Chapter 3 Classification

A year spent in artificial intelligence is enough to make one believe in God.

– Alan Perlis

The linear regression model for continuous variable prediction has been introduced previously. Now let's dive into the classification problem. A typical application of the classification problem is to teach computers how to automatically recognize objects in images. Let's consider one of the simplest tasks in image classification: 0-9 digital picture recognition, which is relatively simple and also has a very wide range of applications, such as postal codes, courier numbers and mobile phone numbers recognition. We will take 0-9 digital picture recognition as an example to explore how to use machine learning to solve the classification problem.

3.1 Handwritten digit picture dataset

Machine learning needs to learn from the data, so it first needs to collect a large amount of real data. Taking handwritten digit picture recognition as an example, as shown in Figure 0.1, we need to collect a large number of 0-9 digit pictures written by real people. In order to facilitate storage and calculation, the collected pictures are generally scaled to a fixed size, such as a 224×224 or 96×96 pixels. These pictures will be used as the input data $\boldsymbol{x}$. At the same time, we need to label each image, which will be used as the real value of the image. This label indicates which specific category the image belongs to. For handwritten digit picture recognition, the label is numbers 0-9 to represent pictures with category names 0-9.

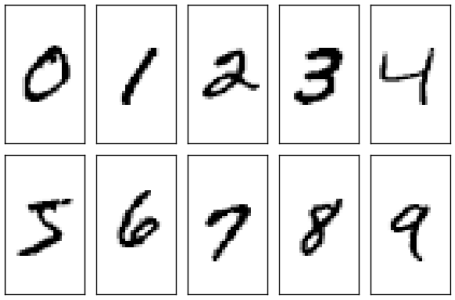


Figure 0.1 Handwritten digit pictures

If we want the model to perform well on new samples, i.e. achieve good model generalization ability, then we need to increase the size and diversity of the data set as much as possible, so that the training data set is as close as possible to the real population distribution. And after training, the model can also perform well on unseen samples.

In order to facilitate algorithm evaluation, Lecun et al. [1] released a handwritten digital picture data set named MNIST, which contains real handwritten pictures of numbers 0 to 9. Each

number has a total of 7000 pictures, collected from different writing styles. The total number of pictures is 70,000. Among them, 60,000 pictures are used for training, and the remaining 10,000 pictures are used as a test set.

Because the information in handwritten digital pictures is relatively simple, each picture is scaled to the same size 28×28 pixels while retaining only grayscale information, as shown in Figure 0.2. These pictures are written by real people, including rich information such as font size, writing style, and line thickness to ensure that the distribution of these pictures is as close as possible to the population distribution of real handwritten digital pictures, thereby ensuring model generalization ability.

Figure 0.2 MNIST dataset examples

Now let's look at the representation of a picture. A picture contains h rows and w columns, and pixel values are stored at each position. Generally, pixel values are integers ranging from 0 to 255 to express color intensity information. For example, 0 represents the lowest intensity, and 255 indicates the highest intensity. If it is a color picture, each pixel contains the intensity information of the three channels R, G, and B, which respectively represent the color intensity of the red channel, the green channel, and the blue channel. Therefore, unlike a grayscale image, each pixel of a color picture is represented by a one-dimensional vector with three elements, which represent the intensity of R, G, and B color. As a result, a color image is saved as a tensor with dimension [h, w, 3], while a grayscale picture only needs a two-dimensional matrix with shape [h, w] or a three-dimensional tensor with shape [h, w, 1] to represent its information. Figure 0.3 shows the matrix content of a picture 8. It can be seen that the black pixels in the picture are represented by 0 and the grayscale information is represented by 0 ~ 255. The whiter pixels in the picture correspond to the larger values in the matrix.

