Chapter 6 Neural Networks

It is difficult to imagine which big industry will not be changed by artificial intelligence. Artificial intelligence will play a major role in these industries, and this trend is very obvious. – Andrew Ng

The ultimate goal of machine learning is to find a good set of parameters, so that the model can learn the mapping relationship $f_\theta: \boldsymbol{x} \to \boldsymbol{y},\ \boldsymbol{x}, \boldsymbol{y} \in \mathbb{D}^{\text{train}}$ from the training set, and use the trained relationship to predict new samples. Neural networks belong to a branch of research in machine learning. It specifically refers to a model that uses multiple neurons to parameterize the mapping function f_θ.

6.1 Perceptron

In 1943, American neuroscientist Warren Sturgis McCulloch and mathematical logicist Walter Pitts were inspired by the structure of biological neurons, and proposed a mathematical model of artificial neurons, which was further developed and proposed by American neurophysicist Frank Rosenblatt, which is known as Perceptron model. In 1957, Frank Rosenblatt implemented the perceptron model on an IBM-704 computer. This model can complete some simple visual classification tasks, such as distinguishing triangles, circles, and rectangles [1].

The structure of the perceptron model is shown in Figure 6.1. It accepts a one-dimensional vector of length n, $\boldsymbol{x} = [x_1, x_2, \dots, x_n]$, and each input node is aggregated as a variable through a connection of weights $w_i, i\epsilon[1, n]$, namely:

$$z = w_1\,x_1 + w_2\,x_2 + \cdots + w_n\,x_n + b$$

Among them, b is called the bias of the perceptron, and the one-dimensional vector $\boldsymbol{w} = [w_1, w_2, \dots, w_n]$ is called the weight of the perceptron, while z is called the net activation value of the perceptron.

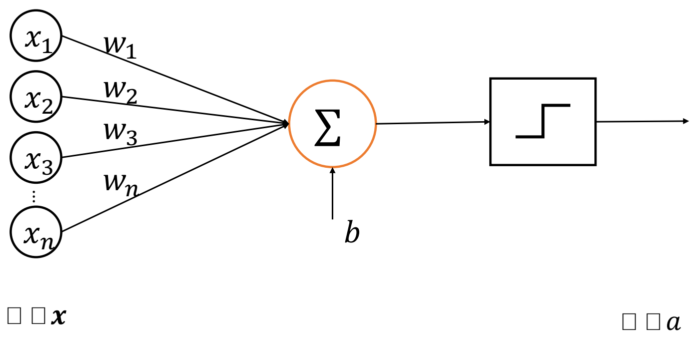


Figure 6.1 Perception model

The above formula can be written in vector form:

$$z = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b$$

Perceptron is a linear model and cannot deal with linear inseparability. The activation value is obtained by adding the activation function after the linear model:

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

The activation function can be a step function. As shown in Figure 6.2, the output of the step function is only 0/1. When $z < 0$, 0 was then output, representing category 0; when $z \geq 0$, 1 was the output, representing category 1, namely:

$$a = \begin{cases} 1 & \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b \geq 0 \\ 0 & \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b < 0 \end{cases}$$

It can also be a sign function as shown in Figure 6.3, and the expression is:

$$a = \begin{cases} 1 & \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b \geq 0 \\ -1 & \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b < 0 \end{cases}$$

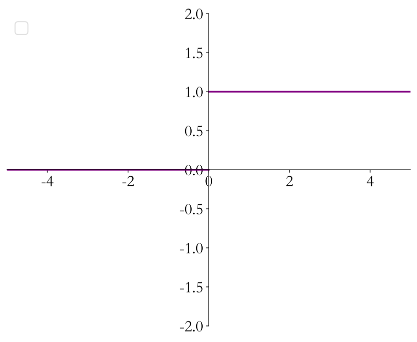


Figure 6.2 Step function

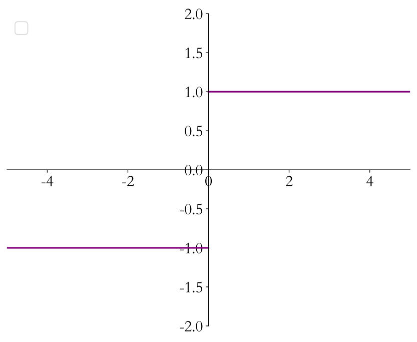


Figure 6.3 Sign function

After adding the activation function, the perceptron model can be used to complete the binary classification task. The step and the sign functions are discontinuous at $z = 0$, so the gradient descent algorithm cannot be used to optimize the parameters.

In order to enable the perceptron model to automatically learn from the data, Frank Rosenblatt proposed a perceptron learning algorithm, as shown in Algorithm 1.

Algorithm 1: Perceptron training algorithm

```
Initialize $\boldsymbol{w} = \boldsymbol{0}, \boldsymbol{b} = \boldsymbol{0}$
repeat
  Randomly select a sample $(\boldsymbol{x_i}, \boldsymbol{y_i})$ from training set
  Calculate the output $\boldsymbol{a} = \mathbf{sign}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x_i} + \boldsymbol{b})$
  If $\boldsymbol{a} \neq \boldsymbol{y_i}$:
    $\boldsymbol{w'} \leftarrow \boldsymbol{w} + \boldsymbol{\eta} \cdot \boldsymbol{y_i} \cdot \boldsymbol{x_i}$
    $\boldsymbol{b'} \leftarrow \boldsymbol{b} + \boldsymbol{\eta} \cdot \boldsymbol{y_i}$
until reach the required number of steps
Output: paramaters $\boldsymbol{w}$ and $\boldsymbol{b}$
```

Here η is learning rate.

Although the perceptron model has been put forward with good development potential, Marvin Lee Minsky and Seymour Papert proved that the linear model represented by the perceptron cannot solve the linear inseparability problem (XOR) in the "Perceptrons" book in 1969, which directly led to the emergence of neural network research to a bottom at the time. Although the perceptron model cannot solve the linear inseparable problem, the book also mentions that it can be solved by nesting multiple layers of neural networks.

6.2 Fully connected layer

The underivable nature of the perceptron model severely constrains its potential, making it only capable of solving extremely simple tasks. In fact, modern deep learning models have a parameter scale of millions or even hundreds of millions, but the core structure is not much different from the perceptron model. On the basis of the perceptron model, they replace the discontinuous step activation function with other smooth continuous derivable activation functions, and stacks multiple network layers to enhance the expressive power of the network.

In this section, we replace the activation function of the perceptron model and stack multiple neurons in parallel to achieve a multi-input and multi-output network layer structure. As shown in Figure 6.4, two neurons are stacked in parallel, that is, two perceptrons with replaced activation functions, forming a network layer of 3 input nodes and 2 output nodes. The first output node is:

$$o_1 = \sigma(w_{11} \cdot x_1 + w_{21} \cdot x_2 + w_{31} \cdot x_3 + b_1)$$

The output of the second node is:

$$o_2 = \sigma(w_{12} \cdot x_1 + w_{22} \cdot x_2 + w_{32} \cdot x_3 + b_2)$$

Putting them together, the output vector is $\boldsymbol{o} = [o_1, o_2]$. The entire network layer can be expressed by the matrix relationship:

$$[o_1 \quad o_2] = [x_1 \quad x_2 \quad x_3]@\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} + [b_1 \quad b_2] \tag{6-1}$$

i.e.

$$\boldsymbol{O} = \boldsymbol{X}@\boldsymbol{W} + \boldsymbol{b}$$

The shape of the input matrix $\boldsymbol{X}$ is defined as $[b, d_{\text{in}}]$, while the number of samples is b and the number of input nodes is d_{in}. The shape of the weight matrix W is defined as $[d_{\text{in}}, d_{\text{out}}]$, while the number of output nodes is d_{out}, and the shape of the offset vector b is $[d_{\text{out}}]$.

