Chapter 4 Basic TensorFlow

I envision that in the future, we may be equivalent to
robot pet dogs, and by then I will also support robots.
−Claude Shannon

TensorFlow is a scientific computing library of deep learning algorithms. All operations are performed based on tensor objects. Complex neural network algorithms are essentially a combination of basic operations such as multiplication and addition of tensors. Therefore, it is important to get familiar with the basic operation of tensors in TensorFlow. Only by mastering these operation methods can we realize various complex and novel network models at will and understand the essence of various models and algorithms.

4.1 Data types

The basic data types in TensorFlow include numeric, string, and Boolean.

4.1.1 Numeric

Numeric tensor is the main data format of TensorFlow. According to the dimension, it can be divided into:

- ❑ Scalar: a single real number, such as 1.2 and 3.4, has a dimension of 0 and a shape of [].
- ❑ Vector: an ordered set of real numbers, wrapped by square brackets, such as $[1.2]$ and $[1.2, 3.4]$, has a dimensions of 1 and a shape of [n] depending on the length.
- ❑ Matrix: an ordered set of real numbers in n rows and m columns, such as $[[1,2],[3,4]]$, has a dimensions of 2 and a shape of $[n, m]$.
- ❑ Tensor: an array with dimension greater than 2. Each dimension of the tensor is also known as the axis. Generally, each dimension represents specific physical meaning. For example, a tensor with a shape of $[2,32,32,3]$ has 4 dimensions. If it represents image data, each dimension or axis represents the number of images, image height, image width, and number of color channels, that is, 2 represents 2 pictures, 32 represents image height and width are both 32, and 3 represents a total of 3 color channels, i.e. RGB. The number of dimensions of the tensor and the specific physical meaning represented by each dimension need to be defined by users.

In TensorFlow, scalars, vectors, and matrices are also collectively referred to as tensors without distinction. You need to make your own judgment based on the dimension or shape of tensors. The same convention applies in this book.

First, let's create a scalar in TensorFlow. The implementation is as follows:

```
In [1]:
a = 1.2 # Create a scalar in Python
aa = tf.constant(1.2)  # Create a scalar in TensorFlow
```

```
type(a), type(aa), tf.is_tensor(aa)
Out[1]:
     (float, tensorflow.python.framework.ops.EagerTensor, True)
```

If we want to use the functions provided by TensorFlow, we must create tensors in the way specified by TensorFlow, not the standard Python language. We can print out the relevant information of tensor x through print (x) or x, the code is as follows:

```
In [2]: x = tf.constant([1,2.,3.3])
x # print out x
Out[2]:
<tf.Tensor: id=165, shape=(3,), dtype=float32, numpy=array([1. , 2. , 3.3],
dtype=float32)>
```

In the output, id is the index of the internal object in TensorFlow, shape represents the shape of the tensor, and dtype represents the numerical precision of the tensor. The numpy () method can return data in the type of Numpy.array, which is convenient for exporting data to other modules in the system.

```
In [3]:  x.numpy()   # Convert TensorFlow (TF) tensor to numpy array
Out[3]:
array([1. , 2. , 3.3], dtype=float32)
```

Unlike scalars, the definition of a vector must be passed to the tf.constant () function through a List container. For example, to create a vector:

```
In [4]:
a = tf.constant([1.2])  # Create a vector with one element
a, a.shape
Out[4]:
(<tf.Tensor: id=8, shape=(1,), dtype=float32, numpy=array([1.2],
dtype=float32)>,
 TensorShape([1]))
```

Create a vector with 3 elements:

```
In [5]:
a = tf.constant([1,2, 3.])
a, a.shape
Out[5]:
 (<tf.Tensor: id=11, shape=(3,), dtype=float32, numpy=array([1., 2., 3.],
dtype=float32)>,
 TensorShape([3]))
```

Similarly, the implementation of a matrix is as follows:

```
In [6]:
a = tf.constant([[1,2],[3,4]])  # Create a 2x2 matrix
a, a.shape
```

```
Out[6]:
(<tf.Tensor: id=13, shape=(2, 2), dtype=int32, numpy=
 array([[1, 2],
       [3, 4]])>, TensorShape([2, 2]))
```

A three-dimensional tensor can be defined as

```
In [7]:
a = tf.constant([[[1,2],[3,4]],[[5,6],[7,8]]])
Out[7]:
<tf.Tensor: id=15, shape=(2, 2, 2), dtype=int32, numpy=
array([[[1, 2],
       [3, 4]],
      [[5, 6],
       [7, 8]]])>
```

4.1.2 String

In addition to numeric types, TensorFlow also supports string type. For example, when processing image data, we can first record the path string of the images, and then read the image tensors according to the path through the preprocessing function. A string tensor can be created by passing in a string object, for example:

```
In [8]:
a = tf.constant('Hello, Deep Learning.')
Out[8]:
<tf.Tensor: id=17, shape=(), dtype=string, numpy=b'Hello, Deep Learning.'>
```

The tf.strings module provides common utility functions for strings, such as lower(), join(), length(), and split(). For example, we can convert all strings to lowercase:

```
In [9]:
tf.strings.lower(a)  # Convert string a to lowercase
Out[9]:
<tf.Tensor: id=19, shape=(), dtype=string, numpy=b'hello, deep learning.'>
```

Deep learning algorithms are mainly based on numerical tensor operations, and string data is used less frequently, so we won't go into too much detail here.