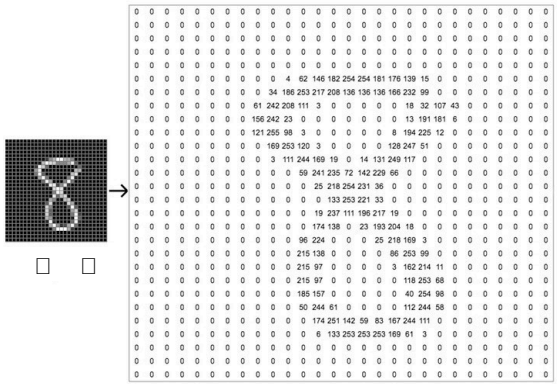


Figure 0.3 How a picture is represented[①]

Deep learning frameworks like TensorFlow and PyTorch can easily download, manage, and load the MNIST dataset through a few lines of code. Here we use TensorFlow to automatically download the MNIST dataset and convert it to a Numpy array format.

```
import os
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, optimizers, datasets
# load MNIST dataset
(x, y), (x_val, y_val) = datasets.mnist.load_data()
# convert to float type and rescale to [-1, 1]
x = 2*tf.convert_to_tensor(x, dtype=tf.float32)/255.-1
# convert to integer tensor
y = tf.convert_to_tensor(y, dtype=tf.int32)
# one-hot encoding
y = tf.one_hot(y, depth=10)
print(x.shape, y.shape)
# create training dataset
train_dataset = tf.data.Dataset.from_tensor_slices((x, y))
# train in batch
train_dataset = train_dataset.batch(512)
```

The load_data () function returns two tuple objects, the first is the training set and the second is the test set. The first element of each tuple is the training picture data $\boldsymbol{X}$, and the second element is the corresponding category number $\boldsymbol{Y}$. The training set $\boldsymbol{X}$ contains 60,000 grayscale images, and each images consists of 28x28 pixels, so its dimension is $(60000,28,28)$. The size of $\boldsymbol{Y}$ is (60,000), representing the 60,000 digital numbers ranging from 0 to 9. Similarly, the test set

[①] Data source: https://towardsdatascience.com/how-to-teach-a-computer-to-see-with-convolutional-neural-networks-96c120827cd1

contains 10,000 test pictures and corresponding digital number with dimensions $(10000,28,28)$ and (10,000) separately.

The MNIST dataset loaded from TensorFlow contains images with values from 0 to 255. In machine learning, it is generally desired that the range of data is distributed in a small range around 0. Therefore, we rescale the pixel range to interval $[-1,1]$, which will benefit the model process.

The calculation process of each picture is universal. Therefore, we can calculate multiple pictures at once, making full use of the parallel computing power of CPU or GPU. We use a matrix of shape $[h,w]$ to represent a picture. For multiple pictures, we can add one more dimension in front and use a tensor of shape $[b,h,w]$ to represent them. Here b represents the batch size. Color pictures can be represented by a tensor with the shape of $[b,h,w,c]$, where *c* represents the number of channels, which is 3 for color pictures. TensorFlow's Dataset object can be used to conveniently convert a dataset into batches using the batch() function.

3.2 Build a model

Recall the biological neuron structure we discussed in the regression chapter. We reduce the input vector $\boldsymbol{x} = [x_1, x_2, \dots, x_{d_{\text{in}}}]^T$ to a single input scalar x, and the model can be expressed as $y = xw + b$. If it is a multi-input, single-output model structure, we need to use the vector form:

$$y = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b = [w_1, w_2, w_3, \dots, w_{d_{in}}] \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{d_{in}} \end{bmatrix} + b$$

More generally, by combining multiple multi-input, single-output neuron models, we can build a multi-input, multi-output model:

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

where $\boldsymbol{x} \in R^{d_{\text{in}}}$, $\boldsymbol{b} \in R^{d_{\text{out}}}$, $\boldsymbol{y} \in R^{d_{\text{out}}}$, $\boldsymbol{W} \in R^{d_{\text{out}} \times d_{\text{in}}}$.

