Chapter 7 Backward Propagation Algorithm

> The longer you look back, the farther you can look forward. - Winston S. Churchill

In Chapter 6, we have systematically introduced the basic neural network algorithm: starting from the representation of inputs and outputs, introducing the perceptron model, multi-input and multi-output fully connected layers, and then expanding to multi-layer neural networks. We also introduced the design of the output layer under different scenarios and the commonly used loss functions and their implementation.

In this chapter, we will learn one of the core algorithms in the neural network from the theoretical level: error back propagation (BP). In fact, the back propagation algorithm has been proposed in the early 1960s, but it has not attracted the attention of the industry. In 1970, Seppo Linnainmaa proposed an automatic chain derivation method in his master thesis, and implemented the back propagation algorithm on the computer. In 1974, Paul Werbos first proposed the possibility of applying the back propagation algorithm to neural networks in his doctoral thesis, but unfortunately, Paul Werbos did not publish subsequent related research. In fact, Paul Werbos believes that this research idea is meaningful for solving perceptron problems, but due to the cold winter of artificial intelligence, the community has generally lost its belief in solving those problems. Until about 10 years later in 1986, Geoffrey Hinton et al. applied the back propagation algorithm on the neural network [1], making the back propagation algorithm vigorous in the neural network community.

With the functions of automatic derivation and automatic parameter updating of deep learning frameworks, algorithm designers can build complex models and networks with little need for in-depth knowledge of back propagation algorithms, and can easily train network models by calling optimization tools. However, the back propagation algorithm and gradient descent algorithm are the core of the neural network, and it is very important to deeply understand its principle. We first review the mathematical concepts such as derivatives and gradients, and then derive the gradient forms of commonly used activation and loss functions, and begin to gradually derive the gradient propagation methods of perceptron and multilayer neural networks.

7.1 Derivatives and gradients

In high school, we came into contact with the concept of derivative, which is defined as the limit of the ratio of the increment Δy of the function output value to the increment Δx of the independent variable x when the independent variable x produces a slight disturbance Δx as Δx approaches to zero:

$$a = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

The derivative of the function $f(x)$ can be written as $f'(x)$ or $\frac{\mathrm{d}y}{\mathrm{d}x}$. From a geometric point of view, the derivative of a univariate function is the slope of the tangent of the function here, that is, the rate of change of the function value along the direction of x. Consider an example in physics, e.g., the expression of the displacement function of free-fall motion $y = \frac{1}{2}gt^2$. The derivative with respect to time is $\frac{\mathrm{d}y}{\mathrm{d}t} = \frac{\mathrm{d}\frac{1}{2}gt^2}{\mathrm{d}t} = gt$. Considering that velocity v is defined as the rate of change of displacement, so the derivative of displacement with respect to time is velocity, i.e. $v = gt$.

In fact, the derivative is a very broad concept. Because most of the functions we have encountered before are univariate functions, and the independent variable has only two directions: x^+ and x^-. When the number of independent variables of a function is greater than one, the concept of the derivative of the function is extended to the rate of change of the function value in any direction. The derivative itself is a scalar and has no direction, but the derivative characterizes the rate of change of the function value in a certain direction. Among these arbitrary directions, several directions along the coordinate axis are relatively special, which is also called partial derivative. For univariate functions, the derivative is written as $\frac{\mathrm{d}y}{\mathrm{d}x}$. For the partial derivative of the multivariate function, it is recorded as $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \cdots$. Partial derivatives are special cases of derivatives and have no direction.

Consider a neural network model that is essentially a multivariate function, such as a weight matrix $\boldsymbol{W}$ of shape $[784, 256]$, which contains a connection weight of 784×256, and we need to ask for a partial derivative of 784×256. It should be noted that in mathematical expression habits, the independent variables to be discussed are generally recorded as $\boldsymbol{x}$, but in neural networks, they are generally used to represent inputs, such as pictures, text, and voice data. The independent variables of the network are network parameter sets $\theta = \{w_1, b_1, w_2, b_2, \cdots\}$. When the gradient descent algorithm is used to optimize the network, all partial derivatives of the network need to be requested. Therefore, we are also concerned about the derivative of the error function $\mathcal{L}$ output along the direction of the independent variable θ_i, that is, $\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial b_1}, \cdots$. Write all partial derivatives of the function in vector form:

$$\nabla_\theta \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}, \frac{\partial \mathcal{L}}{\partial \theta_3}, \cdots, \frac{\partial \mathcal{L}}{\partial \theta_n}\right)$$

The gradient descent algorithm can be updated in the form of a vector:

$$\theta' = \theta - \eta \cdot \nabla_\theta \mathcal{L}$$

$\eta\ is$ learning rate. The gradient descent algorithm is generally to find the minimum value of the function $\mathcal{L}$, and sometimes it is also desirable to solve the maximum value of the function, which need to update the gradient in the following way:

$$\theta' = \theta + \eta \cdot \nabla_\theta \mathcal{L}$$

This update method is called the gradient ascent algorithm. The gradient descent algorithm

and the gradient ascent algorithm are the same in principle. One is to update in the opposite direction of the gradient, and the other is to update in the direction of the gradient. Both need to solve partial derivatives. Here, the vector $\left(\frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}, \frac{\partial \mathcal{L}}{\partial \theta_3}, \cdots, \frac{\partial \mathcal{L}}{\partial \theta_n}\right)$ is called the gradient of the function, which is composed of all partial derivatives and represents the direction. The direction of the gradient indicates the direction in which the function value rises fastest, and the reverse of the gradient indicates the direction in which the function value decreases fastest.

The gradient descent algorithm does not guarantee the global optimal solution, which is mainly caused by the non-convexity of the objective function. Consider the non-convex function in Figure 7.1. The dark blue area is the minimum area. Different optimization trajectories may obtain different optimal numerical solutions. These numerical solutions are not necessarily global optimal solutions.

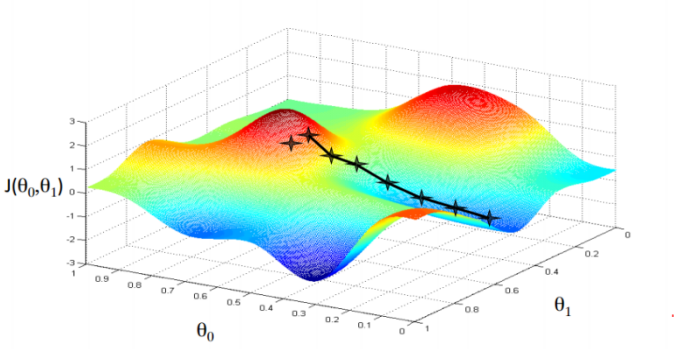

Figure 7.1 Non-convex function example

Neural network model expressions are usually very complex, and the model parameters can reach tens or hundreds of millions of levels. Almost all neural network optimization problems rely on deep learning frameworks to automatically calculate the gradient of network parameters, and then use gradient descent iteratively optimize the network parameters until the performance meets the demand. The main algorithms implemented in deep learning frameworks are back propagation and gradient descent algorithms. So understanding the principles of these two algorithms is helpful to understand the role of deep learning frameworks.

Before introducing the back propagation algorithm of the multilayer neural network, we first introduce the common attributes of the derivative, the gradient derivation of the common activation function and the loss function, and then derive the gradient propagation law of the multilayer neural network.

7.2 Common properties of derivatives

This section introduces the derivation rules and sample explanations of common functions, which paves the way for the derivation of neural network related functions.

7.2.1 Common derivatives

- The derivative of constant function c is 0. E.g. the derivative of $y = 2$ is $\frac{\mathrm{d}y}{\mathrm{d}x} = 0$.

