a system for large-scale machine learning. In: USENIX Symposium on Operating Systems Design and Implementation. USENIX Association, Berkeley

Bahdanau D, Cho K, Bengio Y (2015) Neural machine translation by jointly learning to align and translate. In: Proceedings of the international conference on learning representations (ICLR)

Bishop CM (2006) Pattern recognition and machine learning. Springer, Berlin

Bottou L, Bousquet O (2007) The tradeoffs of large scale learning. In: Proceedings of the 20th international conference on neural information processing systems. Advances in neural information processing systems, vol 20, pp 161–168

Cao Z, Simon Z, Wei SE, Sheikh SE (2017) Realtime multi-person 2D pose estimation using part affinity fields. In: Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)

Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using RNN encoder-decoder for statistical machine translation.

[6]The full source code of chatbot is available at https://github.com/tensorlayer/seq2seq-chatbot.

In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)

Chung J, Gulcehre C, Cho K, Bengio Y (2014) Empirical evaluation of gated recurrent neural networks on sequence modeling. Preprint. arXiv:14123555

Devlin J, Chang MW, Lee K, Toutanova K (2019) BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, vol 1 (long and short papers). Association for Computational Linguistics, Minneapolis, pp 4171–4186. https://doi.org/10.18653/v1/N19-1423

Dong H, Supratak A, Mai L, Liu F, Oehmichen A, Yu S, Guo Y (2017a) TensorLayer: a versatile library for efficient deep learning development. In: Proceedings of the ACM Multimedia (MM). http://tensorlayer.org

Dong H, Zhang J, McIlwraith D, Guo Y (2017b) I2t2i: learning text to image synthesis with textual data augmentation. In: Proceedings of the IEEE international conference on image processing (ICIP)

Duchi J, Hazan E, Singer Y (2011) Adaptive subgradient methods for online learning and stochastic optimization. J Mach Learn Res 12:2121–2159

Gal Y, Ghahramani Z (2016) Dropout as a Bayesian approximation: representing model uncertainty in deep learning. In: Proceedings of the international conference on machine learning (ICML), pp 1050–1059

Glorot X, Bordes A, Bengio Y (2011) Deep sparse rectifier neural networks. In: Proceedings of the international conference on artificial intelligence and statistics (AISTATS), pp 315–323

Goodfellow I, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, Courville A, Bengio Y (2014) Generative adversarial nets. In: Proceedings of the neural information processing systems conference. Advances in neural information processing systems

Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT Press, Cambridge. http://www.deeplearningbook.org

Hara K, Saitoh D, Shouno H (2016) Analysis of dropout learning regarded as ensemble learning. In: Proceedings of the international conference on artificial neural networks (ICANN). Springer, Berlin, pp 72–79

He K, Zhang X, Ren S, Sun J (2015) Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. In: Proceedings of the IEEE international conference on computer vision, pp 1026–1034

He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)

Hinton GE, Srivastava N, Krizhevsky A, Sutskever I, Salakhutdinov RR (2012) Improving neural networks by preventing co-adaptation of feature detectors. Preprint. arXiv:12070580

Hochreiter S, Hochreiter S, Schmidhuber J, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780

Hornik K, Stinchcombe M, White H (1989) Multilayer feedforward networks are universal approximators. Neural Netw 2(5):359–366

Howard AG, Zhu M, Chen B, Kalenichenko D, Wang W, Weyand T, Andreetto M, Adam H (2017) MobileNets: efficient convolutional neural networks for mobile vision applications. Preprint. arXiv:170404861

Ioffe S, Szegedy C (2015) Batch normalization: accelerating deep network training by reducing internal covariate shift. Preprint. arXiv:150203167

James S, Wohlhart P, Kalakrishnan M, Kalashnikov D, Irpan A, Ibarz J, Levine S, Hadsell R, Bousmalis K (2019) Sim-to-real via sim-to-sim: data-efficient robotic grasping via randomized-to-canonical adaptation networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 12627–12637