Considering 2 samples, $\boldsymbol{x}^{(1)} = \left[x_1^{(1)}, x_2^{(1)}, x_3^{(1)}\right]$, $\boldsymbol{x}^{(2)} = \left[x_1^{(2)}, x_2^{(2)}, x_3^{(2)}\right]$, the above equation can also be written as:

$$\begin{bmatrix} o_1^{(1)} & o_2^{(1)} \\ o_1^{(2)} & o_2^{(2)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} \end{bmatrix} @ \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} + [b_1 \quad b_2]$$

Among it, the output matrix $\boldsymbol{O}$ contains the output of b samples, and the shape is $[b, d_{\text{out}}]$. Since each output node is connected to all input nodes, this network layer is called a Fully Connected Layer, or a Dense Layer, with $\boldsymbol{W}$ as weight matrix and $\boldsymbol{b}$ is the bias vector.

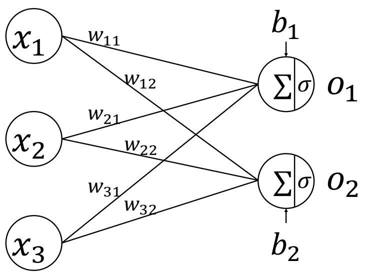


Figure 6.4 Fully connected layer

6.2.1 Tensor mode implementation

In TensorFlow, to achieve a fully connected layer, you only need to define the weight tensor $\boldsymbol{W}$ and bias tensor $\boldsymbol{b}$, and use the batch matrix multiplication function tf.matmul() provided by TensorFlow to complete the calculation of the network layer. For example, for a input matrix $\boldsymbol{X}$ with 2 samples and input feature length of each sample $d_{\text{in}} = 784$ and the number of output nodes $d_{\text{out}} = 256$, the shape of the weight matrix $\boldsymbol{W}$ is $[784,256]$. The shape of the bias vector $\boldsymbol{b}$ is $[256]$. After the addition, the shape of the output layer is $[2,256]$, that is, the features of 2 samples with each feature length as 256. Tge code is implemented as follows:

```
In [1]:
x = tf.random.normal([2,784])
w1 = tf.Variable(tf.random.truncated_normal([784, 256], stddev=0.1))
b1 = tf.Variable(tf.zeros([256]))
o1 = tf.matmul(x,w1) + b1  # linear transformation
o1 = tf.nn.relu(o1)  # activation function
Out[1]:
  <tf.Tensor: id=31, shape=(2, 256), dtype=float32, numpy=
  array([[ 1.51279330e+00,  2.36286330e+00,  8.16453278e-01,
          1.80338228e+00,  4.58602428e+00,  2.54454136e+00,…
```

In fact, we have used the above code many times to implement network layers.

6.2.2 Layer implementation

The fully connected layer is essentially matrix multiplication and addition operations. But as one of the most commonly used network layers, TensorFlow has a more convenient implementation method: layers.Dense(units, activation). Through the layer.Dense class, you only need to specify the number of output nodes (units) and activation function type (activation). It should be noted that the number of input nodes will be determined according to the input shape during the first operation, and the weight tensor and bias tensor will be automatically created and initialized based on the number of input and output nodes. The weight tensor and bias tensor will not be created immediately due to lazy evaluation. The build function or direct calculation will be required to complete the creation of the network parameters. The activation parameter specifies the activation function of the current layer, which can be a common activation function or a custom activation function, or be specified as None, that is, no activation function.

```
In [2]:
x = tf.random.normal([4,28*28])
from tensorflow.keras import  layers
# Create fully-connected layer with output nodes and activation function
fc = layers.Dense(512, activation=tf.nn.relu)
h1 = fc(x)  # calculate and return a new tensor
Out[2]:
<tf.Tensor: id=72, shape=(4, 512), dtype=float32, numpy=
array([[0.63339347, 0.21663809, 0.        , ..., 1.7361937 , 0.39962345,
        2.4346168 ],…
```

We can create a fully connected layer fc with a single line of code in above code with the number of output nodes as 512 and the number of input nodes automatically obtained during calculation. The code creates internal weight tensor and bias tensor automatically as well. We can obtain the weight and bias tensor object through the class member kernel and bias within the class:

```
In [3]: fc.kernel # Get the weight tensor
Out[3]:
<tf.Variable 'dense_1/kernel:0' shape=(784, 512) dtype=float32, numpy=
array([[-0.04067389,  0.05240148,  0.03931375, ..., -0.01595572,
        -0.01075954, -0.06222073],
In [4]: fc.bias # Get the bias tensor
Out[4]:
<tf.Variable 'dense_1/bias:0' shape=(512,) dtype=float32, numpy=
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,…])
```

It can be seen that the shape of the weight and the bias tensor are in line with our understanding. When optimizing parameters, we need to obtain a list of all tensor parameters to be optimized in the network, which can be done through the class trainable_variables.

```
In [5]: fc.trainable_variables
Out[5]:  # Return all parameters to be optimized
 [<tf.Variable 'dense_1/kernel:0' shape=(784, 512) dtype=float32,…,
 <tf.Variable 'dense_1/bias:0' shape=(512,) dtype=float32, numpy=…]
```

In fact, the network layer saves not only the list of trainable_variables to be optimized, but also tensors that do not participate in gradient optimization. For example, the Batch Normalization layer can return all parameter lists that do not need optimization through the non_trainable_variables member. If you want to get a list of all parameters, you can get all internal tensors through the variables member of the class, for example:

```
In [6]: fc.variables  # Get all parameters
Out[6]:
[<tf.Variable 'dense_1/kernel:0' shape=(784, 512) dtype=float32,…,
 <tf.Variable 'dense_1/bias:0' shape=(512,) dtype=float32, numpy=…]
```

For fully connected layers, all internal tensors participate in gradient optimization, so the list returned by variables is the same as trainable_variables.

When using the network layer class object for forward calculation, you only need to call the __call__ method of the class, that is, write it in the fc(x) mode, it will automatically call the __call__ method. This setting is automatically completed by the TensorFlow framework. For a fully connected layer class, the operation logic implemented in the call method is very simple.

6.3 Neural network

By stacking the fully connected layers in Figure 6.4 and ensuring that the number of output nodes of the previous layer matches the number of input nodes of the current layer, a network of any number of layers can be created, which is known as neural networks. As shown in Figure 6.5, by stacking 4 fully connected layers, a neural network with 4 layers can be obtained. Since each layer is a fully connected layer, it is called a fully connected network. Among them, the first to third fully connected layers are called hidden layers, and the output of the last fully connected layer is called the output layer of the network. The number of output nodes of the hidden layers is $[256,128,64]$ respectively, and the nodes of the output layer is 10.

When designing a fully connected network, the hyperparameters such as the configuration of the network can be set freely according to the rule of thumb, and only a few constraints need to be followed. For example, the number of input nodes in the first hidden layer needs to match the actual feature length of the data. The number of input layers in each layer matches the number of output nodes in the previous layer. The activation function and number of nodes in the output layer need to be set according to the specific settings of the required output. In general, the design of the neural network models has a greater degree of freedom. As shown in Figure 6.5, the number of output nodes in each layer does not have to be $[256,128,64,10]$ and can be freely matched, such as $[256,256,64,10]$ or $[512,64,32,10]$. As for which set of hyperparameters is optimal, it requires a lot of field experience and experimentation.