For multiple output and batch training, we write the model in batch form:

$$\boldsymbol{Y} = \boldsymbol{X}@\boldsymbol{W} + \boldsymbol{b} \tag{0.1}$$

where $\boldsymbol{X} \in R^{b \times d_{\text{in}}}$, $\boldsymbol{b} \in R^{d_{\text{out}}}$, $\boldsymbol{Y} \in R^{b \times d_{\text{out}}}$, $\boldsymbol{W} \in R^{d_{\text{in}} \times d_{\text{out}}}$, d_{in} represents input dimension, d_{out} indicates output dimension. $\boldsymbol{X}$ has shape $[b, d_{\text{in}}]$, where b is the number of samples, d_{in} is the length of each sample. $\boldsymbol{W}$ has shape $[d_{\text{in}}, d_{\text{out}}]$, containing $d_{\text{in}} * d_{\text{out}}$ parameters. Bias vector $\boldsymbol{b}$ has shape d_{out}. The @ symbol means matrix multiplication. Since the result of the operation $\boldsymbol{X}@\boldsymbol{W}$ is a matrix of shape $[b, d_{\text{out}}]$, it cannot be directly added to the vector $\boldsymbol{b}$. Therefore, the + sign in batch form needs to support broadcasting, i.e., expand the vector $\boldsymbol{b}$ into a matrix of shape $[b, d_{\text{out}}]$ by replicating $\boldsymbol{b}$.

Consider two samples with $d_{\text{in}} = 3$ and $d_{\text{out}} = 2$, Equation (0.1) is expanded as follow:

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

where superscripts like (1) and (2) represent the sample index and subscripts such as 1 and 2 indicate the elements of a certain sample vector. The corresponding model structure is shown in Figure 0.4.

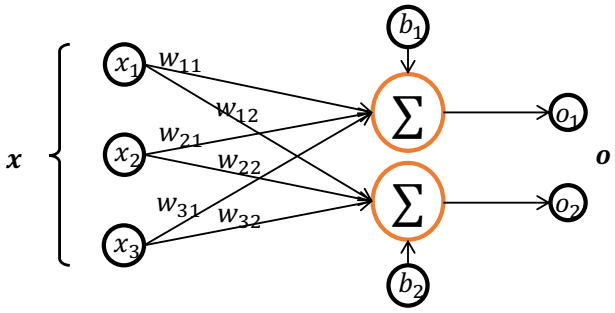


Figure 0.4 A neural network with 3 inputs and 2 outputs

It can be seen that the matrix form is more concise and clear, and at the same time, the parallel acceleration capability of matrix calculation can be fully utilized. So how to transform the input and output of the image recognition task into a tensor form?

A grayscale image is stored using a matrix with shape $[h, w]$, and b pictures are stored using a tensor with shape $[b, h, w]$. However, our model can only accept vectors, so we need to flatten the $[h, w]$ matrix into a vector of length $[h \cdot w]$, as shown in Figure 0.5, where the length of the input features $d_{\text{in}} = h \cdot w$.

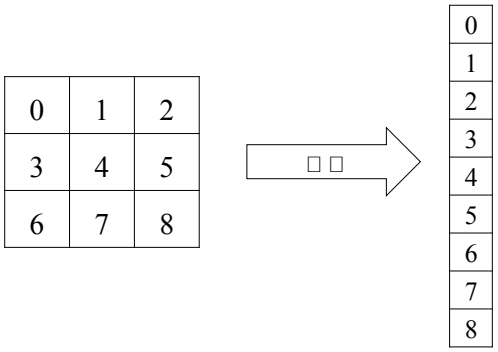


Figure 0.5 Flatten a matrix

For the output label y, the digital coding has been introduced previously. It can use a number to represent the label information. The output only needs one number to represent the predicted category value of the network, such as number 1 for cats and number 3 for fish. However, one of the biggest problems with digital coding is that there is a natural size relationship between numbers. For example, if the tags corresponding to 1, 2, and 3 are cats, dogs, and fish, there is no size relationship between them, but $1 < 2 < 3$. Therefore, if digital coding is used, it will force the model to learn this unnecessary constraint.

So how to solve this problem? The output actually can be set to a set of vectors with length d_{out}, where d_{out} is the same as the number of categories. For example, if the output belongs to the first category, then the index is set to 1 and the other positions are set to 0. The encoding method is called one-hot encoding. Taking the "cat, dog, fish, and bird" recognition system in Figure 0.6 as an example, all the samples belong to only one of the four categories of "cat, dog,

fish, and bird". We use the index positions to indicate the category of cats, dogs, and fish, respectively. For all pictures of cats, their one-hot encoding is $[1,0,0,0]$; for all dog pictures, their one-hot encoding is $[0,1,0,0]$, and so on. One-hot encoding is widely used in classification problems.