4.1.3 Boolean

In order to facilitate the comparison operation, TensorFlow also supports Boolean tensors. We can easily convert Python standard boolean data into a TensorFlow internal Boolean as follows:

```
In [10]: a = tf.constant(True)
Out[10]:
```

```
<tf.Tensor: id=22, shape=(), dtype=bool, numpy=True>
```

Similarly, we can create a boolean vector as follows:

```
In [1]:
a = tf.constant([True, False])
Out[1]:
<tf.Tensor: id=25, shape=(2,), dtype=bool, numpy=array([ True, False])>
```

It should be noted that the Tensorflow and standard Python Boolean types are not always equivalent and cannot be used universally, for example:

```
In [1]:
a = tf.constant(True) # Create TF Boolean data
a is True # Whether a is a Python Boolean
Out[1]:
False # TF Boolean is not a Python Boolean
In [2]:
a == True  # Are they numerically the same?
Out[2]:
<tf.Tensor: id=8, shape=(), dtype=bool, numpy=True> # Yes, numerically, they
are equal.
```

4.2 Numerical precision

For a numeric tensor, it can be saved with different byte length corresponding to different precision. For example, floating point number 3.14 can be saved with 16-bit, 32-bit or 64-bit precision. The longer the bit, the higher the accuracy, and of course, the larger memory space the number occupies. Commonly used precision types in TensorFlow are tf.int16, tf.int32, tf.int64, tf.float16, tf.float32, and tf.float64 where tf.float64 is known as tf.double.

When creating a tensor, we can specify its precision, for example:

```
In [12]:
tf.constant(123456789, dtype=tf.int16)
tf.constant(123456789, dtype=tf.int32)
Out[12]:
<tf.Tensor: id=33, shape=(), dtype=int16, numpy=-13035>
<tf.Tensor: id=35, shape=(), dtype=int32, numpy=123456789>
```

Note that when precision is too low, the data 123456789 overflows and the wrong result is returned. Generally, tf.int32 and tf.int64 precision are used more often for intergers. For floating point numbers, high-precision tensors can represent data more accurately. For example, when tf.float32 is used for π, the actual data saved is 3.1415927.

```
In [1]:
import numpy as np
tf.constant(np.pi, dtype=tf.float32)  # Save pi with 32 byte
```

```
Out[1]:

<tf.Tensor: id=29, shape=(), dtype=float32, numpy=3.1415927>
```

If we use tf.float64, we can get higher precision:

```
In [2]:

tf.constant(np.pi, dtype=tf.float64)  # Save pi with 64 byte
Out[2]:

<tf.Tensor: id=31, shape=(), dtype=float64, numpy=3.141592653589793>
```

For most deep learning algorithms, tf.int32 and tf.float32 are able to generally meet the accuracy requirements. Some algorithms that require higher accuracy, such as reinforcement learning, can use tf.int64 and tf.float64.

The tensor precision can be accessed through the dtype property. For some operations that can only handle a specified precision type, the precision type of the input tensor needs to be checked in advance, and the tensor that does not meet the requirements should be converted to the appropriate type using tf.cast function. For example:

```
In [3]:

a = tf.constant(3.14, dytpe=tf.float16)

print('before:',a.dtype)  # Get a’s precision

if a.dtype != tf.float32:  # If a is not tf.float32, set it to be tf.float32.

    a = tf.cast(a,tf.float32)  # Convert a to tf.float32

print('after :',a.dtype)  # Get a’s current precision
Out[3]:

before: <dtype: 'float16'>

after : <dtype: 'float32'>
```

When performing type conversion, you need to ensure the legality of the conversion operation. For example, when converting a high-precision tensor into a low-precision tensor, hidden data overflow risks may occur:

```
In [4]:

a = tf.constant(123456789, dtype=tf.int32)

tf.cast(a, tf.int16)  # Convert a to lower precision and we have overflow
Out[4]:

<tf.Tensor: id=38, shape=(), dtype=int16, numpy=-13035>
```

Conversions between boolean and integer types are also legal and are common:

```
In [5]:

a = tf.constant([True, False])

tf.cast(a, tf.int32)  # Convert boolean to integers

Out[5]:

<tf.Tensor: id=48, shape=(2,), dtype=int32, numpy=array([1, 0])>
```

In general, 0 means False and 1 means True during type conversion. In TensorFlow, non-zero numbers are treated as True, for example:

```
In [6]:
a = tf.constant([-1, 0, 1, 2])
tf.cast(a, tf.bool)  # 整型转布尔类型
Out[6]:
<tf.Tensor: id=51, shape=(4,), dtype=bool, numpy=array([ True, False,  True,
True])>
```

4.3 Tensors to be optimized

In order to distinguish tensors that need to calculate gradient information from tensors that do not need to calculate gradient information, TensorFlow adds a special data type to support the recording of gradient information: tf.Variable. The tf.Variable adds attributes such as name and trainable on the basis of ordinary tensors to support the construction of computing graphs. Since the gradient operation consumes a large amount of computing resources and automatically updates related parameters, tf.Variable does not need to be encapsulated for tensors that don't need optimization, such as the input $\boldsymbol{X}$ of a neural network. Instead, for tensors that need to calculate the gradient, such as the $\boldsymbol{W}$ and $\boldsymbol{b}$ of neural network layers, need to be wrapped by tf.Variable in order for TensorFlow to track relevant gradient information.