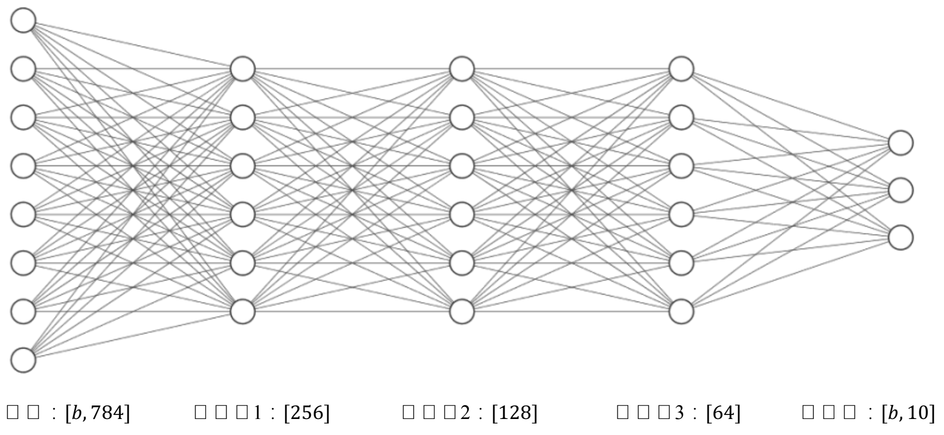


Figure 6.5 4-layer neural network

6.3.1 Tensor mode implementation

For a multi-layer neural network such as Figure 6.5, the weight matrix and bias vector of each layer need to be defined separately. The parameters of each layer can only be used for the corresponding layer, and should not be mixed. The network model in Figure 6.5 is implemented as follows:

```
# Hidden layer 1
w1 = tf.Variable(tf.random.truncated_normal([784, 256], stddev=0.1))
b1 = tf.Variable(tf.zeros([256]))
# Hidden layer 2
w2 = tf.Variable(tf.random.truncated_normal([256, 128], stddev=0.1))
b2 = tf.Variable(tf.zeros([128]))
# Hidden layer 3
w3 = tf.Variable(tf.random.truncated_normal([128, 64], stddev=0.1))
b3 = tf.Variable(tf.zeros([64]))
# Hidden layer 4
w4 = tf.Variable(tf.random.truncated_normal([64, 10], stddev=0.1))
b4 = tf.Variable(tf.zeros([10]))
```

When calculating, you only need to use the output of the previous layer as the input of the current layer, repeat until the last layer, and use the output of the output layer as the output of the network.

```
    with tf.GradientTape() as tape:
        # x: [b, 28*28]
        #  Hidden layer 1 forward calculation, [b, 28*28] => [b, 256]
        h1 = x@w1 + tf.broadcast_to(b1, [x.shape[0], 256])
        h1 = tf.nn.relu(h1)
        # Hidden layer 2 forward calculation, [b, 256] => [b, 128]
        h2 = h1@w2 + b2
        h2 = tf.nn.relu(h2)
        # Hidden layer 3 forward calculation, [b, 128] => [b, 64]
        h3 = h2@w3 + b3
        h3 = tf.nn.relu(h3)
        # Output layer forward calculation, [b, 64] => [b, 10]
        h4 = h3@w4 + b4
```

Whether the activation function needs to be added in the last layer usually depends on the specific task.

When using the TensorFlow automatic derivation function to calculate the gradient, the forward calculation process needs to be placed in the tf.GradientTape() environment, so that the gradient() method of the GradientTape object can be used to automatically solve the gradient of the parameter, and the parameter is updated by the optimizers object.

6.3.2 Layer mode implementation

For the conventional network layer, it is more concise and efficient to implement through the layer method. First, create new network layer classes and specify the activation function types of each layer:

```
#  Import layers modules
from tensorflow.keras import layers,Sequential

fc1 = layers.Dense(256, activation=tf.nn.relu) #  Hidden layer 1
fc2 = layers.Dense(128, activation=tf.nn.relu) #  Hidden layer 2
fc3 = layers.Dense(64, activation=tf.nn.relu) #  Hidden layer 3
fc4 = layers.Dense(10, activation=None) #  Output layer
x = tf.random.normal([4,28*28])
h1 = fc1(x)  #  Get output of hidden layer 1
h2 = fc2(h1) #  Get output of hidden layer 2
h3 = fc3(h2) #  Get output of hidden layer 3
h4 = fc4(h3) #  Get the network output
```

For such a network where data forwards in turn, it can also be encapsulated into a network class object through the Sequential container, and the forward calculation function of the class can be called once to complete the forward calculation of all layers. It is more convenient to use and is implemented as follows :

```
from tensorflow.keras import layers,Sequential

#  Encapsulate a neural network through Sequential container
model = Sequential([
    layers.Dense(256, activation=tf.nn.relu) , # Hidden layer 1
    layers.Dense(128, activation=tf.nn.relu) , # Hidden layer 2
    layers.Dense(64, activation=tf.nn.relu) , # Hidden layer 3
    layers.Dense(10, activation=None) , # Output layer
])
```

In forward calculation, you only need to call the large network objects once to complete the sequential calculation of all layers:

```
out = model(x)
```

6.3.3 Optimization

We call the calculation process of the neural network from input to output as Forward Propagation. The forward propagation process of the neural network is also the process of the flow of the data tensor from the first layer to the output layer. That is, from the input data, tensors are passed through each hidden layer, until the output is obtained and the error is calculated, which is also the origin of the TensorFlow framework name.

The final step of forward propagation is to complete the error calculation:

$$\mathcal{L} = g(f_\theta(\boldsymbol{x}), \boldsymbol{y})$$

In above formular, $f_\theta(\cdot)$ represents a neural network model with parameters θ. $g(\cdot)$ is called an error function, used to describe the gap between the predicted value of the current network $f_\theta(\boldsymbol{x})$ and the real label $\boldsymbol{y}$, such as the commonly used mean square error function. $\mathcal{L}$

is called the error or loss of the network, which is generally a scalar. We hope to minimize the training error $\mathcal{L}$ by learning a set of parameters on the training set $\mathbb{D}^{\text{train}}$:

$$\theta^* = \underbrace{\arg\min}_{\theta}\ g(f_\theta(\boldsymbol{x}), \boldsymbol{y}),\ x \in \mathbb{D}^{\text{train}}$$

The above minimization problem generally uses the Backward Propagation algorithm to solve and uses the Gradient Descent algorithm to iteratively update the parameters:

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

where η is the learning rate.

From another perspective to understand the neural network, it completes the function of feature dimension transformation, such as the 4-layer MNIST hand-written digital image recognition fully connected network, which in turn completes the feature dimensionality reduction process of $784 \to 256 \to 128 \to 64 \to 10$. The original features usually have higher dimensions and contain many low-level features and useless information. Through the layer-by-layer feature transformation, higher dimensions are reduced to lower dimensions where high-level abstract feature information highly correlated to the task is generally generated and specific task can be completed through simple logical determination of these features, such as the classification of pictures.

The amount of network parameters is an important indicator to measure the scale of the network. So how to calculate the amount of parameters of the fully connected layer? Consider a network layer with weight matrix $\boldsymbol{W}$, bias vector $\boldsymbol{b}$, input feature length d_{in}, and output feature length d_{out}. The number of parameters for $\boldsymbol{W}$ is $d_{\text{in}} \cdot d_{\text{out}}$. Adding the bias parameter, the total number of parameter is $d_{\text{in}} \cdot d_{\text{out}} + d_{\text{out}}$. For a multi-layer fully connected neural network, for example $784 \to 256 \to 128 \to 64 \to 10$, the expression of the total parameter amount is:

$$256 \cdot 784 + 256 + 128 \cdot 256 + 128 + 64 \cdot 128 + 64 + 10 \cdot 64 + 10 = 242762$$

The fully connected layer is the most basic type of neural network. It is very important for the research of subsequent neural network models, such as convolutional neural networks and recurrent neural networks. Through learning other network types, we will find that they, more or less, originate from the idea of a fully connected layer network. Because Geoffrey Hinton, Yoshua Bengio and Yann LeCun have insisted on research in the front-line of neural networks, they have made outstanding contributions to the development of artificial intelligence and won the Turing Award in 2018 (Figure 6.6, from the right are Yann LeCun, Geoffrey Hinton, and Yoshua Bengio).

Figure 6.6 2018 Turing Award Winners[①]

[①] Image source: https://www.theverge.com/2019/3/27/18280665/ai-godfathers-turing-award-2018-yoshua-bengio-geoffrey-hinton-yann-lecun

6.4 Activation function

Below we introduce common activation functions in neural networks. Unlike step and symbolic functions, these functions are smooth and derivable, and are suitable for gradient descent algorithms.