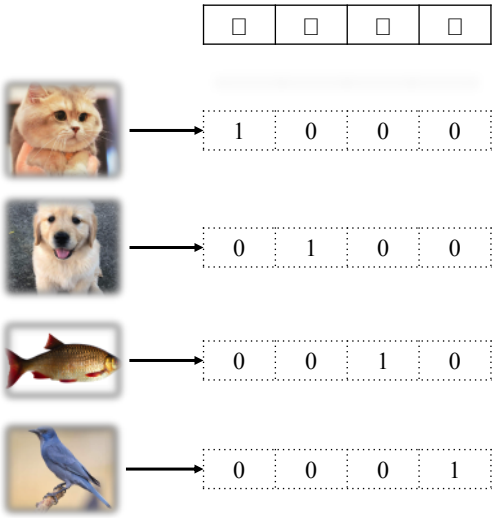


Figure 0.6 One-hot coding example

The total number of categories of handwritten digital pictures is 10, i.e. $d_{\text{out}} = 10$. For a sample, suppose it belongs to a category i, i.e. number i, using one-hot encoding, we can represent it using a vector $\boldsymbol{y}$ with length 10, where the ith element in this vector is 1 and the rest is 0. For example, the one-hot encoding of picture 0 is $[1,0,0,\ldots,0]$, and the one-hot encoding of picture 2 is $[0,0,1,\ldots,0]$, and the one-hot encoding of picture 9 is $[0,0,0,\ldots,1]$. One-hot encoding is very sparse. Compared to digital encoding, it needs more storage, so digital encoding is generally used for storage. During calculation, digital encoding is converted to one-hot encoding which can be achieved through the tf.one_hot() function as follow.

```
y = tf.constant([0,1,2,3]) # digits 0-3
y = tf.one_hot(y, depth=10) # one-hot encoding with length 10
print(y)
Out[1]:
tf.Tensor(
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  # one-hot encoding of number 0
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]  # one-hot encoding of number 1
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]  # one-hot encoding of number 2
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]], shape=(4, 10), dtype=float32)
```

Now let's return to the task of handwritten digital picture recognition. The input is a flattened picture vector $\boldsymbol{x} \in R^{784}$, and the output is a vector of length 10 $\boldsymbol{o} \in R^{10}$ corresponding one-hot encoding of a certain number, which forms a multi-input, multi-output linear model $\boldsymbol{o} = \boldsymbol{W}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{b}$. We hope that the model output is closer to the real label.

3.3 Error calculation

For classification problems, our goal is to maximize a certain performance metric, such as accuracy. But when we use accuracy as a loss function to optimize, we will find that it is in fact indifferentiable, and we cannot use gradient descent algorithm to optimize the model parameters. The general approach is to establish a smooth and derivable proxy objective function, such as optimizing the distance between the output of the model and the one-hot encoded real label. The model obtained by optimizing the proxy objective function generally also perform well on testing dataset. Compared to the regression problem, the optimization objective function and the evaluation objective function of the classification problem are inconsistent. The goal of training a model is to find the optimal numerical solution $\boldsymbol{W}^*$ and $\boldsymbol{b}^*$ by optimizing the loss function $\mathcal{L}$:

$$\boldsymbol{W}^*, \boldsymbol{b}^* = \underbrace{\arg\min}_{\boldsymbol{W},\boldsymbol{b}} \mathcal{L}(\boldsymbol{o}, \boldsymbol{y})$$

For the error calculation of classification problem, it is more common to use the cross entropy loss function instead of the mean square error loss function introduced in the regression problem. We will introduce the cross-entropy loss function in the future chapters. Here we still use the mean squared loss function to solve the handwritten digit recognition problem for simiplity. The mean square error loss function for n samples can be expressed as:

$$\mathcal{L}(\boldsymbol{o}, \boldsymbol{y}) = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{10}\left(o_j^{(i)} - y_j^{(i)}\right)^2$$

Now we only need to use the gradient descent algorithm to optimize loss function to get the optimal solution $\boldsymbol{W}$ and $\boldsymbol{b}$, and then use the obtained model to predict the unknown handwritten digital pictures $\boldsymbol{x} \in \mathbb{D}^{\text{test}}$。.