The tf.Variable () function can be used to convert an ordinary tensor into a tensor to be optimized, for example:

```
In [20]:
a = tf.constant([-1, 0, 1, 2])  # Create TF tensor
aa = tf.Variable(a)  # Convert to tf.Variable type
aa.name, aa.trainable # Get tf.Variable properties
Out[20]:
 ('Variable:0', True)
```

The name and trainable attributes are specific for the tf.Variable type. The name attribute is used to name the variables in the computing graph. This naming system is maintained internally by TensorFlow and generally does not require users to do anything about it. The trainable attribute indicates whether the tensor needs to be optimized. When the Variable object is created, the optimization flag is enabled by default. You can set the trainable attribute to be False to create the tensor without optimization.

In addition to creating tf.Variable tensors through ordinary tensors, you can also create them directly, for example:

```
In [21]:
a = tf.Variable([[1,2],[3,4]])  # Directly create Variable type tensor
Out[21]:
<tf.Variable 'Variable:0' shape=(2, 2) dtype=int32, numpy=
array([[1, 2],
       [3, 4]])>
```

The tf.Variable tensors can be considered as a special type of ordinary tensors. In fact, ordinary tensors can also be temporarily added to a list of tracking gradient information through GradientTape.watch() method in order to support the automatic differentitation function.

4.4 Create tensors

In TensorFlow, you can create tensors in a variety of ways, such as from a Python list, from a Numpy array, or from a known distribution.

4.4.1 Create tensors from array and list

Numpy array and Python list are very important data containers in Python. Many data are loaded into the array or list before being converted to tensors. The output data of TensorFlow are also usually exported to array or list, which makes them easy to use for other modules.

The tf.convert_to_tensor function can be used to create a new tensor and import data stored in a Python list or Numpy array into the new tensor, for example:

```
In [22]:
# Create a tensor from a Python list
tf.convert_to_tensor([1,2.])
Out[22]:
<tf.Tensor: id=86, shape=(2,), dtype=float32, numpy=array([1., 2.],
dtype=float32)>
In [23]:
# Create a tensor from a Numpy array
tf.convert_to_tensor(np.array([[1,2.],[3,4]]))
Out[23]:
<tf.Tensor: id=88, shape=(2, 2), dtype=float64, numpy=
array([[1., 2.],
       [3., 4.]])>
```

Note that Numpy floating-point arrays store data with 64-bit precision by default. When converting to tensor type, the precision is tf.float64. You can convert it to tf.float32 when needed. In fact, both tf.constant() and tf.convert_to_tensor() can automatically convert Numpy arrays or Python lists to tensor types.

4.4.2 Create all 0 or all 1 tensors

Creating tensors with all 0s or 1s is a very common tensor initialization method. Consider linear transformation $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, the weight matrix $\boldsymbol{W}$ can be initialized with a matrix of all 1s, and $\boldsymbol{b}$ is initialized with a vector of all 0s. So the linear transformation changes to $\boldsymbol{y} = \boldsymbol{x}$. We can use tf.zeros() and tf.ones() to create tensors with arbitrary shapes and all 0s or 1s

```
In [24]:tf.zeros([]),tf.ones([])
Out[24]:
 (<tf.Tensor: id=90, shape=(), dtype=float32, numpy=0.0>,
 <tf.Tensor: id=91, shape=(), dtype=float32, numpy=1.0>)
```

Create a vector of all 0s and all 1s:

```
In [25]:tf.zeros([1]),tf.ones([1])
Out[25]:
(<tf.Tensor: id=96, shape=(1,), dtype=float32, numpy=array([0.],
dtype=float32)>,
 <tf.Tensor: id=99, shape=(1,), dtype=float32, numpy=array([1.],
dtype=float32)>)
```

Create a matrix of all zeros:

```
In [26]:tf.zeros([2,2])
Out[26]:
<tf.Tensor: id=104, shape=(2, 2), dtype=float32, numpy=
array([[0., 0.],
       [0., 0.]], dtype=float32)>
```

Create a matrix of all 1s:

```
In [27]:tf.ones([3,2])
Out[27]:
<tf.Tensor: id=108, shape=(3, 2), dtype=float32, numpy=
array([[1., 1.],
       [1., 1.],
       [1., 1.]], dtype=float32)>
```

With tf.zeros_like, tf.ones_like, you can easily create a tensor with all 0s or 1s that is consistent with the shape of another tensor. For example, to create an all-zero tensor with the same shape as the tensor a:

```
In [28]:a = tf.ones([2,3])  # Create a 2x3 tensor with all 1s
tf.zeros_like(a)  # Create a all zero tensor with the same shape of a
Out[28]:
<tf.Tensor: id=113, shape=(2, 3), dtype=float32, numpy=
array([[0., 0., 0.],
       [0., 0., 0.]], dtype=float32)>
```

Create a all 1 tensor with the same shape as the tensor a:

```
In [29]: a = tf.zeros([3,2])  # Create a 3x2 tensor with all 0s
tf.ones_like(a)  # Create a all 1 tensor with the same shape of a
Out[29]:
```

```
<tf.Tensor: id=120, shape=(3, 2), dtype=float32, numpy=
array([[1., 1.],
      [1., 1.],
      [1., 1.]], dtype=float32)>
```

tf.*_ like is a series of convenient functions that can be implemented by tf.zeros (a.shape) and other methods.