6.4.1 Sigmoid

The Sigmoid function is also called the Logistic function, which is defined as

$$\text{Sigmoid}(x) \triangleq \frac{1}{1+\mathrm{e}^{-x}}$$

One of its excellent features is the ability to "compress" the input $x \in R$ to an interval $x \in (0,1)$. The value of this interval is commonly used in machine learning to express the following meanings:

- **Probability distribution** The output of the interval $(0,1)$ matches the distribution range of probability. The output can be translated into a probability by the Sigmoid function
- **Signal strength** Usually, 0 ~ 1 can be understood as the strength of a certain signal, such as the color intensity of the pixel, 1 represents the strongest color of the current channel, 0 represents the current channel without color. It can also be used to represent the current Gate status, i.e. 1 means open and 0 indicates closed.

The Sigmoid function is continuously derivable, as shown in Figure 6.7. The gradient descent algorithm can be directly used to optimize the network parameters.

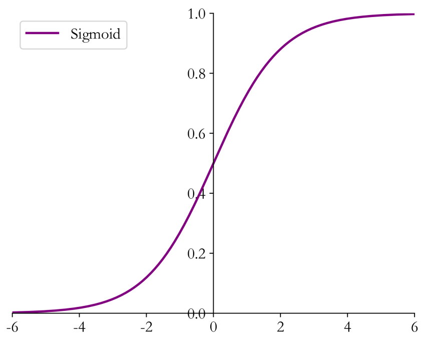


Figure 6.7 Sigmoid function

In TensorFlow, the Sigmoid function can be implemented through tf.nn.sigmoid function as follows:

```
In [7]:x = tf.linspace(-6.,6.,10)
x # Create input vector -6~6
Out[7]:
<tf.Tensor: id=5, shape=(10,), dtype=float32, numpy=
array([-6.        , -4.6666665, -3.3333333, -2.        , -0.6666665,
        0.666667 ,  2.        ,  3.333334 ,  4.666667 ,  6.        ]…
```

```
In [8]:tf.nn.sigmoid(x) # Pass x to Sigmoid function
Out[8]:
<tf.Tensor: id=7, shape=(10,), dtype=float32, numpy=
array([0.00247264, 0.00931597, 0.03444517, 0.11920291, 0.33924365,
      0.6607564 , 0.8807971 , 0.96555483, 0.99068403, 0.9975274 ],
     dtype=float32)>
```

As you can see, the range $[-6,6]$ of element values in the vector is mapped to the interval $(0,1)$.

6.4.2 ReLU

Before the ReLU (REctified Linear Unit) activation function was proposed, the Sigmoid function was usually the first choice for activation functions of neural networks. However, when the input value of Sigmoid function is too large or too small, the gradient value is close to 0, which is known as the gradient dispersion phenomenon. When this phenomenon occurs, the network parameters will not be updated for a long time, which leads to the phenomenon that the training does not converge. The gradient dispersion phenomenon is more likely to occur in deeper network models. The 8-layer AlexNet model proposed in 2012 uses an activation function called ReLU, which makes the number of network layers reach 8. Since then, the ReLU function has become more and more widely used. The ReLU function is defined as

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

The function curve is shown in Figure 6.8. It can be seen that ReLU suppresses all values less than 0 to 0; for positive numbers, it outputs those directly. This unilateral suppression characteristic comes from biology. In 2001, neuroscientists Dayan and Abott simulated a more accurate model of brain neuron activation, as shown in Figure 6.9. It has characteristics such as unilateral suppression and relatively loose excitation boundaries. The design of the ReLU function is very similar to it [2].

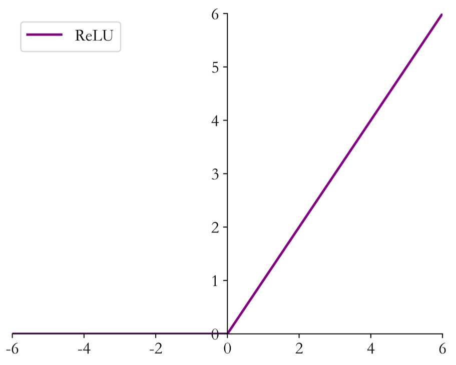


Figure 6.8 ReLU function

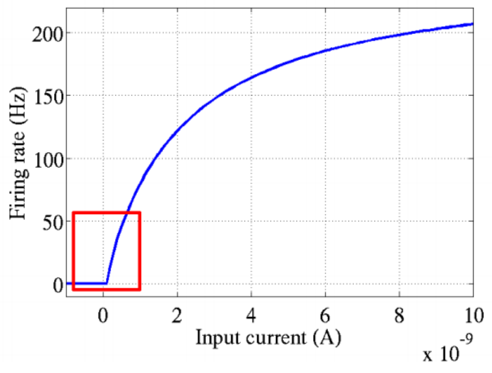


Figure 6.9 Human brain activation function [2]

In TensorFlow, the ReLU function can be implemented through tf.nn.relu function as follows:

```
In [9]:tf.nn.relu(x)
Out[9]:
<tf.Tensor: id=11, shape=(10,), dtype=float32, numpy=
array([0.       , 0.       , 0.       , 0.       , 0.       , 0.666667,
       2.       , 3.333334, 4.666667, 6.       ], dtype=float32)>
```

It can be seen that after the ReLU activation function, the negative numbers are all suppressed to 0, and the positive numbers are retained.

In addition to using the functional interface tf.nn.relu to implement the ReLU function, the ReLU function can also be added to the network as a network layer like the Dense layer. The corresponding class is layers.ReLU(). Generally speaking, the activation function class is not the main network computing layer and does not count into the number of network layers.

The design of the ReLU function is derived from neuroscience. The calculation of function values and derivative values is very simple. At the same time, it has excellent gradient characteristics. It has been verified to be very effective in a large number of deep learning applications.

6.4.3 LeakyReLU

The derivative of the ReLU function is always 0 when $x < 0$, which may also cause gradient dispersion. To overcome this problem, the LeakyReLU function (Figure 6.10) is proposed.

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

where p is a small value set by users, such as 0.02. When $p = 0$, the LeakyReLU function degenerates to the ReLU function. When $p \neq 0$, a small derivative value can be obtained at $x < 0$, thereby avoiding the phenomenon of gradient dispersion.

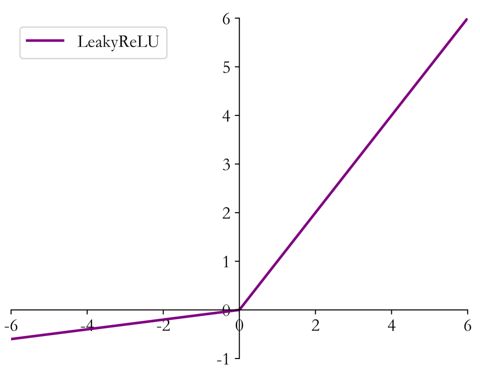


Figure 6.10 LeakyReLU function

In TensorFlow, LeakyReLU function can be implemented through tf.nn.leaky_relu as follows:

```
In [10]:tf.nn.leaky_relu(x, alpha=0.1)
```

```
Out[10]:
```

```
<tf.Tensor: id=13, shape=(10,), dtype=float32, numpy=
array([-0.6       , -0.46666667, -0.33333334, -0.2       , -0.06666666,
        0.666667  , 2.        , 3.333334  , 4.666667  , 6.       ],
     dtype=float32)>
```

The alpha parameter represents p. The corresponding class of tf.nn.leaky_relu is layers.LeakyReLU. You can create a LeakyReLU network layer through LeakyReLU(alpha) and set the parameter p. Like the Dense layer, the LeakyReLU layer can be placed in a suitable position on the network.