- The derivative of linear function $y = ax + c$ is a. E.g. the derivative of $y = 2x + 1$ is $\frac{\mathrm{d}y}{\mathrm{d}x} = 2$.
- The derivative of function x^a is ax^{a-1}. E.g. the derivative of $y = x^2$ is $\frac{\mathrm{d}y}{\mathrm{d}x} = 2x$.
- The derivative of exponential function a^x is $a^x \ln a$. E.g. the derivative of $y = e^x$ is $\frac{\mathrm{d}y}{\mathrm{d}x} = e^x \ln e = e^x$
- The derivative of log function $\log_a x$ is $\frac{1}{x \ln a}$. E.g. the derivative of $y = \ln x$ is $\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{1}{x \ln e} = \frac{1}{x}$

7.2.2 Common property of derivatives

- $(f + g)' = f' + g'$
- $(fg)' = f' \cdot g + f \cdot g'$
- $\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}, \ g \neq 0$
- Consider function of function $f(g(x))$, let $u = g(x)$, the derivative is:

$$\frac{\mathrm{d}f(g(x))}{\mathrm{d}x} = \frac{\mathrm{d}f(u)}{\mathrm{d}u}\frac{\mathrm{d}g(x)}{\mathrm{d}x} = f'(u) \cdot g'(x)$$

7.2.3 Hands-on derivative finding

Considering objective function $\mathcal{L} = x \cdot w^2 + b^2$, its derivative is:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial x \cdot w^2}{\partial w} = x \cdot 2w$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial b^2}{\partial b} = 2b$$

Considering objective function $\mathcal{L} = x \cdot e^w + e^b$, its derivative is:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial x \cdot e^w}{\partial w} = x \cdot e^w$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial e^b}{\partial b} = e^b$$

Considering objective function $\mathcal{L} = [y - (xw + b)]^2 = [(xw + b) - y]^2$, let $g = xw + b - y$, and the derivative is:

$$\frac{\partial \mathcal{L}}{\partial w} = 2g \cdot \frac{\partial \mathrm{g}}{\partial w} = 2g \cdot x = 2(xw + b - y) \cdot x$$

$$\frac{\partial \mathcal{L}}{\partial b} = 2g \cdot \frac{\partial \mathrm{g}}{\partial b} = 2g \cdot 1 = 2(xw + b - y)$$

Considering objective function $\mathcal{L} = a\ln(xw + b)$, let $g = xw + b$, and the derivative is:

$$\frac{\partial \mathcal{L}}{\partial w} = a \cdot \frac{1}{g} \cdot \frac{\partial g}{\partial w} = \frac{a}{xw + b} \cdot x$$

$$\frac{\partial \mathcal{L}}{\partial b} = a \cdot \frac{1}{g} \cdot \frac{\partial g}{\partial b} = \frac{a}{xw + b}$$

7.3 Derivative of activation function

Here we introduce the derivation of the activation function commonly used in neural networks.

7.3.1 Derivative of Sigmoid function

The expression of Sigmoid function is:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Let's derive the derivative expression of the Sigmoid function:

$$\frac{\mathrm{d}}{\mathrm{d}x}\sigma(x) = \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{1}{1 + e^{-x}}\right)$$

$$= \frac{\mathrm{e}^{-x}}{(1 + \mathrm{e}^{-x})^2}$$

$$= \frac{(1 + \mathrm{e}^{-x}) - 1}{(1 + \mathrm{e}^{-x})^2}$$

$$= \frac{1 + e^{-x}}{(1 + e^{-x})^2} - \left(\frac{1}{1 + e^{-x}}\right)^2$$

$$= \sigma(x) - \sigma(x)^2$$

$$= \sigma(1 - \sigma)$$

It can be seen that the derivative expression of the Sigmoid function can finally be expressed as a simple operation of the output value of the activation function. Using this property, we can calculate its derivate by caching the output value of the Sigmoid function of each layer in the gradient calculation of the neural network. The derivative function of the Sigmoid function is shown in Figure 7.2.

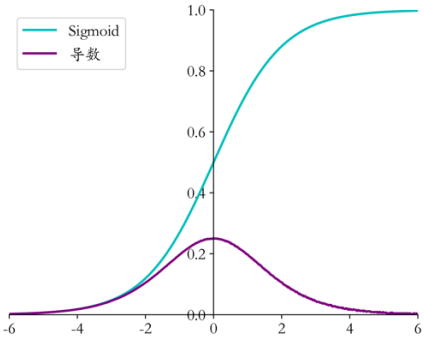

Figure 7.2 Sigmoid function and its derivative

In order to help understand the implementation details of the backpropagation algorithm, this chapter chooses not to use TensorFlow's automatic derivation function. This chapter uses Numpy to implement a multi-layer neural network optimized by backpropagation algorithm. Here the derivative of the Sigmoid function is implemented by Numpy:

```
import numpy as np # import numpy library
def sigmoid(x): # implement sigmoid function
    return 1 / (1 + np.exp(-x))

def derivative(x):  # calculate derivative of sigmoid
    # Using the derived expression of the derivatives
    return sigmoid(x)*(1-sigmoid(x))
```

7.3.2 Derivative of ReLU function

Recall the expression of the ReLU function:

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

The derivation of its derivative is very simple:

$$\frac{\mathrm{d}}{\mathrm{d}x}\mathrm{ReLU} = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

It can be seen that the derivative calculation of the ReLU function is simple. When x is greater than or equal to zero, the derivative value is always 1. In the process of back propagation, it will neither amplify the gradient, causing gradient exploding, nor shrink the gradient, causing gradient vanishing phenomenon. The derivative curve of the ReLU function is shown in Figure 7.3.

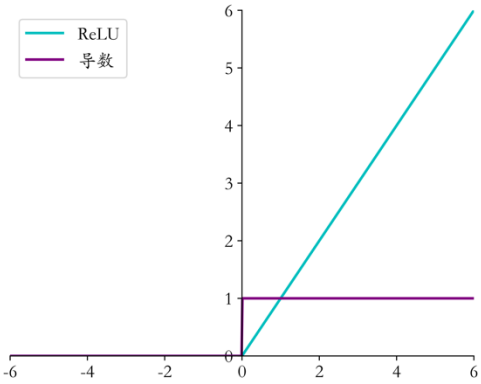

Figure 7.3 ReLU function and its derivative

Before the ReLU function was widely used, the activation function in neural networks was mostly Sigmoid. However, the Sigmoid function was prone to gradient dispersion. When the number of layers of the network increased, because the gradient values become very small, the parameters of the network cannot be effectively updated. As a result, deeper neural networks cannot be trained, resulting in the research of neural networks staying at the shallow level. With the introduction of the ReLU function, the phenomenon of gradient dispersion is well alleviated, and the number of layers of the neural network can reach deeper layers. For example, the ReLU activation function is used in AlexNet, and the number of layers reaches 8. Some convolutional neural networks with over 100 layers also mostly uses the ReLU activation function.

Through Numpy, we can easily achieve the derivative of the ReLU function, the code is as follows:

```
def derivative(x):  # Derivative of ReLU
    d = np.array(x, copy=True)
    d[x < 0] = 0
    d[x >= 0] = 1
    return d
```

7.3.3 Derivative of LeakyReLU function

Recall the expression of LeakyReLU function:

$$\text{LeakyReLU} = \begin{cases} x & x \geq 0 \\ px & x < 0 \end{cases}$$

Its derivative can be derived as:

$$\frac{\mathrm{d}}{\mathrm{d}x}\text{LeakyReLU} = \begin{cases} 1 & x \geq 0 \\ p & x < 0 \end{cases}$$

It's different from the ReLU function because when x is less than zero, the derivative value of the LeakyReLU function is not 0, but a constant p, which is generally set to a smaller value, such as 0.01 or 0.02. The derivative curve of the LeakyReLU function is shown in Figure 7.4.

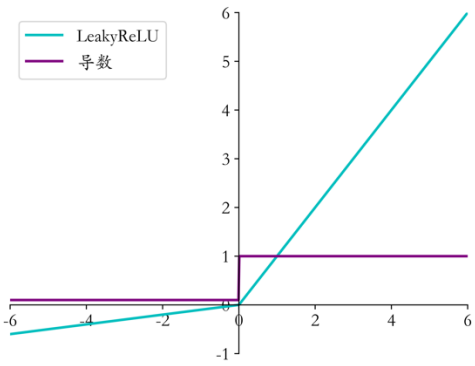

Figure 7.4 LeakyReLU function and its derivative

The LeakyReLU function effectively overcomes the defects of the ReLU function and is also widely used. We can implement the derivative of LeakyReLU function through Numpy as follows:

```
def derivative(x, p): # p is the slope of negative part of LeakyReLU
    dx = np.ones_like(x)  # initialize a vector with 1
    dx[x < 0] = p  # set negative part to p
    return dx
```

7.3.4 Derivative of Tanh function

Recall the expression of the tanh function:

$$\tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

$$= 2 \cdot \text{sigmoid}(2x) - 1$$

Its derivative expression is:

$$\frac{\mathrm{d}}{\mathrm{d}x}\tanh(x) = \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2}$$

$$= 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - \tanh^2(x)$$

The tanh function and its derivative curve are shown in Figure 7.5.