Jozefowicz R, Zaremba W, Sutskever I (2015) An empirical exploration of recurrent network architectures. In: International conference on machine learning, pp 2342–2350

Kingma D, Ba J (2014) Adam: a method for stochastic optimization. In: Proceedings of the international conference on learning representations (ICLR)

Ko T, Peddinti V, Povey D, Khudanpur S (2015) Audio augmentation for speech recognition. In: Annual conference of the international speech communication association

Krizhevsky A, Hinton G et al (2009) Learning multiple layers of features from tiny images. Technical Report. Citeseer

Krizhevsky A, Sutskever I, Hinton GE (2012) ImageNet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp 1097–1105

LeCun Y, Boser B, Denker JS, Henderson D, Howard RE, Hubbard W, Jackel LD (1989) Backpropagation applied to handwritten zip code recognition. Neural Comput 1(4):541–551

LeCun Y, Bottou L, Bengio Y, Haffner P et al (1998) Gradient-based learning applied to document recognition. Proc IEEE 86(11):2278–2324

LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521(7553):436

Ledig C, Theis L, Huszar F, Caballero J, Cunningham A, Acosta A, Aitken A, Tejani A, Totz J, Wang Z, Shi W (2017) Photo-realistic single image super-resolution using a generative adversarial network. In: Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)

Lee JY, Dernoncourt F (2016) Sequential short-text classification with recurrent and convolutional neural networks. In: Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies. Association for Computational Linguistics, San Diego, pp 515–520. https://doi.org/10.18653/v1/N16-1062

Liao B, Zhang J, Cai M, Tang S, Gao Y, Wu C, Yang S, Zhu W, Guo Y, Wu F (2018a) Dest-ResNet: a deep spatiotemporal residual network for hotspot traffic speed prediction. In: 2018 ACM multimedia conference on multimedia conference. ACM, New York, pp 1883–1891

Liao B, Zhang J, Wu C, McIlwraith D, Chen T, Yang S, Guo Y, Wu F (2018b) Deep sequence learning with auxiliary information for traffic prediction. In: Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining. ACM, New York, pp 537–546

Liu PJ, Saleh M, Pot E, Goodrich B, Sepassi R, Kaiser L, Shazeer N (2018) Generating wikipedia by summarizing long sequences. In: International conference on learning representations. https://openreview.net/forum?id=Hyg0vbWC-

Luong T, Pham H, Manning CD (2015) Effective approaches to attention-based neural machine translation. In: Proceedings of the 2015 conference on empirical methods in natural language processing. Association for Computational Linguistics, Lisbon, pp 1412–1421. https://doi.org/10.18653/v1/D15-1166

Maas AL, Daly RE, Pham PT, Huang D, Ng AY, Potts C (2011) Learning word vectors for sentiment analysis. In: Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies, HLT ’11, vol 1. Association for Computational Linguistics, Stroudsburg, pp 142–150. http://dl.acm.org/citation.cfm?id=2002472.2002491

Mikolov T, Karafiát M, Burget L, Cernocky J, Khudanpur S (2010) Recurrent neural network based language model. In: INTERSPEECH 2010, 11th annual conference of the international speech communication association, Makuhari

Nallapati R, Zhai F, Zhou B (2017) SummaRuNNer: a recurrent neural network based sequence model for extractive summarization of documents. In: Proceedings of the thirty-first AAAI conference on artificial intelligence, AAAI’17. AAAI Press, Palo Alto, pp 3075–3081

Ng AY, Jordan MI (2002) On discriminative vs. generative classifiers: a comparison of logistic regression and naive Bayes. In: Proceedings of the neural information processing systems. Advances in neural information processing systems. Conference, pp 841–848

Noh H, Hong S, Han B (2015) Learning deconvolution network for semantic segmentation. In: Proceedings of the international conference on computer vision (ICCV), pp 1520–1528

Olah C (2015) Understanding LSTM networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Peng XB, Andrychowicz M, Zaremba W, Abbeel P (2018) Sim-to-real transfer of robotic control with dynamics randomization. In: 2018 IEEE international conference on robotics and automation (ICRA). IEEE, Piscataway, pp 1–8