6.4.4 Tanh

The Tanh function can "compress" the input $x \in R$ to an interval $(-1,1)$, defined as:

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

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

It can be seen that the Tanh activation function can be realized after zooming and translated by the Sigmoid function, as shown in Figure 6.11.

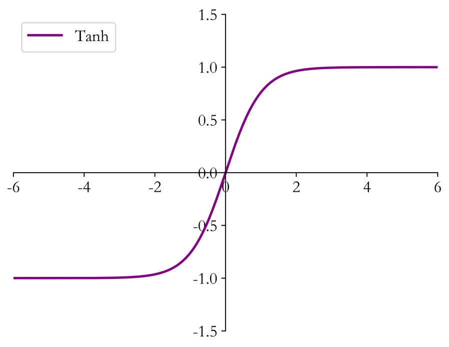


Figure 6.11 tanh function

In Tensorflow, the Tanh function can be implemented using tf.nn.tanh as follows:

```
In [11]:tf.nn.tanh(x)
```

```
Out[11]:
<tf.Tensor: id=15, shape=(10,), dtype=float32, numpy=
array([-0.9999877 , -0.99982315, -0.997458  , -0.9640276 , -0.58278286,
       0.5827831 ,  0.9640276 ,  0.997458  ,  0.99982315,  0.9999877 ],
     dtype=float32)>
```

You can see that the range of vector element values is mapped to $(-1,1)$.

6.5 Design of output layer

Let's discuss the design of the last layer of network in particular. In addition to all hidden layers, it completes the functions of dimensional transformation and feature extraction, and it is also used as an output layer. It is necessary to decide whether to use the activation function and what type of activation function to use according to the specific tasks.

We will classify the discussions based on the range of output values. Common types of output include:

- ❑ $o_i \in R^d$ The output belongs to the entire real number space, or a certain part of real number space, such as function value trend prediction and age prediction problems.
- ❑ $o_i \in [0,1]$ The output value falls in the interval $[0,1]$, such as image generation, and the pixel value of the image is generally expressed by values in interval $[0,1]$; or the probability of the binary classification problem, such as the probability of the tail or face of a coin.
- ❑ $o_i \in [0,\ 1],\ \sum_i\ o_i = 1$ The output value falls within the interval $[0,1]$, and the sum of all output values is 1. Common problems include multi-classification problems, such as MNIST handwritten digital picture recognition, which the sum of the probability that the picture belongs to 10 categories should be 1.
- ❑ $o_i \in [-1,\ 1]$ output value is between -1 and 1.

6.5.1 Common real number space

This type of problem is more common. For example, sine function curve, age prediction, and stock trend prediction all belong to the whole or part of continuous real number space, and the output layer may not have an activation function. The calculation of the error is directly based on the output $\boldsymbol{o}$ of the last layer and the true value $\boldsymbol{y}$. For example, the mean square error function is used to measure the distance between the output value $\boldsymbol{o}$ and the true value $\boldsymbol{y}$:

$$\mathcal{L} = g(\boldsymbol{o}, \boldsymbol{y})$$

where g represents a specific error calculation function, such as MSE.

6.5.2 [0, 1] interval

It is also common for output values to belong to interval $[0, 1]$, such as image generation, and binary classification problems. In machine learning, image pixel values are generally normalized to intervals $[0, 1]$. If the values of the output layer are used directly, the pixel value range will be distributed in the entire real number space. In order to map the pixel values to the effective real number space $[0,1]$, a suitable activation function needs to be added after the output layer. The Sigmoid function is a good choice here.

Similarly, for binary classification problems, such as the prediction of the face and tail of coins, the output layer can only be one node which is the probability of an event A occurring $P(\mathrm{A}|\boldsymbol{x})$ giving the network input x. If we use the output scalar o of the network to represent the probability of the occurrence of positive events, then the probability of the occurrence of negative events is $1 - o$. The network structure is shown in Figure 6.12.

$$P(Face|\boldsymbol{x}) = o$$

$$P(Tail|\boldsymbol{x}) = 1 - o$$

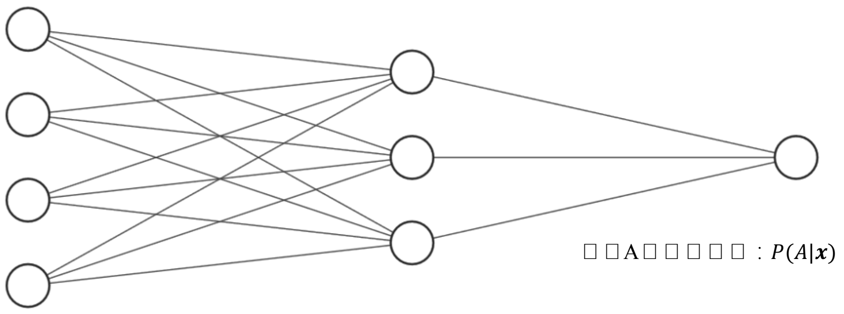


Figure 6.12 Binary classification network with single output node

In this case, you only need to add the Sigmoid function after the value of the output layer to translate the output into a probability value. For the binary classification problem, in addition to using a single output node to represent the probability of the occurrence of event A $P(\mathrm{A}|\boldsymbol{x})$, you can also separately predict $P(\mathrm{A}|\boldsymbol{x})$ and $P(\overline{\mathrm{A}}|\boldsymbol{x})$, and satisfy the constraints

$$P(\mathrm{A}|\boldsymbol{x}) + P(\overline{\mathrm{A}}|\boldsymbol{x}) = 1$$

where $\overline{\mathrm{A}}$ indicates the opposite event of event A. As shown in Figure 6.13, the output layer of the binary classification network is 2 nodes. The output value of the first node represents the probability of the occurrence of event A $P(\mathrm{A}|\boldsymbol{x})$, and the output value of the second node represents the probability of the occurrence of the opposite event $P(\bar{A}|\boldsymbol{x})$. The function can only compress a single value to the interval $(0,1)$, and does not consider the relationship between the two node values. We hope that in addition to satisfy $o_i \in [0,1]$, they can satisfy the constraint that the sum of probabilities is 1:

$$\sum_i o_i = 1$$

This situation is the problem setting to be introduced in the next section.

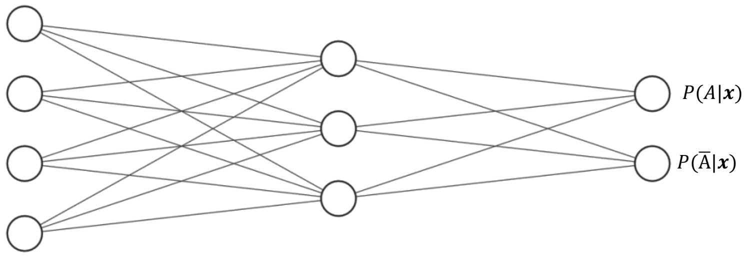


Figure 6.13 Binary classification network with two outputs

6.5.3 [0,1] interval with sum 1

For cases that the output value $o_i \in [0,1]$, and the sum of all output values is 1, it is the most common problem with multi-classification. As shown in Figure 6.15, each output node of the output layer represents a category. The network structure in the figure is used to handle 3 classification tasks. The output value distribution of the 3 nodes represents the probability that the current sample belongs to category A, B and C: $P(\mathrm{A}|\boldsymbol{x})$, $P(\mathrm{B}|\boldsymbol{x})$, and $P(\mathrm{C}|\boldsymbol{x})$. Because the sample in the multi-classification problem can only belong to one of the categories, so the sum of the probabilities of all categories should be 1.

How to implement this constraint logic? This can be achieved by adding a Softmax function to the output layer. The Softmax function is defined as

$$Softmax(z_i) \triangleq \frac{e^{z_i}}{\sum_{j=1}^{d_{\text{out}}} e^{z_j}}$$

The Softmax function can not only map the output value to the interval $[0,1]$, but also satisfy the characteristic that the sum of all output values is 1. As shown in the example in Figure 6.14, the output of the output layer is $[2.0,1.0,0.1]$. After went through the Softmax function, the output becomes $[0.7,0.2,0.1]$. Each value represents the probability that the current sample belongs to each category, and the sum of the probability values is 1. The output of the output layer can be translated into category probabilities through the Softmax function, which is used very often in classification problems.