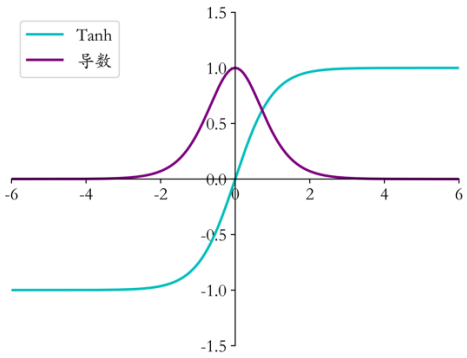

Figure 7.5 Tanh function and its derivative

In Numpy, the derivative of the Tanh function is implemented through the Sigmoid function as follows:

```
def sigmoid(x):  # sigmoid function
    return 1 / (1 + np.exp(-x))

def tanh(x):  # tanh function
    return 2*sigmoid(2*x) - 1

def derivative(x):  # derivative of tanh
    return 1-tanh(x)**2
```

7.4 Gradient of loss function

The common loss functions have been introduced previously. Here we mainly derive the gradient expressions of the mean square error loss function and the cross-entropy loss function.

7.4.1 Gradient of mean square error function

The mean square error loss function expression is:

$$\mathcal{L} = \frac{1}{2}\sum_{k=1}^{K}(y_k - o_k)^2$$

The terms $\frac{1}{2}$ in the above formula are used to simplify the calculation, and $\frac{1}{K}$ can also be used for averaging instead. None of these scaling operations will change the gradient direction. Then its partial derivative $\frac{\partial \mathcal{L}}{\partial o_i}$ can be expanded to:

$$\frac{\partial \mathcal{L}}{\partial o_i} = \frac{1}{2}\sum_{k=1}^{K}\frac{\partial}{\partial o_i}(y_k - o_k)^2$$

Decomposition by the law of derivative of composite function:

$$\frac{\partial \mathcal{L}}{\partial o_i} = \frac{1}{2}\sum_{k=1}^{K} 2 \cdot (y_k - o_k) \cdot \frac{\partial (y_k - o_k)}{\partial o_i}$$

i.e.

$$\frac{\partial \mathcal{L}}{\partial o_i} = \sum_{k=1}^{K} (y_k - o_k) \cdot -1 \cdot \frac{\partial o_k}{\partial o_i}$$

$$= \sum_{k=1}^{K} (o_k - y_k) \cdot \frac{\partial o_k}{\partial o_i}$$

Considering that $\frac{\partial o_k}{\partial o_i}$ is 1 when $k = i$ and $\frac{\partial o_k}{\partial o_i}$ is 0 for other cases, that is, the partial derivative $\frac{\partial \mathcal{L}}{\partial o_i}$ is only related to the ith node, so the summation symbol in the above formula can be removed. The derivative of the mean square error function can be expressed as:

$$\frac{\partial \mathcal{L}}{\partial o_i} = (o_i - y_i)$$

7.4.2 Gradient of cross entropy function

When calculating the cross-entropy loss function, the Softmax function and the cross-entropy function are generally implemented in a unified manner. We first derive the gradient of the Softmax function, and then derive the gradient of the cross-entropy function.

Gradient of Softmax Recall of the expression of Softmax：

$$p_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

Its function is to convert the values of the output nodes into probabilities and ensure that the sum of probabilities is 1, as shown in Figure 7.6.

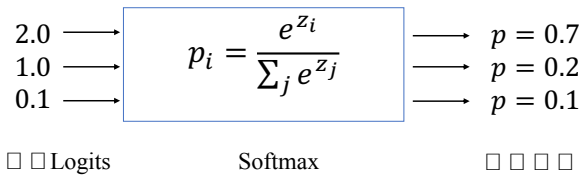

Figure 7.6 Softmax illustration

Recall

$$f(x) = \frac{g(x)}{h(x)}$$

The derivative of the function is：

$$f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h(x)^2}$$

For Softmax function，$g(x) = e^{z_i}$，$h(x) = \sum_{k=1}^{K} e^{z_k}$. We'll derive its gradient at two conditions: $i = j$ and $i \neq j$.

- $i = j$. The derivative of Softmax $\frac{\partial p_i}{\partial z_j}$ is:

$$\frac{\partial p_i}{\partial z_j} = \frac{\partial \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}}{\partial z_j} = \frac{e^{z_i} \sum_{k=1}^{K} e^{z_k} - e^{z_j} e^{z_i}}{(\sum_{k=1}^{K} e^{z_k})^2}$$

$$= \frac{e^{z_i}(\sum_{k=1}^{K} e^{z_k} - e^{z_j})}{(\sum_{k=1}^{K} e^{z_k})^2}$$

$$= \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \times \frac{(\sum_{k=1}^{K} e^{z_k} - e^{z_j})}{\sum_{k=1}^{K} e^{z_k}}$$

The above expression is the multiplication of p_i and $1 - p_j$，and $p_i = p_j$. So when $i = j$，the derivative of Softmax $\frac{\partial p_i}{\partial z_j}$ is：

$$\frac{\partial p_i}{\partial z_j} = p_i(1 - p_j), i = j$$

- $i \neq j$. Extend the Softmax function

$$\frac{\partial p_i}{\partial z_j} = \frac{\partial \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}}{\partial z_j} = \frac{0 - e^{z_j} e^{z_i}}{(\sum_{k=1}^{K} e^{z_k})^2}$$

$$= \frac{-e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \times \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

i.e.

$$\frac{\partial p_i}{\partial z_j} = -p_j \cdot p_i$$

It can be seen that although the gradient derivation process of the Softmax function is slightly complicated, the final expression is still very concise. The partial derivative expression is as follows:

$$\frac{\partial p_i}{\partial z_j} = \begin{cases} p_i(1 - p_j) & \text{when } i = j \\ -p_i \cdot p_j & \text{when } i \neq j \end{cases}$$

Gradient of cross-entropy function Consider the expression of the cross-entropy loss function:

$$\mathcal{L} = -\sum_{k} y_k \log(p_k)$$

Here we directly derive the partial derivative of the final loss value $\mathcal{L}$ to the logits variable z_i of the network output, which expands to

$$\frac{\partial \mathcal{L}}{\partial z_i} = -\sum_k y_k \frac{\partial \log(p_k)}{\partial z_i}$$

Decompose the composite function $\log h$ into:

$$= -\sum_k y_k \frac{\partial \log(p_k)}{\partial p_k} \cdot \frac{\partial p_k}{\partial z_i}$$

i.e.

$$= -\sum_k y_k \frac{1}{p_k} \cdot \frac{\partial p_k}{\partial z_i}$$

where $\frac{\partial p_k}{\partial z_i}$ is the partial derivative of the Softmax function that we have derived.

Split the summation symbol into the two cases: $k = i$ and $k \neq i$, and substitute the expression of $\frac{\partial p_k}{\partial z_i}$, we can get

$$\frac{\partial \mathcal{L}}{\partial z_i} = -y_i(1 - p_i) - \sum_{k \neq i} y_k \frac{1}{p_k}(-p_k \cdot p_i)$$

$$= -y_i(1 - p_i) + \sum_{k \neq i} y_k \cdot p_i$$

$$= -y_i + y_i p_i + \sum_{k \neq i} y_k \cdot p_i$$

i.e.

$$\frac{\partial \mathcal{L}}{\partial z_i} = p_i \left(y_i + \sum_{k \neq i} y_k \right) - y_i$$

In particular, the one-hot encoding method for the label in the classification problem has the following relationship:

$$\sum_k y_k = 1$$

$$y_i + \sum_{k \neq i} y_k = 1$$

Therefore, the partial derivative of cross entropy can be further simplified to:

$$\frac{\partial \mathcal{L}}{\partial z_i} = p_i - y_i$$

7.5 Gradient of fully connected layer

After introducing the basic knowledge of gradients, we formally entered the derivation of the neural network's back propagation algorithm. The structure of the neural network is diverse, and it is impossible to analyze the gradient expressions one by one. We will use a neural network with a fully connected layer network, a sigmoid function as the activation function, and a softmax + MSE loss function as the error function to derive the gradient propagation law.