3.4 Do we really solve the problem?

According to the above solution, is the problem of handwritten digital picture recognition really solved perfectly? At present, there are at least two major problems:

- **Linear model** Linear model is one of the simplest models in machine learning. It has only a few parameters and simple calculation, but it can only express linear relationships. Even if it is as simple as a digital picture recognition task, it also belongs to the category of image recognition. The perception and decision-making of complex brains is far more complex than a linear model. Therefore, the linear model is clearly not enough.
- **Complexity** Complexity is the model ability to approximate complex distributions. The above solution only uses a one-layer neural network model composed of a small number of neurons. Compared to the 100 billion-level neuron interconnection structure in the human brain, its expression ability is obviously weaker.

Figure 0.7 shows an example of model complexity and data distribution. The distribution of sampling points with observation errors is plotted in the figure. The actual distribution may be a quadratic parabolic model. As shown in Figure 0.7 (a), if you use a linear model to fit the data, it is difficult to learn a good model; if you use a suitable polynomial function model to learn, such as a

quadratic polynomial, you can learn a suitable model as shown in Figure 0.7 (b); but when the model is too complex, such as a 10-degree polynomial, it is likely to overfit and hurt the generalization ability of the model, as shown in Figure 0.7 (c)).

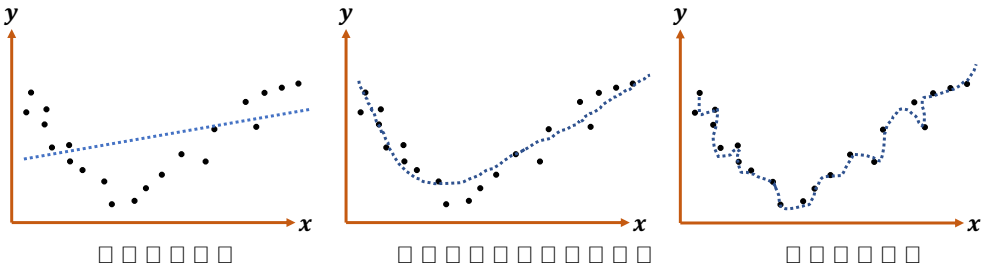


Figure 0.7 Model complexity

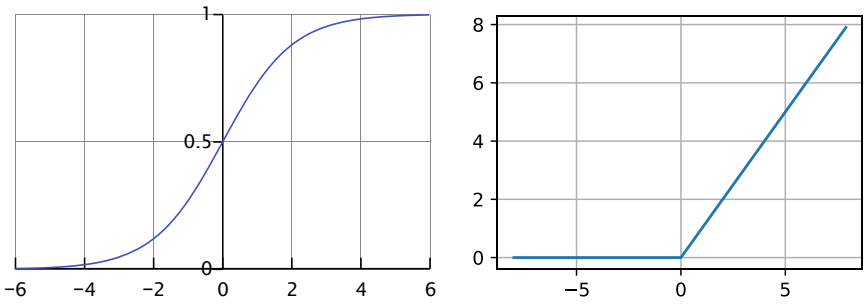
The multi-neuronal model we currently use is still a linear model with weak expression ability. Next, we try to solve these two problems.