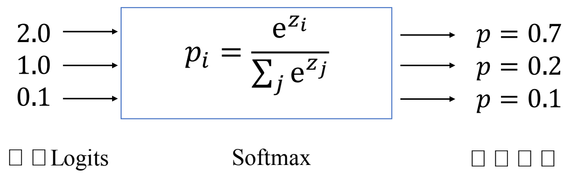
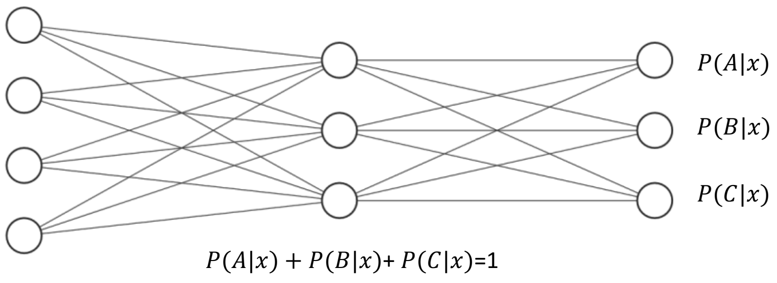


Figure 6.14 Softmax function example



Figrue 6.15 Multiclassificaiton network structure

In TensorFlow, the Softmax function can be implemented through tf.nn.softmax as follows:

```
In [12]: z = tf.constant([2.,1.,0.1])
tf.nn.softmax(z)
Out[12]:
<tf.Tensor: id=19, shape=(3,), dtype=float32, numpy=array([0.6590012,
0.242433 , 0.0985659], dtype=float32)>
```

Similar to the Dense layer, the Softmax function can also be used as a network layer class. It is convenient to add the Softmax layer through the layers.Softmax (axis = -1) class, where the axis parameter specifies the dimension to be calculated.

In the numerical calculation process of the Softmax function, the numerical overflow phenomenon is likely to occur due to the large input value. Similar problem may happen when calculating the cross entropy. For the stability of numerical calculation, TensorFlow provides a unified interface that implements Softmax and cross-entropy loss function at the same time, and also handles the anomalies of numerical instability. It is generally recommended to use these interface functions. The functional interface is tf.keras.losses.categorical_crossentropy(y_true, y_pred, from_logits = False), where y_true represents the one-hot encoded true label, y_pred represents the predicted value of the network. When from_logits is set to True, y_pred represents the variable z that has not went through the Softmax function. When from_logits is set to False, y_pred is expressed as the output of the Softmax function. For numerical calculation stability, generally set from_logits to True, so that tf.keras.losses.categorical_crossentropy will perform Softmax function calculation internally, and there is no need to explicitly call the Softmax function in the model explicitly. For example:

```
In [13]:
z = tf.random.normal([2,10]) # Create output of the output layer
y_onehot = tf.constant([1,3]) # Create real label
y_onehot = tf.one_hot(y_onehot, depth=10) # one-hot encoding
# The Softmax function is not explicitly used in output layer, so
# from_logits=True. categorical_crossentropy function will use Softmax
# function first in this case.
loss = keras.losses.categorical_crossentropy(y_onehot,z,from_logits=True)
loss = tf.reduce_mean(loss) # calculate the loss
loss
```

```
Out[13]:
<tf.Tensor: id=210, shape=(), dtype=float32, numpy= 2.4201946>
```

In addition to the functional interface, you can also use the losses.CategoricalCrossentropy(from_logits) class method to simultaneously calculate the Softmax and cross-entropy loss functions. For example:

```
In [14]:
criteon = keras.losses.CategoricalCrossentropy(from_logits=True)
loss = criteon(y_onehot,z)
loss
```

```
Out[14]:
<tf.Tensor: id=258, shape=(), dtype=float32, numpy= 2.4201946>
```

6.5.4 (-1, 1) interval

If you want the range of output values to be distributed in intervals $(-1, 1)$, you can simply use the tanh activation function:

```
In [15]:
x = tf.linspace(-6.,6.,10)
tf.tanh(x)
```

```
Out[15]:
<tf.Tensor: id=264, shape=(10,), dtype=float32, numpy=
array([-0.9999877 , -0.99982315, -0.997458  , -0.9640276 , -0.58278286,
        0.5827831 ,  0.9640276 ,  0.997458  ,  0.99982315,  0.9999877 ],
      dtype=float32)>
```

The design of the output layer has a certain flexibility, which can be designed according to the actual application scenario, and make full use of the characteristics of the existing activation function.

6.6 Error calculation

After building the model structure, the next step is to select the appropriate error function to calculate the error. Common error functions are Mean Square Error, Cross Entropy, KL

Divergence, and Hinge Loss. Among them, the Mean Square Error function and Cross Entropy function are more common in deep learning. The Mean Square Error function is mainly used for regression problems, and the Cross Entropy function is mainly used for classification problem.

6.6.1 Mean Square Error function

Mean Squared Error (MSE) function maps the output vector and the true vector to two points in the Cartesian coordinate system, by calculating the Euclidean distance between these two points (to be precise, the square of Euclidean distance) to measure the difference between the two vectors:

$$\mathrm{MSE}(\boldsymbol{y},\boldsymbol{o}) \triangleq \frac{1}{d_{\text{out}}}\sum_{i=1}^{d_{\text{out}}}(y_i - o_i)^2$$

The value of MSE is always greater than or equal to 0. When the MSE function reaches the minimum value of 0, the output is equal to the true label, and the parameters of the neural network reach the optimal state.

The MSE function is widely used in regression problems. In fact, the MSE function can also be used in classification problems. In TensorFlow, MSE calculation can be implemented in a functional or layer manner. For example, implement MSE calculation using a function as follows:

```
In [16]:
o = tf.random.normal([2,10]) # Network output
y_onehot = tf.constant([1,3]) # Real label
y_onehot = tf.one_hot(y_onehot, depth=10)
loss = keras.losses.MSE(y_onehot, o) # Calculate MSE
loss
Out[16]:
<tf.Tensor: id=27, shape=(2,), dtype=float32, numpy=array([0.779179 ,
1.6585705], dtype=float32)>
```

In particular, the MSE function returns the mean square error of each sample. You need to average again in the sample dimension to obtain the mean square error of the average sample. The implementation is as follows:

```
In [17]:
loss = tf.reduce_mean(loss)
loss
Out[17]:
<tf.Tensor: id=30, shape=(), dtype=float32, numpy=1.2188747>
```

It can also be implemented in layer mode. The corresponding class is keras.losses.MeanSquaredError(). Like other classes, the __call__ function can be called to complete the forward calculation. The code is as follows:

```
In [18]:
criteon = keras.losses.MeanSquaredError()
loss = criteon(y_onehot,o)
```

```
loss
Out[18]:
<tf.Tensor: id=54, shape=(), dtype=float32, numpy=1.2188747>
```

6.6.2 Cross entropy error function

Before introducing the cross-entropy loss function, we first introduce the concept of entropy in informatics. In 1948, Claude Shannon introduced the concept of entropy in thermodynamics into information theory to measure the uncertainty of information. Entropy is also called information entropy or Shannon entropy in information science. The greater the entropy, the greater the uncertainty and the greater the amount of information. The entropy of a distribution $P(i)$ is defined as

$$H(P) \triangleq -\sum_i P(i)\log_2 P(i)$$

In fact, other base functions can also be used. For example, for the 4-category classification problem, if the true label of a sample is category 4, then the one-hot encoding of the label is $[0,0,0,1]$. That is, the classification of this picture is uniquely determined, and it belongs to category 4 with uncertainty 0, and its entropy can be simply calculated as:

$$-0\cdot\log_2 0-0\cdot\log_2 0-0\cdot\log_2 0-1\cdot\log_2 1=0$$

That is to say, for a certain distribution, the entropy is 0 and the uncertainty is the lowest.