Reed S, Akata Z, Yan X, Logeswaran L, Schiele B, Lee H (2016) Generative adversarial text to image synthesis. In: Proceedings of the international conference on machine learning (ICML)

Rish I et al (2001) An empirical study of the naive Bayes classifier. In: International joint conference on artificial intelligence 2001 workshop on empirical methods in artificial intelligence. vol 3, pp 41–46

Rosenblatt F (1958) The perceptron: a probabilistic model for information storage and organization in the brain. Psychol Rev 65(6):386

Ruck DW, Rogers SK, Kabrisky M, Oxley ME, Suter BW (1990) The multilayer perceptron as an approximation to a Bayes optimal discriminant function. IEEE Trans Neural Netw 1(4):296–298

Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by back-propagating errors. Nature 323(6088):533

Rusu AA, Rabinowitz NC, Desjardins G, Soyer H, Kirkpatrick J, Kavukcuoglu K, Pascanu R, Hadsell R (2016) Progressive neural networks. Preprint. arXiv:160604671

Samuel A (1959) Some studies in machine learning using the game of checkers. IBM J Res Dev 3:210–219

Simonyan K, Zisserman A (2015) Very deep convolutional networks for large-scale image recognition. In: Proceedings of the international conference on learning representations (ICLR)

Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R (2014) Dropout: a simple way to prevent neural networks from overfitting. J Mach Learn Res 15(1):1929–1958

Sutskever I, Vinyals O, Le QV (2014) Sequence to sequence learning with neural networks. In: Proceedings of the neural information processing systems. Advances in neural information processing systems. Conference, pp 3104–3112

Tieleman T, Hinton G (2017) Divide the gradient by a running average of its recent magnitude. COURSERA: neural networks for machine learning. Technical Report

van den Oord A, Dieleman S, Zen H, Simonyan K, Vinyals O, Graves A, Kalchbrenner N, Senior A, Kavukcuoglu K (2016) WaveNet: a generative model for raw audio. In: Arxiv. https://arxiv.org/abs/1609.03499

Wierstra D, Förster A, Peters J, Schmidhuber J (2010) Recurrent policy gradients. Log J IGPL 18(5):620–634

Xu B, Wang N, Chen T, Li M (2015) Empirical evaluation of rectified activations in convolutional network. In: Proceedings of the international conference on machine learning (ICML) workshop

Yang G, Yu S, Dong H, Slaubaugh, GG, Dragotti PL, Ye X, Liu F, Arridge SR, Keegan J, Guo Y, Firmin DN (2018) DAGAN: deep de-aliasing generative adversarial networks for fast compressed sensing MRI reconstruction. IEEE Trans Med Imaging 37(6):1310–1321

Yang Z, Dai Z, Yang Y, Carbonell J, Salakhutdinov RR, Le QV (2019) XLNet: generalized autoregressive pretraining for language understanding. In: Advances in neural information processing systems, pp 5754–5764

Yin W, Kann K, Yu M, Schütze H (2017) Comparative study of CNN and RNN for natural language processing. Preprint. arXiv:170201923

Zhang X, Zhao J, LeCun Y (2015) Character-level convolutional networks for text classification. In: Advances in neural information processing systems, pp 649–657

Zhang J, Lertvittayakumjorn P, Guo Y (2019a) Integrating semantic knowledge to tackle zero-shot text classification. In: Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, vol 1 (long and short papers). Association for Computational Linguistics, Minneapolis, pp 1031–1040. https://doi.org/10.18653/v1/N19-1108

Zhang J, Zhao Y, Saleh M, Liu PJ (2019b) PEGASUS: Pre-training with extracted gap-sentences for abstractive summarization. Preprint. arXiv:191208777

Zoph B, Le QV (2016) Neural architecture search with reinforcement learning. Preprint. arXiv:161101578

Zoph B, Vasudevan V, Shlens J, Le QV (2018) Learning transferable architectures for scalable image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 8697–8710