3.5 Nonlinear model

Since a linear model is not feasible, we can embed a nonlinear function in the linear model and convert it to a nonlinear model. We call this nonlinear function the Activation Function, which is represented by σ:

$$\boldsymbol{o} = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

Here σ represents a specific nonlinear activation function, such as the Sigmoid function (Figure 0.8 (a)) and the ReLU function (Figure 0.8 (b)).



a Sigmoid□ □ b ReLU□ □

Figure 0.8 Common activation functions

The ReLU function only retains the positive part of function $y = x$, and set the negative part to be zeros. It has a unilateral suppression characteristic. Although simple, the ReLU function has excellent nonlinear characteristics, easy gradient calculation, and stable training process. It is one of the most widely used activation functions for deep learning models. Here we convert the model to a nonlinear model by embedding the ReLU function:

$$\boldsymbol{o} = \text{ReLU}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

3.6 Model complexity

To increase the model complexity, we can repeatedly stack multiple transformations such as:

$$\boldsymbol{h}_1 = \text{ReLU}(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)$$

$$\boldsymbol{h}_2 = \text{ReLU}(\boldsymbol{W}_2\boldsymbol{h}_1 + \boldsymbol{b}_2)$$

$$\boldsymbol{o} = \boldsymbol{W}_3\boldsymbol{h}_2 + \boldsymbol{b}_3$$

In the above equations, we take the output value $\boldsymbol{h}_1$ of the first layer neuron as the input of the second layer neuron model, and then take the output $\boldsymbol{h}_2$ of the second layer neuron as the input of the third layer neuron, and the output of the last layer neuron is the model output.

As shown in Figure 0.9, the function embedding appears as the connected network one after the other. We call the layer where the input node $\boldsymbol{x}$ is located the input layer. The output of each nonlinear module $\boldsymbol{h}_i$ along with its parameters $\boldsymbol{W}_i$ and $\boldsymbol{b}_i$ is called a network layer. In particular, the layer in the middle of the network is called the hidden layer and the last layer is called the output layer. This network structure formed by the connection of a large number of neurons is called a neural network. The number of nodes in each layer and the number of layers determine the complexity of the neural network.

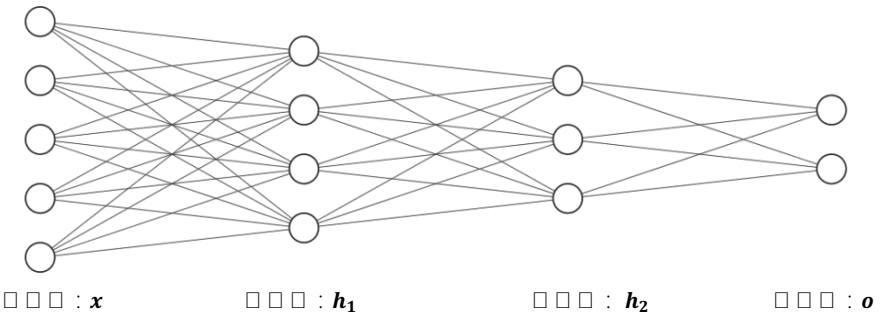


Figure 0.9 3 layer neural network architecture

Now our network model has been upgraded to a 3-layer neural network, which has a descent complexity and good nonlinear expression. Next, let's discuss how to optimize the network parameters.

3.7 Optimization method

For a network model with only one layer, we can directly derive the partial derivative expression of $\frac{\partial \mathcal{L}}{\partial w}$ and $\frac{\partial \mathcal{L}}{\partial b}$, and then calculate the gradient for each step, and update the parameters w and b using the gradient descent algorithm. However, as complex nonlinear functions are embedded, the number of network layers and the length of data features increases, the model becomes very complicated, and it is difficult to manually derive the gradient expressions. Besides, once the network structure changes, the model function and corresponding gradient expressions also change, therefore, it is obviously not feasible to rely on the manual calculation of the gradient.

That is why we have the invention of deep learning frameworks. With the help of auto

differentiation technology, deep learning frameworks can build the neural network's computational graph during the calculation of each layer's output and corresponding loss function, and then automatically calculate the gradient $\frac{\partial \mathcal{L}}{\partial \theta}$ of any parameter θ. Users only need to set up the network structure, and the gradient will automatically be calculated and updated, which is very convenient and efficient to use.

3.8 Hands-on handwritten digital image recognition

In this section, we will take everyone to experience the fun of neural networks without introducing too much details of TensorFlow. The main purpose of this section is not to teach every detail, but to give readers a comprehensive and intuitive experience of neural network algorithms. Let's start experiencing the magical image recognition algorithm!