If the predicted probability distribution is $[0.1,0.1,0.1,0.7]$, its entropy can be calculated as:

$$-0.1\cdot\log_2 0.1-0.1\cdot\log_2 0.1-0.1\cdot\log_2 0.1-0.7\cdot\log_2 0.7\approx 1.356$$

Considering a random classifier, its prediction probability for each category is equal: $[0.25,0.25,0.25,0.25]$. In the same way, its entropy can be calculated to be about 2, and the uncertainty in this case is slightly larger than the above case.

Because, the entropy is always greater than or equal to 0. When the entropy reaches a minimum value of 0, the uncertainty is 0. The distribution of one-hot coding for classification problems is a typical example of entropy of 0. In TensorFlow, we can use tf.math.log to calculate entropy.

After introducing the concept of entropy, we'll derive the definition of Cross Entropy based on entropy:

$$H(p||q) \triangleq -\sum_i p(i)\log_2 q(i)$$

Through transformation, cross entropy can be decomposed into the sum of entropy and KL divergence (Kullback-Leibler Divergence):

$$H(p||q)=H(p)+D_{KL}(p||q)$$

where KL divergence is

$$D_{KL}(p||q) = \sum_i p(i)\log\left(\frac{p(i)}{q(i)}\right)$$

KL divergence is an indicator used by Solomon Kullback and Richard A. Leibler in 1951 to measure the distance between two distributions. When $p = q$, the minimum value of $D_{KL}(p||q)$ is 0. The greater the difference between p and q, the greater $D_{KL}(p||q)$ is. It should be noted that neither the cross entropy nor the KL divergence are symmetrical, namely:

$$H(p||q) \neq H(q||p)$$

$$D_{KL}(p||q) \neq D_{KL}(q||p)$$

Cross entropy is a good measure of the "distance" between two distributions. In particular, when the distribution of y in the classification problem uses one-hot coding, $H(p) = 0$. Then,

$$H(p||q) = H(p) + D_{KL}(p||q) = D_{KL}(p||q)$$

That is, cross entropy degenerate to the KL divergence between the true label distribution and the output probability distribution.

According to the definition of KL divergence, we derive the calculation expression of cross entropy in the classification problem:

$$H(p||q) = D_{KL}(p||q) = \sum_j y_j\log\left(\frac{y_j}{o_j}\right)$$

$$= 1 \cdot \log\frac{1}{o_i} + \sum_{j\neq i} 0 \cdot \log\left(\frac{0}{o_j}\right)$$

$$= -\log o_i$$

where i is the index number of 1 in the one-hot encoding, which is also the real category. It can be seen that the cross entropy is only related to the probability on the real category o_i, and the larger the corresponding probability o_i, the smaller $H(p||q)$ is. When the probability on the corresponding category is 1, the cross-entropy achieves the minimum value of 0. At this time, the network output is completely consistent with the real label, and the neural network obtains the optimal state.

Therefore, the process of minimizing the cross-entropy loss function is also the process of maximizing the prediction probability of the correct category. From this perspective, understanding the cross-entropy loss function is very intuitive and easy.

6.7 Types of neural networks

The fully connected layer is the most basic type of neural network, and it has made a tremendous contribution to the subsequent research of neural networks. The forward calculation process of the fully connected layer is relatively simple, and the gradient derivation is also relatively simple, but it has one of the biggest defects. When processing data with a large feature length, the parameter amount of the fully connected layer is often large, making the number of parameters of the fully connected network huge and difficult to train. In recent years, the development of social media has produced a large number of digital resources such as pictures, videos, and texts, which has greatly promoted the research of neural networks in the fields of computer vision and natural language processing, and has successively proposed a series of neural network types.

6.7.1 Convolutional Neural Network

How to identify, analyze, and understand data such as pictures and videos is a core problem of computer vision. When the fully connected layer processes high-dimensional pictures and video data, it often has problems such as huge network parameters and very difficult to train. By using the idea of local correlation and weight sharing, Yann Lecun proposed Convolutional Neural Network (CNN) in 1986. With the prosperity of deep learning, the performance of convolutional neural networks in computer vision has greatly surpassed other algorithms, showing a tendency to dominate the field of computer vision. Popular models for image classification include AlexNet, VGG, GoogLeNet, ResNet, and DenseNet. For objective recognition, there are RCNN, Fast RCNN, Faster RCNN, Mask RCNN, YOLO, and SSD. We will introduce the principles of convolutional neural networks in detail in Chapter 10.

6.7.2 Recurrent neural network

In addition to data such as pictures and videos with spatial structure, sequence signals are also a very common type of data. One of the most representative sequence signals is text. How to process and understand text data is a core issue of natural language processing. Convolutional neural networks are not good at processing sequence signals due to the lack of memory mechanism and the ability to process signals of indefinite length. Recurrent Neural Network (RNN), under continuous research by Yoshua Bengio, Jürgen Schmidhuber and others, is proved to be very good at processing sequence signals. In 1997, Jürgen Schmidhuber proposed the LSTM network. As a variant of RNN, it better overcomes the problems of RNN that lacks long-term memory and is not good at processing long sequences. LSTM has been widely used in natural language processing. Based on the LSTM model, Google has proposed the Seq2Seq model for machine translation, and it has been successfully used in the Google Neural Machine Translation System (GNMT). Other RNN variants include GRU and bidirectional RNN. We will introduce the principles of recurrent neural networks in detail in Chapter 11.

6.7.3 Attention (mechanism) network

RNN is not the ultimate solution for natural language processing. In recent years, with the attention mechanism proposed, it overcomes the deficiencies of RNN such as training instability and difficulty in parallelization. It has gradually emerged in the fields of natural language processing and image generation. The attention mechanism was originally proposed on the image classification task, but gradually began to become more effective in natural language processing. In 2017, Google proposed the first network model Transformer using a pure attention mechanism, and then based on the Transformer model, a series of attention network models for machine translation, such as GPT, BERT, and GPT-2, were successively proposed. In other fields, the network based on the attention mechanism, especially the self-attention mechanism has also achieved good results, such as the BigGAN model.

6.7.4 Graph convolutional neural network

Data such as pictures and texts that have a regular spatial or temporal structure called Euclidean Data. Convolutional neural networks and recurrent neural networks are very good at handling this type of data. For data like a series of irregular spatial topologies, social networks, communication networks, and protein molecular structures, those networks seem to be powerless. In 2016, Thomas Kipf et al. proposed a graph convolution network (GCN) model based on the first-order approximate spectral convolution algorithm. The GCN algorithm is simple to implement and can be intuitively understood from the perspective of spatial first-order neighbor information aggregation, and therefore has achieved good results on semi-supervised tasks. Subsequently, a series of network models have been proposed, such as GAT, EdgeConv, and DeepGCN.

6.8 Hands-on of automobile fuel consumption prediction

In this section, we will use the fully-connected network model to complete the prediction of MPG (Mile Per Gallon) of the car.

6.8.1 Dataset

We use the Auto MPG data set, which include the real data of various vehicle performance indicators and other factors such as the number of cylinders, weight, and horsepower. The first 5 items of the data set is shown in Table 6.1. In addition to the numeric field of origin indicates the category, the other fields are numeric types. For the place of origin, 1 indicates the United States, 2 indicates Europe, and 3 indicates Japan.