4.4.3 Create a customized numeric tensor

In addition to initialize a tensor with all 0s or 1s, sometimes it is also necessary to initialize the tensor with a specific value, such as -1. With tf.fill(shape, value), we can create a tensor with a specific numeric value, where the dimension is specified by the shape parameter. For example, to create a scalar with element -1:

```
In [30]:tf.fill([], -1)  #
Out[30]:
<tf.Tensor: id=124, shape=(), dtype=int32, numpy=-1>
```

To create a vector with all elements -1:

```
In [31]:tf.fill([1], -1)
Out[31]:
<tf.Tensor: id=128, shape=(1,), dtype=int32, numpy=array([-1])>
```

To create a matrix with all elements 99:

```
In [32]:tf.fill([2,2], 99)  # Create a 2x2 matrix with all 99s
Out[32]:
<tf.Tensor: id=136, shape=(2, 2), dtype=int32, numpy=
array([[99, 99],
      [99, 99]])>
```

4.4.4 Create a tensor from a known distribution

Normal distribution (or Gaussian distribution) and uniform distribution are the two most common distributions. It is very useful to create tensors sampled from these two distributions. For example, in convolutional neural networks, the convolution kernel $\boldsymbol{W}$ is usually initialized from a normal distribution to facilitate the training process. In adversarial networks, hidden variables $\boldsymbol{z}$ are generally sampled from a uniform distribution.

With tf.random.normal(shape, mean=0.0, stddev=1.0), we can create a tensor with dimension defined by the shape parameter and values sampled from a normal distribution $\mathcal{N}(\text{mean}, \text{stddev}^2)$. For example, to create a tensor from a normal distribution with mean 0 and standard deviation of 1:

```
In [33]: tf.random.normal([2,2])  # Create a 2x2 tensor from a normal
distribution
```

```
Out[33]:

<tf.Tensor: id=143, shape=(2, 2), dtype=float32, numpy=

array([[-0.4307344 ,  0.44147003],

      [-0.6563149 , -0.30100572]], dtype=float32)>
```

To create a tensor from a normal distribution with mean of 1 and standard deviation of 2:

```
In [34]: tf.random.normal([2,2], mean=1,stddev=2)

Out[34]:

<tf.Tensor: id=150, shape=(2, 2), dtype=float32, numpy=

array([[-2.2687864, -0.7248812],

      [ 1.2752185,  2.8625617]], dtype=float32)>
```

With tf.random.uniform(shape, minval=0, maxval=None, dtype=tf.float32) we can create an uniformly distributed tensor sampled from the interval $[minval, maxval)$. For example, to create a matrix uniformaly sampled from the interval $[0,1)$ with a shape of $[2,2]$:

```
In [35]: tf.random.uniform([2,2])

Out[35]:

<tf.Tensor: id=158, shape=(2, 2), dtype=float32, numpy=

array([[0.65483284, 0.63064325],

      [0.008816  , 0.81437767]], dtype=float32)>
```

To create a matrix uniformly sampled from an interval $[0,10)$ with a shape of $[2,2]$:

```
In [36]: tf.random.uniform([2,2],maxval=10)

Out[36]:

<tf.Tensor: id=166, shape=(2, 2), dtype=float32, numpy=

array([[4.541913  , 0.26521802],

      [2.578913  , 5.126876  ]], dtype=float32)>
```

If we need to uniformly sample integers, we must specify the maxval parameter and set the data type as tf.int*:

```
In [37]:

# Create a integer tensor from a uniform distribution with interval [0,100)

tf.random.uniform([2,2],maxval=100,dtype=tf.int32)

Out[37]:

<tf.Tensor: id=171, shape=(2, 2), dtype=int32, numpy=

array([[61, 21],

      [95, 75]])>
```

4.4.5 Create a sequence

When looping or indexing a tensor, it is often necessary to create a continuous sequence of integers, which can be implemented by the tf.range() function. tf.range(limit, delta=1) can create integer sequences with delta steps and within interval $[0, \text{limit})$. For example, to create an integer sequence of 0 to 10 with step of 1:

```
In [38]: tf.range(10)  # 0~10, 10 is not included
Out[38]:
<tf.Tensor: id=180, shape=(10,), dtype=int32, numpy=array([0, 1, 2, 3, 4, 5,
6, 7, 8, 9])>
```

To create a integer sequence between 0 and 10 with step of 2:

```
In [39]: tf.range(10,delta=2) # 10 is not included
Out[39]:
<tf.Tensor: id=185, shape=(5,), dtype=int32, numpy=array([0, 2, 4, 6, 8])>
```

With tf.range(start, limit, delta=1), we can create an integer sequence within interval $[\text{start}, \text{limit})$ and step of delta:

```
In [40]: tf.range(1,10,delta=2)  # 1~10, 10 is not included
Out[40]:
<tf.Tensor: id=190, shape=(5,), dtype=int32, numpy=array([1, 3, 5, 7, 9])>
```

4.5 Typical applications of tensors

After introducing the relevant properties and creation methods of tensor, the following will introduce the typical application of tensor in each dimension, so that readers can intuitively think of its main physical meaning and purpose and lay the foundation for the study of a series of abstract operations such as the dimensional transformation of subsequent tensors.