7.5.1 Gradient of a single neuron

For a neuron model using Sigmoid activation function, its mathematical model can be written as:

$$o^{(1)} = \sigma(\boldsymbol{w}^{(1)^{\mathrm{T}}}\boldsymbol{x} + b^{(1)})$$

The superscript of the variable represents the number of layers. For example, $o^{(1)}$ represents the output of the first layer and $\boldsymbol{x}$ is the input of the network. We take the partial derivative derivation $\frac{\partial \mathcal{L}}{\partial w_{j1}}$ of the weight parameter w_{j1} as an example. For the convenience of demonstration, we draw the neuron model as shown in Figure 7.7. The bias b is not shown in the figure, and the number of input nodes is J. The weight connection from the input of the jth node to the output $o^{(1)}$ is denoted as $w_{j1}^{(1)}$, where the superscript indicates the number of layers to which the weight parameter belongs, and the subscript indicates the starting node number and the ending node number of the current connection. For example, the subscript $j1$ indicates the jth node of the previous layer to the 1st node of the current layer. The variable before the activation function σ is called $z_1^{(1)}$, and the variable after the activation function σ is called $o_1^{(1)}$. Because there is only one output node, so $o_1^{(1)} = o^{(1)} = o$. The error value $\mathcal{L}$ is calculated by the error function between the output and the real label.

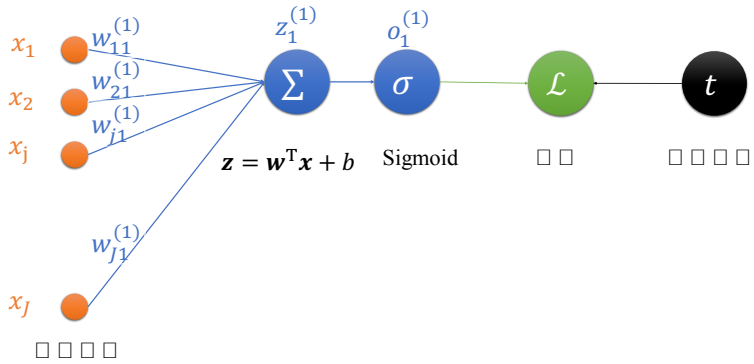

Figure 7.7 Neuron model

If we use the mean square error function, considering that a single neuron has only one output $o_1^{(1)}$, then the loss can be expressed as:

$$\mathcal{L} = \frac{1}{2}\left(o_1^{(1)} - t\right)^2 = \frac{1}{2}(o_1 - t)^2$$

Among them, t is the real label value. Adding $\frac{1}{2}$ does not affect the direction of the gradient, and the calculation is simpler. We take the weight variable w_{j1} of the jth ($j \in [1, J]$) node as an example, and consider the partial derivative $\frac{\partial \mathcal{L}}{\partial w_{j1}}$ of the loss function $\mathcal{L}$:

$$\frac{\partial \mathcal{L}}{\partial w_{j1}} = (o_1 - t)\frac{\partial o_1}{\partial w_{j1}}$$

Considering $o_1 = \sigma(z_1)$ and the derivative of the Sigmoid function is $\sigma' = \sigma(1 - \sigma)$, we

have：

$$\frac{\partial \mathcal{L}}{\partial w_{j1}} = (o_1 - t)\frac{\partial \sigma(z_1)}{\partial w_{j1}}$$

$$= (o_1 - t)\sigma(z_1)\big(1 - \sigma(z_1)\big)\frac{\partial z_1^{(1)}}{\partial w_{j1}}$$

Write $\sigma(z_1)$ as o_1：

$$\frac{\partial \mathcal{L}}{\partial w_{j1}} = (o_1 - t)o_1(1 - o_1)\frac{\partial z_1^{(1)}}{\partial w_{j1}}$$

Consider $\frac{\partial z_1^{(1)}}{\partial w_{j1}} = x_j$， we have：

$$\frac{\partial \mathcal{L}}{\partial w_{j1}} = (o_1 - t)o_1(1 - o_1)x_j$$

It can be seen from the above formula that the partial derivative of the error to the weight w_{j1} is only related to the output value o_1, the true value t, and the input x_j connected to the current weight.

7.5.2 Gradient of fully connected layer

We generalize the single neuron model to a single-layer network of fully connected layers, as shown in Figure 7.8. The input layer obtains the output vector $\boldsymbol{o}^{(1)}$ through a fully connected layer, and calculates the mean square error with the real label vector $\boldsymbol{t}$. The number of input nodes is J, and the number of output nodes is K.

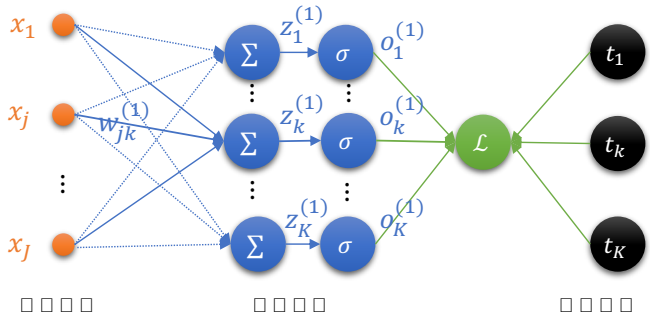

Figure 7.8 Fully connected layer

The multi-output fully connected network layer model differs from the single neuron model in that it has many more output nodes $o_1^{(1)}, o_2^{(1)}, o_3^{(1)}, \cdots, o_K^{(1)}$, and each output node corresponds to a real label $t_1, t_2, \ldots, t_K$. w_{jk} is the connection weight of the jth input node and the kth output node. The mean square error can be expressed as:

$$\mathcal{L} = \frac{1}{2}\sum_{i=1}^{K}\left(o_i^{(1)} - t_i\right)^2$$

Since $\frac{\partial \mathcal{L}}{\partial w_{jk}}$ is only associated with node $o_k^{(1)}$, the summation symbol in the above formula can be removed, i.e. $i = k$:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}} = (o_k - t_k)\frac{\partial o_k}{\partial w_{jk}}$$

Substitute $o_k = \sigma(z_k)$:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}} = (o_k - t_k)\frac{\partial \sigma(z_k)}{\partial w_{jk}}$$

Consider the derivative of the Sigmoid function $\sigma' = \sigma(1 - \sigma)$:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}} = (o_k - t_k)\sigma(z_k)\big(1 - \sigma(z_k)\big)\frac{\partial z_k^{(1)}}{\partial w_{jk}}$$

Write $\sigma(z_k)$ as o_k:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}} = (o_k - t_k)o_k(1 - o_k)\frac{\partial z_k^{(1)}}{\partial w_{jk}}$$

Consider $\frac{\partial z_k^{(1)}}{\partial w_{jk}} = x_j$:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}} = (o_k - t_k)o_k(1 - o_k)x_j$$

It can be seen that the partial derivative of w_{jk} is only related to the output node $o_k^{(1)}$ of the current connection, the label $t_k^{(1)}$ of the corresponding true, and the corresponding input node x_j.

Let $\delta_k = (o_k - t_k)o_k(1 - o_k)$, $\frac{\partial \mathcal{L}}{\partial w_{jk}}$ becomes:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}} = \delta_k x_j$$

The variable δ_k characterizes a certain characteristic of the error gradient propagation of the end node of the connection line. After using the representation δ_k, the partial derivative $\frac{\partial \mathcal{L}}{\partial w_{jk}}$ is only related to the start node x_j and the end node δ_k of the current connection. Later we will see the role of δ_k in cyclically deriving gradients.

Now that the gradient propagation method of the single-layer neural network (that is, the output layer) has been derived, next we try to derive the gradient propagation method of the penultimate layer. After completing the propagation derivation of the penultimate layer, similarly, the gradient propagation mode of all hidden layers can be derived cyclically to obtain gradient calculation expressions of all layer parameters.