Table 6.1 First 5 items of the Auto MPG dataset

MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	Origin
18.0	8	307.0	130.0	3504.0	12.0	70	1
15.0	8	350.0	165.0	3693.0	11.5	70	1
18.0	8	318.0	150.0	3436.0	11.0	70	1
16.0	8	304.0	150.0	3433.0	12.0	70	1
17.0	8	302.0	140.0	3449.0	10.5	70	1

The Auto MPG dataset includes a total of 398 records. We download and read the data set from the UCI server to a DataFrame object. The code is as follows:

```
# Download the dataset online
```

```
dataset_path = keras.utils.get_file("auto-mpg.data",
"http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-
mpg.data")
# Use Pandas library to read the dataset
column_names = ['MPG','Cylinders','Displacement','Horsepower','Weight',
             'Acceleration', 'Model Year', 'Origin']
raw_dataset = pd.read_csv(dataset_path, names=column_names,
                  na_values = "?", comment='\t',
                  sep=" ", skipinitialspace=True)
dataset = raw_dataset.copy()
# Show some data
dataset.head()
```

The data in the original table may contain missing values. These record items need to be cleared:

```
dataset.isna().sum() # Calculate the number of missing values
dataset = dataset.dropna() # Drop missing value records
dataset.isna().sum() # Calculate the number of missing values again
```

After clearing, the data set record items were reduced to 392 items.

Since the Origin field is categorical data, we first remove it and then convert it into three new fields: USA, Europe, and Japan, which represent whether they are from this origin:

```
origin = dataset.pop('Origin')
dataset['USA'] = (origin == 1)*1.0
dataset['Europe'] = (origin == 2)*1.0
dataset['Japan'] = (origin == 3)*1.0
dataset.tail()
```

Split the data into training (80%) and testing (20%) datasets:

```
train_dataset = dataset.sample(frac=0.8,random_state=0)
test_dataset = dataset.drop(train_dataset.index)
```

Move MPG out and use it real label:

```
train_labels = train_dataset.pop('MPG')
test_labels = test_dataset.pop('MPG')
```

Calculate the mean and standard deviation of each field value of the training set, and complete the standardization of the data, through the norm() function, the code is as follows:

```
train_stats = train_dataset.describe()
train_stats.pop("MPG")
train_stats = train_stats.transpose()
# Normalize the data
def norm(x): # minus mean and divide by std
  return (x - train_stats['mean']) / train_stats['std']
normed_train_data = norm(train_dataset)
```

```
normed_test_data = norm(test_dataset)
```

Print the shape of training and testing datasets:

```
print(normed_train_data.shape,train_labels.shape)
print(normed_test_data.shape, test_labels.shape)
(314, 9) (314,) # 314 records in training dataset with 9 features.
(78, 9) (78,) # 78 records in training dataset with 9 features.
```

Create TensorFlow dataset:

```
train_db = tf.data.Dataset.from_tensor_slices((normed_train_data.values,
train_labels.values))
train_db = train_db.shuffle(100).batch(32) # Shuffle and batch
```

We can observe the influence of each field on MPG by simply observing the distribution between each field in the data set, as shown in Figure 6.16. It can be roughly observed that the relationship between car displacement, weight and MPG is relatively simple. As the displacement or weight increases, the MPG of the car decreases and the energy consumption increases; the smaller the number of cylinders, the better MPG can be, which is in line with our life experience.

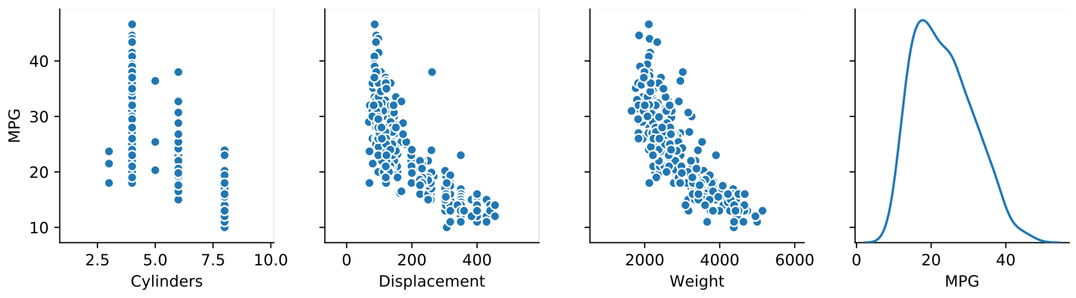


Figure 6.16 Relations between features

6.8.2 Create a network

Considering the small size of the Auto MPG data set, we only create a 3-layer fully connected network to complete the MPG prediction task. There are 9 input features, so the number of input nodes in the first layer is 9. The number of output nodes of the first layer and the second layer is designed as 64 and 64. Since there is only one kind of prediction value, the output node of the output layer is designed as 1. Because MPG belong to the real number space, the activation function of the output layer may not be added.

We implement the network as a custom network class. We only need to create each sub-network layer in the initialization function, and implement the calculation logic of the custom network class in the forward calculation function. The custom network class inherits from the keras.Model class, which is also the standard writing method of the custom network class, in order to conveniently use the various convenient functions such as trainable_variables and save_weights provided by the keras.Model class. The network model class is implemented as follows:

```
class Network(keras.Model):
    # regression network
    def __init__(self):
        super(Network, self).__init__()
        # create 3 fully-connected layers
        self.fc1 = layers.Dense(64, activation='relu')
        self.fc2 = layers.Dense(64, activation='relu')
        self.fc3 = layers.Dense(1)

    def call(self, inputs, training=None, mask=None):
        # pass through the 3 layers sequentially
        x = self.fc1(inputs)
        x = self.fc2(x)
        x = self.fc3(x)

        return x
```

6.8.3 Training and testing

After the creation of the main network model class, let's instantiate the network object and create the optimizer as follows:

```
model = Network() # Instantiate the network
# Build the model with 4 batch and 9 features
model.build(input_shape=(4, 9))
model.summary() # Print the network
# Create the optimizer with learning rate 0.001
optimizer = tf.keras.optimizers.RMSprop(0.001)
```

Next, implement the network training part. Through the double-layer loop training network composed of Epoch and Step, a total of 200 Epochs are trained.

```
for epoch in range(200): # 200 Epoch
    for step, (x,y) in enumerate(train_db): # Loop through training set once
        # Set gradient tape
        with tf.GradientTape() as tape:
            out = model(x) # Get network output
            loss = tf.reduce_mean(losses.MSE(y, out)) # Calculate MSE
            mae_loss = tf.reduce_mean(losses.MAE(y, out)) # Calculate MAE

        if step % 10 == 0: # Print training loss every 10 steps
            print(epoch, step, float(loss))
        # Calculate and update gradients
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

For regression problems, in addition to the mean square error (MSE), the mean absolute error (MAE) can also be used to measure the performance of the model.

$$\text{mae} \triangleq \frac{1}{d_{\text{out}}} \sum_i |y_i - o_i|$$

We can record the MAE at the end of each Epoch for the training and testing data set, and draw the change curve as shown in Figure 6.17.

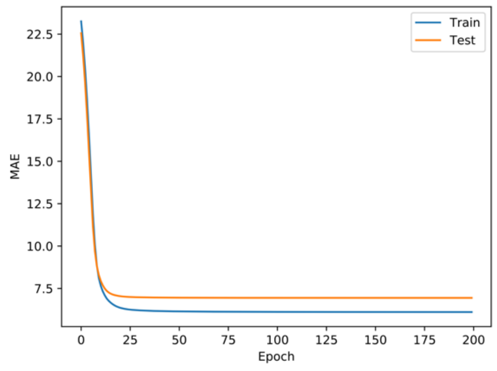


Figure 6.17 MAE curve

It can be seen that when training reaches about the 25th Epoch, the decline of MAE becomes slower, in which the MAE of the training set continues to decline slowly, but the MAE of the test set remains almost unchanged, so we can end the training around the 25th epoch, and use the network parameters at this time to predict new input.

6.9 References

[1] Nick, 2017. A brief history of aritificial intelligence.

[2] X. Glorot, A. Bordes and Y. Bengio, “Deep Sparse Rectifier Neural Networks,” *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, Fort Lauderdale, FL, USA, 2011.

[3] J. Mizera-Pietraszko and P. Pichappan, Lecture Notes in Real-Time Intelligent Systems, Springer International Publishing, 2017.