This section will inevitably mention the network models or algorithms that will be learned in future chapters. You don't need to fully understand them now, but can have a preliminary impression.

4.5.1 Scalar

In TensorFlow, a scalar is the easiest to understand. It is a simple number with 0 dimension and a shape of []. Typical uses of scalars are the representation of error values and various metrics, such as accuracy, precision, and recall.

Consider the training curve of a model. As shown in Figure 0.1, the x-axis is the number of training steps, and the y-axis is Loss per Query Image error change (Figure 0.1 (a)) and accuracy change (Figure 0.1 (b)), where the loss value and accuracy are scalars generated by tensor calculation.

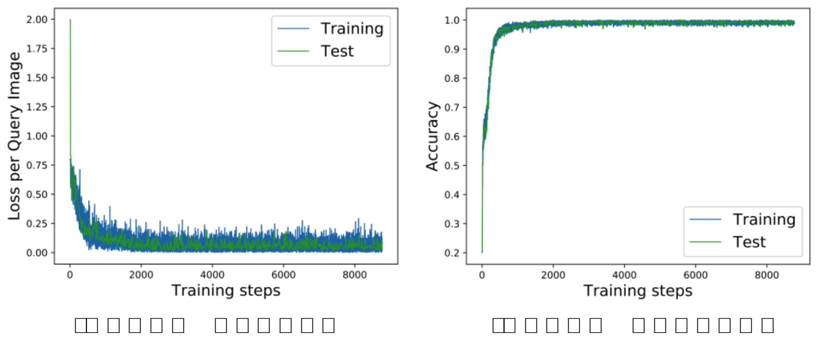

Figure 0.1 Loss and accuracy curves

Take the mean square error function as an example, after tf.keras.losses.mse (or tf.keras.losses.MSE, the same function) returns the error value on each sample, and finally takes the average value of the error as the error of the current Batch, it automatically becomes a scalar:

```
In [41]:

out = tf.random.uniform([4,10]) # Create a model output example
y = tf.constant([2,3,2,0]) # Create a real observation
y = tf.one_hot(y, depth=10) # one-hot encoding
loss = tf.keras.losses.mse(y, out) # Calculate MSE for each sample

loss = tf.reduce_mean(loss) # Calculate the mean of MSE

print(loss)

Out[41]:

tf.Tensor(0.19950335, shape=(), dtype=float32)
```

4.5.2 Vector

Vectors are very common in neural networks. For example, in fully connected networks and convolutional neural networks, bias tensors $\boldsymbol{b}$ are represented by vectors. As shown in Figure 0.2, an bias value is added to the output nodes of each fully connected layer, and the bias of all output nodes are represented as a vector form $\boldsymbol{b} = [b_1, b_2]^{\mathrm{T}}$:

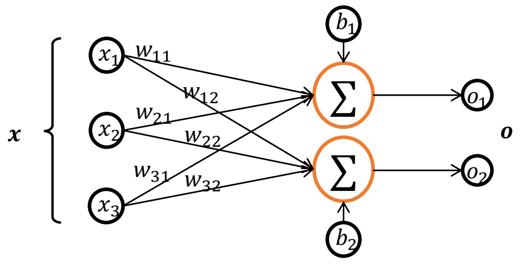

Figure 0.2 Application of bias vectors

Considering a network layer of 2 output nodes, we create a bias vector of length 2 and add back on each output node:

```
In [42]:
# Suppose z is the output of an activation function
z = tf.random.normal([4,2])
b = tf.zeros([2]) # Create a bias vector
z = z + b
Out[42]:
<tf.Tensor: id=245, shape=(4, 2), dtype=float32, numpy=
array([[ 0.6941646 ,  0.4764454 ],
       [-0.34862405, -0.26460952],
       [ 1.5081744 , -0.6493869 ],
       [-0.26224667, -0.78742725]], dtype=float32)>
```

Note that the tensor $\boldsymbol{z}$ with shape [4, 2] and the vector $\boldsymbol{b}$ with shape [2] can be added directly. Why is this? We will reveal it in the Broadcasting section.

For a network layer created through the high-level interface class Dense(), the tensors $\boldsymbol{W}$ and $\boldsymbol{b}$ are automatically created and managed by the class internally. The bias variable $\boldsymbol{b}$ can be accessed through the bias member of the fully connected layer. For example, if a linear layer network with 4 input nodes and 3 output nodes is created, then the length of its bias vector $\boldsymbol{b}$ should have length of 3 as follows:

```
In [43]:
fc = layers.Dense(3) # Create a dense layer with output length of 3
# Create W and b through build function with input nodes of 4
fc.build(input_shape=(2,4))
fc.bias # Print bias vector
Out[43]:
<tf.Variable 'bias:0' shape=(3,) dtype=float32, numpy=array([0., 0., 0.],
dtype=float32)>
```

It can be seen that the bias member of the class is a vector of length 3 and is initialized to all 0s. This is also the default initialization scheme of the bias $\boldsymbol{b}$. Besides, the type of the bias vector is Variable, because $\boldsymbol{W}$ and $\boldsymbol{b}$ are both parameters to be optimized.