Before introducing the back propagation algorithm, we first learn a core rule of derivative propagation - the chain rule.

7.6 Chain rule

Earlier, we introduced the gradient calculation method of the output layer. We now introduce the chain rule, which is a core formula that can derive the gradient layer by layer without explicitly deducing the mathematical expression of the neural network.

In fact, the chain rule has been used more or less in the process of deriving the gradient. Considering the compound function $y = f(u)$, $u = g(x)$, we can derive $\frac{dy}{dx}$ from $\frac{dy}{du}$ and $\frac{du}{dx}$:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} = f'(g(x)) \cdot g'(x)$$

Consider the compound function with two variables $z = f(x, y)$, where $x = g(t), y = h(t)$, then the derivative $\frac{dz}{dt}$ can be derived from $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$:

$$\frac{dz}{dt} = \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt}$$

For example, $z = (2t + 1)^2 + e^{t^2}$, let $x = 2t + 1, y = t^2$, then $z = x^2 + e^y$. Using above formula, we have:

$$\frac{dz}{dt} = \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt} = 2x \cdot 2 + e^y \cdot 2t$$

Let $x = 2t + 1, y = t^2$:

$$\frac{dz}{dt} = 2(2t + 1) \cdot 2 + e^{t^2} \cdot 2t$$

i.e.:

$$\frac{dz}{dt} = 4(2t + 1) + 2te^{t^2}$$

The loss function $\mathcal{L}$ of the neural network comes from each output node $o_k^{(K)}$, as shown in Figure 7.9 below, where the output node $o_k^{(K)}$ is associated with the output node $o_j^{(J)}$ of the hidden layer, so the chain rule is very suitable for the gradient derivation of the neural network. Let us consider how to apply the chain rule to the loss function.

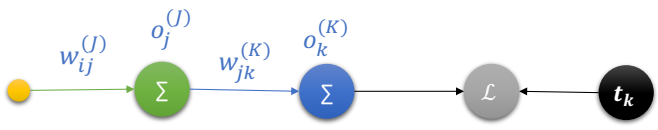

Figure 7.9 Gradient propagation illustration

In forward propagation, the data goes through $w_{ij}^{(J)}$ to the node $o_j^{(J)}$ in the penultimate layer and then propagates to the node $o_k^{(K)}$ in the output layer. When there is only one node per layer, the

chain rule can be used to decompose $\frac{\partial \mathcal{L}}{\partial w_{ij}^{(J)}}$ layer by layer into:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(J)}} = \frac{\partial \mathcal{L}}{\partial o_j^{(J)}} \frac{\partial o_j^{(J)}}{\partial w_{ij}^{(J)}} = \frac{\partial \mathcal{L}}{\partial o_k^{(K)}} \frac{\partial o_k^{(K)}}{\partial o_j^{(J)}} \frac{\partial o_j^{(J)}}{\partial w_{ij}^{(J)}}$$

where 中$\frac{\partial \mathcal{L}}{\partial o_k^{(K)}}$ can be directly derived from the error function, and $\frac{\partial o_k^{(K)}}{\partial o_j^{(J)}}$ can be derived from the fully connected layer formula. The derivative $\frac{\partial o_j^{(J)}}{\partial w_{ij}^{(J)}}$ is the input $x_i^{(I)}$. It can be seen that through the chain rule, we do not need specific mathematical expressions for the derivative of $\mathcal{L} = f(w_{ij}^{(J)})$, instead we can directly decompose the partial derivatives, and iteratively derive the derivatives layer by layer.

Here we simply use TensorFlow automatic derivation function to experience the charm of the chain rule.

```
import tensorflow as tf
# Create vectors
x = tf.constant(1.)
w1 = tf.constant(2.)
b1 = tf.constant(1.)
w2 = tf.constant(2.)
b2 = tf.constant(1.)
# Create gradient recorder
with tf.GradientTape(persistent=True) as tape:
    # Manually record gradient info for non-tf.Variable variables
    tape.watch([w1, b1, w2, b2])
    # Create two layer neural network
    y1 = x * w1 + b1
    y2 = y1 * w2 + b2

# Solve partial derivatives
dy2_dy1 = tape.gradient(y2, [y1])[0]
dy1_dw1 = tape.gradient(y1, [w1])[0]
dy2_dw1 = tape.gradient(y2, [w1])[0]

# Valdiate chain rule
print(dy2_dy1 * dy1_dw1)
print(dy2_dw1)
```

In above code, we calculated $\frac{\partial y_2}{\partial y_1}$, $\frac{\partial y_1}{\partial w_1}$ and $\frac{\partial y_2}{\partial w_1}$ through auto-gradient-calculation in Tensorflow and through chain rule we know $\frac{\partial y_2}{\partial y_1} \cdot \frac{\partial y_1}{\partial w_1}$ and $\frac{\partial y_2}{\partial w_1}$ should be equal. Their results are as follow:

```
tf.Tensor(2.0, shape=(), dtype=float32)
tf.Tensor(2.0, shape=(), dtype=float32)
```

7.7 Back propagation algorithm

Now let's derive the gradient propagation law of the hidden layer. Briefly review the partial derivative formula of the output layer:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}} = (o_k - t_k)o_k(1 - o_k)x_j = \delta_k x_j$$

Consider the partial derivative of the penultimate layer $\frac{\partial \mathcal{L}}{\partial w_{ij}}$, as shown in Figure 7.10. The number of output layer nodes is K, and the output is $\boldsymbol{o}^{(K)} = \left[o_1^{(K)}, o_2^{(K)}, \cdots, o_K^{(K)}\right]$. The penultimate layer has J nodes，and output is $\boldsymbol{o}^{(J)} = \left[o_1^{(J)}, o_2^{(J)}, \cdots, o_J^{(J)}\right]$. The antepenultimate layer has I nodes，and the output is $\boldsymbol{o}^{(I)} = \left[o_1^{(I)}, o_2^{(I)}, \cdots, o_I^{(I)}\right]$.

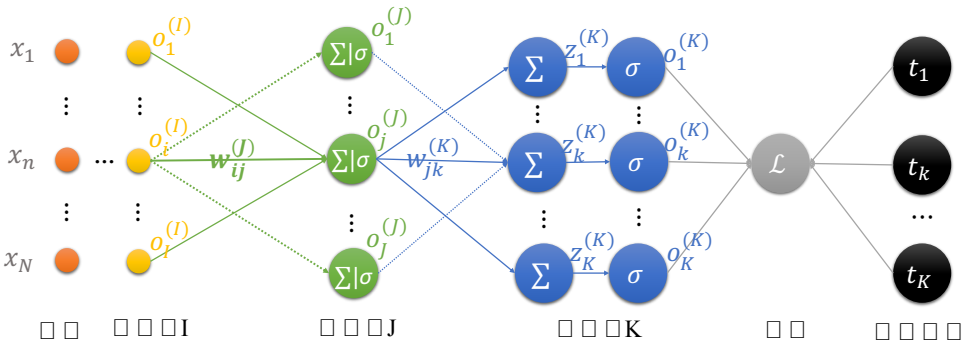

Figure 7.10 Back propagation algorithm

In order to express conciseness, the superscripts of some variables are sometimes omitted. First expand the mean square error function:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \frac{1}{2} \sum_k (o_k - t_k)^2$$

Because $\mathcal{L}$ is associated with w_{ij} through each output node o_k, the summation sign cannot be removed here, and the mean square error function can be disassembled using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \sum_k (o_k - t_k) \frac{\partial}{\partial w_{ij}} o_k$$

Substitute $o_k = \sigma(z_k)$:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \sum_k (o_k - t_k) \frac{\partial}{\partial w_{ij}} \sigma(z_k)$$

The derivative of the Sigmoid function is $\sigma' = \sigma(1 - \sigma)$, so：

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \sum_k (o_k - t_k)\sigma(z_k)\big(1 - \sigma(z_k)\big)\frac{\partial z_k}{\partial w_{ij}}$$