3.8.1 Build the network

For the first layer, the input is $\boldsymbol{x} \in R^{784}$ and output $\boldsymbol{h}_1 \in R^{256}$ is a vector of length 256. We don't need to explicitly write the calculation logic of $\boldsymbol{h}_1 = \text{ReLU}(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)$. It can be achieved in TensorFlow with a single line of code:

```
# Create one layer with 256 output dimension and ReLU activation funciton
layers.Dense(256, activation='relu')
```

Using TensorFlow's Sequential function can easily build a multilayer network. For a 3-layer network, we can implement as follows:

```
# Build a 3-layer network. The output of 1st layer is the input of 2nd layer.
model = keras.Sequential([
    layers.Dense(256, activation='relu'),
    layers.Dense(128, activation='relu'),
    layers.Dense(10)])
```

The number of output nodes in the three layers is 256, 128 and 10 respectively. Calling model (x) can directly return the output of the last layer of the model.

3.8.2 Model training

After building the 3-layer neural network, given the input x, call model(x) to get the model output o, and calculate the current loss $\mathcal{L}$:

```
with tf.GradientTape() as tape: # Record the gradient calculation
    # Flatten x, [b, 28, 28] => [b, 784]
    x = tf.reshape(x, (-1, 28*28))
    # Step1. get output [b, 784] => [b, 10]
    out = model(x)
    # [b] => [b, 10]
```

```
y_onehot = tf.one_hot(y, depth=10)
# Calculate squared error, [b, 10]
loss = tf.square(out-y_onehot)
# Calculate the mean squared error, [b]
loss = tf.reduce_sum(loss) / x.shape[0]
```

Then we use the auto differentiation function from TensorFlow -- tape.gradient(loss, model.trainable_variables) to calculate all the gradients $\frac{\partial \mathcal{L}}{\partial \theta}, \theta \in \{\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2, \boldsymbol{W}_3, \boldsymbol{b}_3\}$。

```
# Step3. Calcualte gradients w1, w2, w3, b1, b2, b3
grads = tape.gradient(loss, model.trainable_variables)
```

The gradient results are saved using the grads list variable. Then we use the optimizers object to automatically update the model parameters θ according to the gradient update rule.

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

Code is as follows:

```
# Auto gradient calculation
grads = tape.gradient(loss, model.trainable_variables)
# w' = w - lr * grad, update parameters
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

After multiple iterations, the learned model f_θ can be used to predict the categorical probability of unknown pictures. The model testing part is not discussed here for now.

The training error curve of the MNIST data set is shown in Figure 0.10. Because the 3-layer neural network has relatively strong expression ability and the task of handwritten digital picture recognition is relatively simple, the training error decreases quickly and steadily. In the figure, the x-axis represents the times of iterating over all training samples, which is called Epoch. Iterating all training samples once is call one Epoch. We can test the model's accuracy and other indicators after several Epoches to monitor the model training effect.

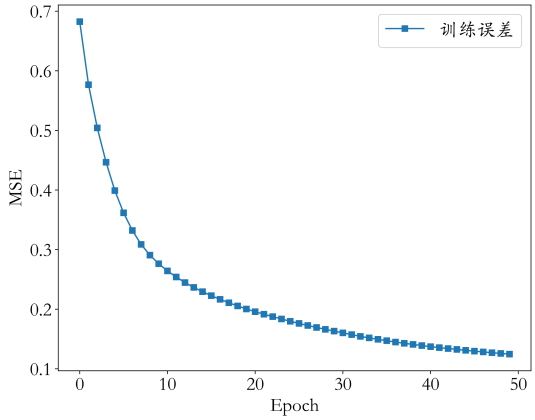


Figure 0.10 Training error of MNIST dataset

3.9 Summary

In this chapter, by analogizing a one-layer linear regression model to the classification problem, we proposed a three-layer nonlinear neural network model to solve the problem of handwritten digital picture recognition. After this chapter, everyone should have a good understanding of the (shallow) neural network algorithms. Next, we will learn some basic knowledge of TensorFlow and lay a solid foundation for subsequent learning and implementation of deep learning algorithms.

3.10 Reference

[1] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, 1998.