4.5.3 Matrix

A matrix is also a very common type of tensor. For example, the shape of a batch input tensor $\boldsymbol{X}$ of a fully connected layer is $[b, d_{\text{in}}]$, where b represents the number of input samples, that is, Batch Size, and d_{in} represents the length of the input feature. For example, the feature length is 4, and the input containing a total of 2 samples can be expressed as a matrix:

```
x = tf.random.normal([2,4])  # A tensor with 2 samples and 4 features
```

Let the number of output nodes of the fully connected layer be 3, then the shape of its weight tensor $\boldsymbol{W}$ is [4,3]. We can directly implement a network layer using the tensors $\boldsymbol{X}$, $\boldsymbol{W}$ and vector

b, the code is as follows:

```
In [44]:
w = tf.ones([4,3])
b = tf.zeros([3])
o = x@w+b # @ means matrix mulitiplication
Out[44]:
<tf.Tensor: id=291, shape=(2, 3), dtype=float32, numpy=
array([[ 2.3506963,  2.3506963,  2.3506963],
      [-1.1724043, -1.1724043, -1.1724043]], dtype=float32)>
```

In above code, both $\boldsymbol{X}$ and $\boldsymbol{W}$ are matrices. The above code implements a linear transformation network layer, and the activation function is empty. In general, the network layer $\sigma(\boldsymbol{X}@\boldsymbol{W}+\boldsymbol{b})$ is called a fully connected layer, which can be directly implemented by the Dense() class in TensorFlow. In particular, when the activation function σ is empty, the fully connected layer is also called a linear layer. We can create a network layer with 4 input nodes and 3 output nodes through the Dense() class, and view its weight matrix $\boldsymbol{W}$ through the kernel member names of the fully connected layer:

```
In [45]:
fc = layers.Dense(3) # Create fully-connectted layer with 3 output nodes
fc.build(input_shape=(2,4)) # Define the input nodes to be 4
fc.kernel # Check kernel matrix W
Out[45]:
<tf.Variable 'kernel:0' shape=(4, 3) dtype=float32, numpy=
array([[ 0.06468129, -0.5146048 , -0.12036425],
      [ 0.71618867, -0.01442951, -0.5891943 ],
      [-0.03011459,  0.578704  ,  0.7245046 ],
      [ 0.73894167, -0.21171576,  0.4820758 ]], dtype=float32)>
```

4.5.4 Three-dimensional tensor

A typical application of a three-dimensional tensor is to represent a sequence signal. Its format is

$$\boldsymbol{X} = [b, \text{sequence length}, \text{feature length}]$$

Where the number of sequence signals is b, sequence length represents the number of sampling points or steps in the time dimension, and feature length represents the feature length of each point.

Consider the representation of sentences in Natural Language Processing (NLP), such as the sentiment classification network that evaluates whether a sentence is a positive sentiment or not, as shown in Figure 0.3. In order to facilitate the processing of strings by neural networks, words are generally encoded into vectors of fixed length through the Embedding Layer. For example, "a" is encoded as a vector of length 3, then 2 sentences with equal length (each sentence has 5 words) can be expressed as a 3-dimensional tensor with shape of [2,5,3], where 2 represents the number of sentences, 5 represents the number of words, and 3 represents the length of the encoded word vector. We demonstrate how to represent sentences through the IMDB dataset as follows:

```
In [46]:  # Load IMDB dataset
(x_train,y_train),(x_test,y_test)=keras.datasets.imdb.load_data(num_words=10
000)
# Convert each sentence to length of 80 words
x_train = keras.preprocessing.sequence.pad_sequences(x_train,maxlen=80)
x_train.shape
Out [46]: (25000, 80)
```

We can see that the shape of the x_train is [25000,80], where 25000 represents the number of sentences, 80 represents a total of 80 words in each sentence, and each word is represented by a numeric encoding method. Next we use the layers.Embedding function to convert each numeric encoded word into a vector of length 100:

```
In [47]: # Create Embedding layer with 100 output length
embedding=layers.Embedding(10000, 100)
# Convert numeric encoded words to word vectors
out = embedding(x_train)
out.shape
Out[47]: TensorShape([25000, 80, 100])
```

Through the Embedding layer, the shape of the sentence tensor becomes [25000,80,100], where 100 represents that each word is encoded as a vector of length 100.