Write $\sigma(z_k)$ as o_k, and consider chain rule, we have:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \sum_k (o_k - t_k)o_k(1 - o_k)\frac{\partial z_k}{\partial o_j}\cdot\frac{\partial o_j}{\partial w_{ij}}$$

where $\frac{\partial z_k}{\partial o_j} = w_{jk}$, so:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \sum_k (o_k - t_k)o_k(1 - o_k)w_{jk}\frac{\partial o_j}{\partial w_{ij}}$$

Because $\frac{\partial o_j}{\partial w_{ij}}$ is not associated with *k*, we have:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial o_j}{\partial w_{ij}}\sum_k (o_k - t_k)o_k(1 - o_k)w_{jk}$$

Because $o_j = \sigma\big(z_j\big)$ and $\sigma' = \sigma(1 - \sigma)$, we have:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = o_j\big(1 - o_j\big)\frac{\partial z_j}{\partial w_{ij}}\sum_k (o_k - t_k)o_k(1 - o_k)w_{jk}$$

where $\frac{\partial z_j}{\partial w_{ij}}$ is o_i, so:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = o_j\big(1 - o_j\big)o_i\sum_k \underbrace{(o_k - t_k)o_k(1 - o_k)}_{\delta_k^{(K)}} w_{jk}$$

where $\delta_k^{(K)} = (o_k - t_k)o_k(1 - o_k)$, so:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = o_j\big(1 - o_j\big)o_i\sum_{\mathrm{k}} \delta_k^{(K)}w_{jk}$$

Similarly as the format of $\frac{\partial \mathcal{L}}{\partial w_{jk}} = \delta_k^{(K)}x_j$, define δ_j^J as:

$$\delta_j^J \triangleq o_j\big(1 - o_j\big)\sum_k \delta_k^{(K)}w_{jk}$$

At this time, $\frac{\partial \mathcal{L}}{\partial w_{ij}}$ can be written as a simple multiplication of the output value o_i of the currently connected start node and the gradient variable information $\delta_j^{(J)}$ of the end node:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \delta_j^{(J)}o_i^{(I)}$$

It can be seen that by defining variable δ, the gradient expression of each layer becomes more clear and concise, where δ can be simply understood as the contribution value of the current weight w_{ij} to the error function.

Let's summarize the propagation law of the partial derivative of each layer.

Output layer:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}} = \delta_k^{(K)} o_j$$

$$\delta_k^{(K)} = o_k(1 - o_k)(o_k - t_k)$$

Penultimate layer：

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \delta_j^{(J)} o_i$$

$$\delta_j^{(J)} = o_j\left(1 - o_j\right) \sum_k \delta_k^{(K)} w_{jk}$$

Antepenultimate layer：

$$\frac{\partial \mathcal{L}}{\partial w_{ni}} = \delta_i^{(I)} o_n$$

$$\delta_i^{(I)} = o_i(1 - o_i) \sum_j \delta_j^{(J)} w_{ij}$$

where o_n is the input of the antepenultimate layer.

According to this law, the partial derivative of the current layer can be obtained only by calculating the values $\delta_k^{(K)}$, $\delta_j^{(J)}$, and $\delta_i^{(I)}$ of each node of each layer iteratively, so as to obtain the gradient of the weight matrix $\boldsymbol{W}$ of each layer, and then iteratively optimize the network parameters through the gradient descent algorithm.

So far, the back propagation algorithm is fully introduced.

Next, we will conduct two hands-on cases: the first case is to use the automatic derivation provided by TensorFlow to optimize the extreme value of the Himmelblau function. The second case is to implement the back propagation algorithm based on Numpy and complete the multi-layer neural network training for binary classification problem.

7.8 Hangs-on optimization of Himmelblau

The Himmelblau function is one of the commonly used sample functions for testing optimization algorithms. It contains two independent variables x and y, and the mathematical expression is:

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

First, we implement the expression of the Himmelblau function through the following code:

```
def himmelblau(x):
    # Himmelblau function implementation. Input x is a list with 2 elements.
    return (x[0] ** 2 + x[1] - 11) ** 2 + (x[0] + x[1] ** 2 - 7) ** 2
```

Then we complete the visualization of the Himmelblau function. Use np.meshgrid function (meshgrid function is also available in TensorFlow) to generate two-dimensional plane grid point coordinates as follows:

```
x = np.arange(-6, 6, 0.1) # x-axis
y = np.arange(-6, 6, 0.1) # y-axis
print('x,y range:', x.shape, y.shape)
X, Y = np.meshgrid(x, y)
print('X,Y maps:', X.shape, Y.shape)
Z = himmelblau([X, Y])
```

Use the Matplotlib library to visualize the Himmelblau function, as shown in Figure 7.11:

```
# Plot the Himmelblau function
fig = plt.figure('himmelblau')
ax = fig.gca(projection='3d')
ax.plot_surface(X, Y, Z)
ax.view_init(60, -30)
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```

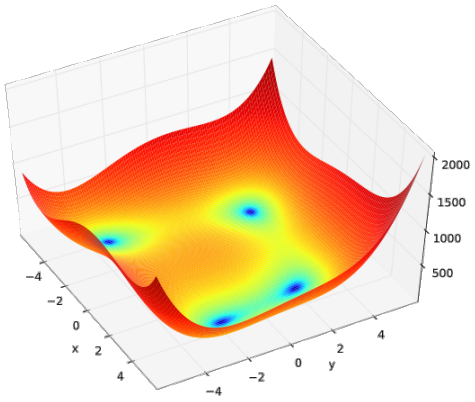

Figure 7.11 Himmelblau function

Figure 7.12 is a contour map of the Himmelblau function. It can be roughly seen that it has 4 local minimum points and the local minimum values are all 0, so these 4 local minimum values are also global minimum values. We can calculate the precise coordinates of the local minimum by analytical methods, they are:

$$(3,2), (-2.805, 3.131), (-3.779, -3.283), (3.584, -1.848)$$

Knowing the analytical solution of the extreme value, we now use the gradient descent algorithm to optimize the minimum numerical solution of the Himmelblau function.

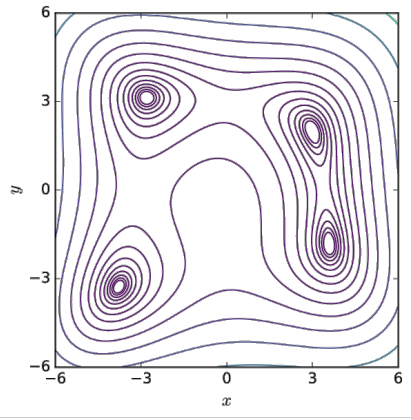

Figure 7.12 Himmelblau function contour plot

We can use TensorFlow automatic derivation to find the partial derivative of the sum of the function, and iteratively update the sum value as follows:

```
# The influence of the initialization value of the parameter on the
optimization cannot be ignored, you can try different initialization values
# Test the minimum value of function optimization
# [1., 0.], [-4, 0.], [4, 0.]
x = tf.constant([4., 0.]) # Initialization

for step in range(200):# Loop 200 times
    with tf.GradientTape() as tape: #record gradient
        tape.watch([x]) # Add to the gradient recording list
        y = himmelblau(x) # forward propagation
    # backward propagration
    grads = tape.gradient(y, [x])[0]
    # update paramaters with learning rate of 0.01
    x -= 0.01*grads
    # print info
    if step % 20 == 19:
        print ('step {}: x = {}, f(x) = {}'
               .format(step, x.numpy(), y.numpy()))
```

After 200 iterations of updating, the program can find a minimum solution, at which point the function value is close to zero. The numerical solution is