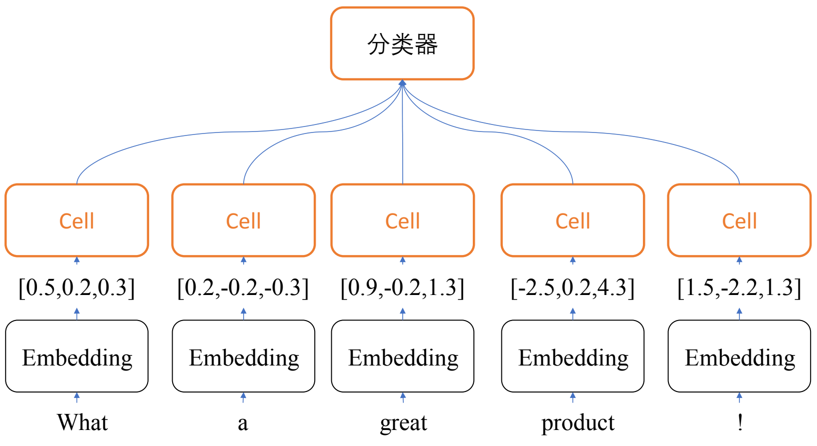

Figure 0.3 Sentiment classification network

For a sequence signal with one feature, such as the price of a product within 60 days, only one scalar is required to represent the product price, so the price change of 2 products can be expressed using a tensor of shape [2,60]. In order to facilitate the uniform format, the price change is also expressed as a tensor of shape [2,60,1], where 1 represents the feature length of 1.

4.5.5 Four-dimensional tensor

Most times we only use tensors with dimension less than five. For larger dimension tensors, such as five-dimensional tensor representation in meta learning, similar principle can be applied. Four-dimensional tensors are widely used in convolutional neural networks. They are used to save feature maps. The format is generally defined as

$$[b, h, w, c]$$

where b indicates the number of input samples, h and w represent the height and width of the feature map respectively, and c is the number of channels. Some deep learning frameworks also use the format of $[b, c, h, w]$, such as PyTorch. Image data is a type of feature map. For a color image with 3 channels of RGB, each image contains h rows and w columns of pixels. Each point requires 3 values to represent the color intensity of the RGB channel, so a picture can be expressed using a tensor of shape $[h, w, 3]$. As shown in Figure 0.4, the top picture represents the original image, which contains the intensity information of the three lower channels.

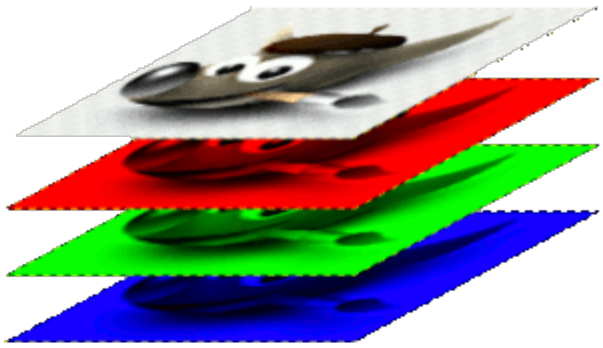

Figure 0.4 Feature maps of RGB images

In neural networks, multiple inputs are generally calculated in parallel to improve the computation efficiency, so the tensor of b pictures can be expressed as $[b, h, w, 3]$:

```
In [48]:
# Create 4 32x32 color images
x = tf.random.normal([4,32,32,3])
# Create convolutional layer
layer = layers.Conv2D(16,kernel_size=3)
out = layer(x)
out.shape
Out[48]:TensorShape([4, 30, 30, 16])
```

The convolution kernel tensor is also a 4-dimensional tensor, which can be accessed through the kernel member variable:

```
In [49]:layer.kernel.shape
Out[49]:TensorShape([3, 3, 3, 16])
```

4.6 Indexing and slicing

Part of the tensor data can be extracted through indexing and slicing operations, and they are used very frequently.

4.6.1 Indexing

In TensorFlow, the standard Python indexing method is supported, such as $[i][j]$, comma and :. Consider 4 color pictures with 32×32 size (for the convenience, most of the tensors are generated by random distribution simulation, the same hereinafter), the corresponding tensor has shape $[4,32,32,3]$ as follow:

```
x = tf.random.normal([4,32,32,3])
```

Next we use the index method to read part of the data from the tensor.

- ❑ Read the first image data:

```
x = tf.random.normal ([4,32,32,3]) #Create a 4D tensor
In [51]:x[0]  # Index 0 indicates the 1st element in Python
Out[51]:<tf.Tensor: id=379, shape=(32, 32, 3), dtype=float32, numpy=
array([[[ 1.3005302 ,  1.5301839 , -0.32005513],
        [-1.3020388 ,  1.7837263 , -1.0747638 ], ...
        [-1.1092019 , -1.045254  , -0.4980363 ],
        [-0.9099222 ,  0.3947732 , -0.10433522]]], dtype=float32)>
```

- ❑ Read the 2nd row of the 1st picture:

```
In [52]:x[0][1]
Out[52]:
<tf.Tensor: id=388, shape=(32, 3), dtype=float32, numpy=
array([[ 4.2904025e-01,  1.0574218e+00,  3.1540772e-01],
       [ 1.5800388e+00, -8.1637271e-02,  6.3147342e-01], ...,
       [ 2.8893018e-01,  5.8003378e-01, -1.1444757e+00],
       [ 9.6100050e-01, -1.0985689e+00,  1.0827581e+00]], dtype=float32)>
```

- ❑ Read the 2nd row and 3rd column of the 1st picture:

```
In [53]: x[0][1][2]
Out[53]:
<tf.Tensor: id=401, shape=(3,), dtype=float32, numpy=array([-0.55954427,
0.14497331,  0.46424514], dtype=float32)>
```

- ❑ Select the 2nd row, 1st column and 2nd (B) channel of the 3rd picture:

```
In [54]:x[2][1][0][1]
Out[54]:
<tf.Tensor: id=418, shape=(), dtype=float32, numpy=-0.84922135>
```

When the number of dimension is large, the way of using $[i][j]...[k]$ is inconvenient. Instead, we can use the $[i, j, ..., k]$ for indexing. They are equivalent.