```
step 199: x = [ 3.584428  -1.8481264], f(x) = 1.1368684856363775e-12
```

This is almost the same as one of the analytical solutions $(3.584, -1.848)$.

In fact, by changing the initialization state of the network parameters, the program can obtain a variety of minimum numerical solutions. The initialization state of the parameters may affect the search trajectory of the gradient descent algorithm, and it may even search out completely different numerical solutions, as shown in Table 7.1. This example explains the effect of different initial states on the gradient descent algorithm.

Table 7.1 The effect of initial values on optimization results

$Initial\ value\ of\ x$	*Numerical solution*	*Analytical solution*
$(\mathbf{4}, \mathbf{0})$	(3.58,-1.84)	(3.58,-1.84)
(1,0)	(3,1.99)	(3,2)
(-4,0)	(-3.77,-3.28)	(-3.77,-3.28)
(-2,2)	(-2.80,3.13)	(-2.80,3.13)

7.9 Hands-on back propagation algorithm

In this section, we will use the gradient derivation results of the multi-layer fully connected network introduced earlier, and directly use Python to calculate the gradient of each layer, and manually update according to the gradient descent algorithm. Since TensorFlow has an automatic derivation function, we choose Numpy without the automatic derivation functionality to implement the network, and use Numpy to manually calculate the gradient and manually update the network parameters.

It should be noted that the gradient propagation formula derived in this chapter is for multiple fully connected layers with only Sigmoid function, and the loss function is a network type of mean square error function. For other types of networks, such as networks with ReLU activation function and cross-entropy loss function, the gradient propagation expression needs to be derived again, but the method is similar. It is precisely because the method of manually deriving the gradient is more limited, it is rarely used in practice.

We will implement a 4-layer fully connected network to complete the binary classification task. The number of network input nodes is 2, and the number of nodes in the hidden layer is designed as 20, 50 and 25. The two nodes in the output layer represent the probability of belonging to category 1 and 2, respectively, as shown in Figure 7.13. Here, the Softmax function is not used to constrain the sum of the network output probability values. Instead, the mean square error function is directly used to calculate the error between prediction and the one-hot encoded real label. All activation functions are Sigmoid. This design is to directly use our gradient propagation formula.

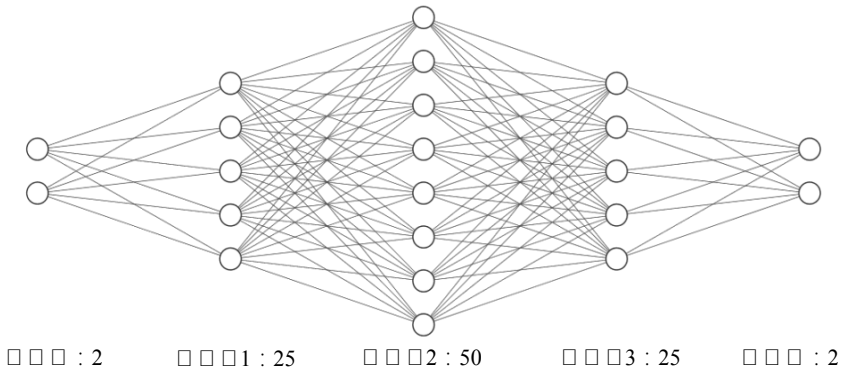

Figure 7.13 Network structure

7.9.1 Data set

Through the convenient tool provided by the scikit-learn library, 2000 linear inseparable 2-class data sets are generated. The feature length of the data is 2. The sampled data distribution is shown in Figure 7.14. The red points are in one category and the blue points belong to the other category. The distribution of each category is crescent-shaped and is linearly inseparable, which means a linear network cannot be used to obtain good results. In order to test the performance of the network, we divide the training set and the test set according to the ratio 7:3. $2000 \cdot 0.3 = 600$ sample points is used for testing and does not participate in the training. The remaining 1400 points are used for network training.

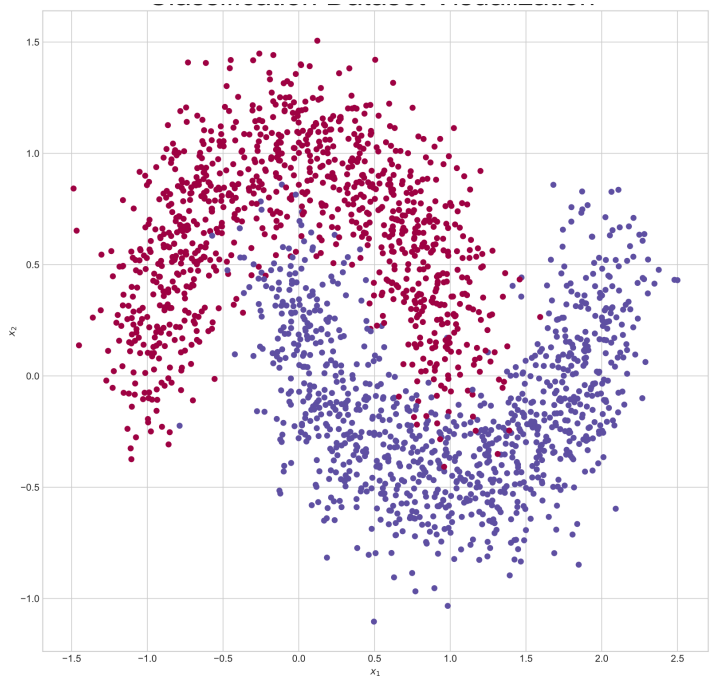

Figure 7.14 Data set distribution

The collection of the data set is directly generated using the make_moons function provided by scikit-learn, and the number of sampling points and testing ratio are set as follows:

```
N_SAMPLES = 2000 # number of sampling points
TEST_SIZE = 0.3 # testing ratio
# Use make_moons function to generate data set
X, y = make_moons(n_samples = N_SAMPLES, noise=0.2, random_state=100)
# Split traning and testing data set
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=TEST_SIZE, random_state=42)
print(X.shape, y.shape)
```

The distribution of the data set can be drawn by the following visualization code, as shown in Figure 7.14.

```
# Make a plot
```

```
def make_plot(X, y, plot_name, file_name=None, XX=None, YY=None, preds=None,
dark=False):
    if (dark):
        plt.style.use('dark_background')
    else:
        sns.set_style("whitegrid")
    plt.figure(figsize=(16,12))
    axes = plt.gca()
    axes.set(xlabel="$x_1$", ylabel="$x_2$")
    plt.title(plot_name, fontsize=30)
    plt.subplots_adjust(left=0.20)
    plt.subplots_adjust(right=0.80)
    if(XX is not None and YY is not None and preds is not None):
        plt.contourf(XX, YY, preds.reshape(XX.shape), 25, alpha = 1,
cmap=cm.Spectral)
        plt.contour(XX, YY, preds.reshape(XX.shape), levels=[.5],
cmap="Greys", vmin=0, vmax=.6)
    # Use color to distinguish labels
    plt.scatter(X[:, 0], X[:, 1], c=y.ravel(), s=40, cmap=plt.cm.Spectral,
edgecolors='none')

    plt.savefig('dataset.svg')
    plt.close()
# Make distribution plot
make_plot(X, y, "Classification Dataset Visualization ")
plt.show()
```

7.9.2 Network layer

A new Layer class is used to implement a network layer. Parameters such as the number of input nodes, the number of output nodes, and the type of activation function are passed into the network layer. The weights and bias tensor bias are automatically generated based on the number of input and output nodes during initialization as below:

```
class Layer:
    # Fully connected layer
    def __init__(self, n_input, n_neurons, activation=None, weights=None,
bias=None):
        """
        :param int n_input: input nodes
        :param int n_neurons: output nodes
        :param str activation: activation function
        :param weights: weight vectors
        :param bias: bias vectors
```

```
        """
        # Initialize weights through Normal distribution
        self.weights = weights if weights is not None else
np.random.randn(n_input, n_neurons) * np.sqrt(1 / n_neurons)
        self.bias = bias if bias is not None else np.random.rand(n_neurons) *
0.1
        self.activation = activation # activation function, e.g. 'sigmoid'
        self.last_activation = None # output of activation function o
        self.error = None
        self.delta = None
```

The forward propagation function of the network layer is implemented as follows, where the last_activation variable is used to save the output value of the current layer:

```
    def activate(self, x):
        # Forward propagation function
        r = np.dot(x, self.weights) + self.bias  # X@W+b
        # Get output through activation function
        self.last_activation = self._apply_activation(r)
        return self.last_activation
```

The self._apply_activation function in the above code implements the forward calculation process of different types of activation functions, although here we only use the Sigmoid activation function.

```
    def _apply_activation(self, r):
        # Calculate output of activation function
        if self.activation is None:
            return r # No activation function
        # ReLU
        elif self.activation == 'relu':
            return np.maximum(r, 0)
        # tanh
        elif self.activation == 'tanh':
            return np.tanh(r)
        # sigmoid
        elif self.activation == 'sigmoid':
            return 1 / (1 + np.exp(-r))

        return r
```

For different types of activation functions, their derivatives are calculated as follows:

```
    def apply_activation_derivative(self, r):
        # Calculate the derivative of activation functions
        # If no activation function, derivative is 1
        if self.activation is None:
            return np.ones_like(r)
```

```
        # ReLU
        elif self.activation == 'relu':
            grad = np.array(r, copy=True)
            grad[r > 0] = 1.
            grad[r <= 0] = 0.
            return grad
        # tanh
        elif self.activation == 'tanh':
            return 1 - r ** 2
        # Sigmoid
        elif self.activation == 'sigmoid':
            return r * (1 - r)

        return r
```

It can be seen that the derivative of the Sigmoid function is implemented as $r\,(1 - r)$, where r is $\sigma(z)$.

7.9.3 Network model

After creating a single-layer network class, we implement the NeuralNetwork class of the network model, which internally maintains the network Layer object of each layer. You can add the network layer through the add_layer function to achieve the purpose of creating a network model with different structures as below:

```
class NeuralNetwork:
    # Neural Network Class
    def __init__(self):
        self._layers = []  # list of network class

    def add_layer(self, layer):
        # Add layers
        self._layers.append(layer)
```

The forward propagation of the network only needs to cyclically adjust the forward calculation function of each network layer object. The code is as follows:

```
    def feed_forward(self, X):
        # Forward calculation
        for layer in self._layers:
            # Loop through every layer
            X = layer.activate(X)
        return X
```

According to the network structure configuration in Figure 7.13, we use the NeuralNetwork class to create a network object and add a 4-layer fully connected network. The code is as follows:

```
nn = NeuralNetwork()
nn.add_layer(Layer(2, 25, 'sigmoid'))  # Hidden layer 1, 2=>25
```

```
nn.add_layer(Layer(25, 50, 'sigmoid')) # Hidden layer 2, 25=>50
nn.add_layer(Layer(50, 25, 'sigmoid')) # Hidden layer 3, 50=>25
nn.add_layer(Layer(25, 2, 'sigmoid'))  # Hidden layer, 25=>2
```

The back propagation of the network model is slightly more complicated. We need to start from the last layer and calculate the variable δ of each layer, and then store the calculated variable δ in the delta variable of the Layer class according to the derived gradient formula as below:

```
    def backpropagation(self, X, y, learning_rate):
        # Back propagation
        # Get result of forward calculation
        output = self.feed_forward(X)
        for i in reversed(range(len(self._layers))):  # reverse loop
            layer = self._layers[i]  # get current layer
            # If it's output layer
            if layer == self._layers[-1]:  # output layer
                layer.error = y - output
                # calculate delta
                layer.delta = layer.error *
layer.apply_activation_derivative(output)
            else:  # For hidden layer
                next_layer = self._layers[i + 1]
                layer.error = np.dot(next_layer.weights, next_layer.delta)
                # Calculate delta
                layer.delta = layer.error *
layer.apply_activation_derivative(layer.last_activation)
                … # See following code
```

After the reverse calculation of the variable δ of each layer, it is only necessary to calculate the gradient of the parameters of each layer according to the formula $\frac{\partial \mathcal{L}}{\partial w_{ij}} = o_i \delta_j^{(J)}$ and update the network parameters. Because the delta in the code is actually calculated as $-\delta$, the plus sign is used when updating. Code is as below:

```
    def backpropagation(self, X, y, learning_rate):
        … # Continue above code
        # Update weights
        for i in range(len(self._layers)):
            layer = self._layers[i]
            # o_i is output of previous layer
            o_i = np.atleast_2d(X if i == 0 else self._layers[i -
1].last_activation)
            # Gradient descent
            layer.weights += layer.delta * o_i.T * learning_rate
```

Therefore, in the back propagation function, the variable δ of each layer are reversely

calculated, and the gradient values of the parameters of each layer are calculated according to the gradient formula, and the parameter update is completed according to the gradient descent algorithm.

7.9.4 Network training

The binary classification network here is designed with two output nodes, so the real label needs to be one-hot encoded. The code is as follows:

```
    def train(self, X_train, X_test, y_train, y_test, learning_rate,
max_epochs):
        # Train network
        # one-hot encoding
        y_onehot = np.zeros((y_train.shape[0], 2))
        y_onehot[np.arange(y_train.shape[0]), y_train] = 1
```

Calculate the mean square error of the one-hot encoded real label and the output of the network, and call the back propagation function to update the network parameters, and iterate the training set 1000 times as below:

```
        mses = []
        for i in range(max_epochs):  # Train 1000 epoches
            for j in range(len(X_train)):  # Train one sample per time
                self.backpropagation(X_train[j], y_onehot[j], learning_rate)
            if i % 10 == 0:
                # Print MSE Loss
                mse = np.mean(np.square(y_onehot - self.feed_forward(X_train)))
                mses.append(mse)
                print('Epoch: #%s, MSE: %f' % (i, float(mse)))

                # Print accuracy
                print('Accuracy: %.2f%%' % (self.accuracy(self.predict(X_test),
y_test.flatten()) * 100))

        return mses
```

7.9.5 Network performance

We record the training loss value $\mathcal{L}$ of each Epoch and draw it as a curve, as shown in Figure 7.15.

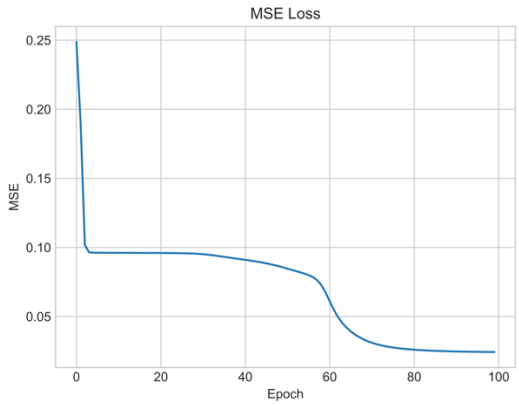

Figure 7.15 Training error plot

After training 1000 Epochs, the accuracy rate obtained on 600 samples in the test set is:

```
Epoch: #990, MSE: 0.024335
Accuracy: 97.67%
```

It can be seen that by manually calculating the gradient formula and manually updating the network parameters, we can also obtain a lower error rate for simple binary classification tasks. Through fine-tuning network hyperparameters and other techniques, you can also get better network performance.

In each Epoch, we complete an accuracy test on the test set and draw it into a curve, as shown in Figure 7.16. It can be seen that with the progress of Epoch, the accuracy of the model has been steadily improved, the initial stage is faster, and the subsequent improvement is relatively smooth.

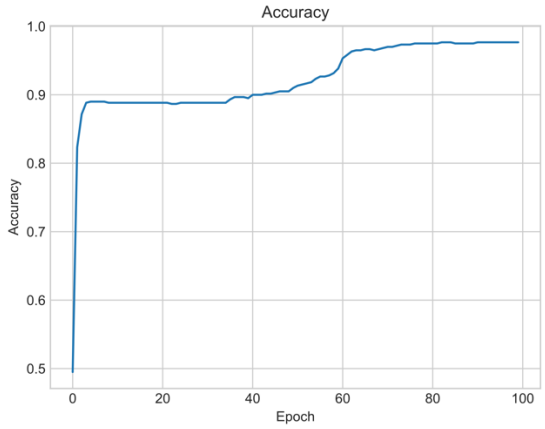

Figure 7.16 Testing accuracy

Through this binary classification fully connected network based on Numpy's manual calculation of gradients, I believe readers can more deeply appreciate the role of deep learning frameworks in algorithm implementation. Without frameworks such as TensorFlow, we can also implement complex neural networks, but flexibility, stability, development efficiency, and computational efficiency are poor. Algorithm design and training based on these deep learning frameworks will greatly improve the work of algorithm developers. effectiveness. At the same

time, we can also realize that the framework is just a tool. More importantly, our understanding of the algorithm itself is the most important ability of algorithm developers.

7.10 Reference

[1] D. E. Rumelhart, G. E. Hinton and R. J. Williams, “{Learning Representations by Back-propagating Errors},” *Nature,* 323, 6088, pp. 533-536, 1986.

[2] Nick. A brief history of Artifical Intellegence, Turing Education, 2017.