❑ Read the 10th row and 3rd column of the 2nd picture:

```
In [55]: x[1,9,2]
Out[55]:
<tf.Tensor: id=436, shape=(3,), dtype=float32, numpy=array([ 1.7487534 , -
0.41491988, -0.2944692 ], dtype=float32)>
```

4.6.2 Slicing

A slice of data can be easily extracted by start: end: step, where start is the index of the starting reading position, end is the index of the ending reading position (excluding), and step is the sampling step size.

Taking the image tensor with shape $[4,32,32,3]$ as an example, we explain how to use slicing to obtain data at different positions. For example, reading the second and third pictures as follows:

```
In [56]: x[1:3]
Out[56]:
<tf.Tensor: id=441, shape=(2, 32, 32, 3), dtype=float32, numpy=
array([[[[ 0.6920027 ,  0.18658352,  0.0568333 ],
        [ 0.31422952,  0.75933754,  0.26853144],
        [ 2.7898    , -0.4284912 , -0.26247284],...
```

There are many abbreviations for the start: end: step slicing method. The start, end, and step parameters can be selectively omitted as needed. When all of them are omitted like ::, it indicates that the reading is from the beginning to the end, and the step size is 1. For example, x [0, ::] means read all the rows of the first picture, where :: means all the rows in the row dimension, which is equivalent to x [0]:

```
In [57]:x[0,::]  # Read 1st picture
Out[57]:
<tf.Tensor: id=446, shape=(32, 32, 3), dtype=float32, numpy=
array([[[ 1.3005302 ,  1.5301839 , -0.32005513],
       [-1.3020388 ,  1.7837263 , -1.0747638 ],
       [-1.1230233 , -0.35004002,  0.01514002],
       ...
```

For brevity, :: can be shortened to a single colon :, for example:

```
In [58]: x[:,0:28:2,0:28:2,:]
Out[58]:
<tf.Tensor: id=451, shape=(4, 14, 14, 3), dtype=float32, numpy=
array([[[[ 1.3005302 ,  1.5301839 , -0.32005513],
         [-1.1230233 , -0.35004002,  0.01514002],
         [ 1.3474811 ,  0.639334  , -1.0826371 ],
         ...
```

The above code represents reading all pictures, interlaced sampling, and reading all channel data, which is equivalent to scaling 50% of the original height and width of the picture.

Let's summarize different ways of slicing, where “start” can be omitted when reading from the first element, that is, “start = 0” can be omitted, “end” can be omitted when the last element is taken, and “step” can be omitted when the step length is 1. The details are summarized in Table 4.1:

Table 0.1 Summary of slicing methods

Method	Meaning
start:end:step	Read from “start” to “end” (excluding) with step length of “step”
start:end	Read from “start” to “end” (excluding) with step length of 1
start:	Read from “start” to the end of object with step length of 1
start::step	Read from “start” to the end of object with step length of “step”
:end:step	Read from 0th item to “end” (excluding) with step length of “step”
:end	Read from 0th item to “end” (excluding) with step length of 1
::step	Read from 0th item to the last item with step length of “step”
::	Read all items
:	Read all items

In particular, step can be negative. For example, $\text{start}:\text{end}:-1$, it means starting from “start”, reading in reverse order and ending with “end” (excluding), and the index “end” is smaller than “start”. Consider a simple sequence vector from 0 to 9, and take the first element in reverse order, excluding the first element:

```
In [59]: x = tf.range(9)  # Create the vector
x[8:0:-1]  # Reverse slicing
Out[59]:
<tf.Tensor: id=466, shape=(8,), dtype=int32, numpy=array([8, 7, 6, 5, 4, 3,
2, 1])>
```

Fetch all elements in reverse order as follows:

```
In [60]: x[::-1]
Out[60]:
<tf.Tensor: id=471, shape=(9,), dtype=int32, numpy=array([8, 7, 6, 5, 4, 3,
2, 1, 0])>
```

Reverse sampling every 2 items is implemented as follows:

```
In [61]: x[::-2]
Out[61]:
<tf.Tensor: id=476, shape=(5,), dtype=int32, numpy=array([8, 6, 4, 2, 0])>
```

Read all the channels of each picture, where both rows and columns are sampled in reverse every two elements. The implementation is as follows:

```
In [62]: x = tf.random.normal([4,32,32,3])
x[0,::-2,::-2]
Out[62]:
<tf.Tensor: id=487, shape=(16, 16, 3), dtype=float32, numpy=
array([[[ 0.63320625,  0.0655185 ,  0.19056146],
       [-1.0078577 , -0.61400175,  0.61183935],
       [ 0.9230892 , -0.6860094 , -0.01580668],
       …
```

When the tensor has large dimensions, the dimensions that do not need to be sampled generally use a single colon ":" to indicate that all elements are selected. As a result, a lot of ":" may appear. Consider the image tensor with shape $[4,32,32,3]$. When the data on the G channel needs to be read, all the previous dimensions are extracted as:

```
In [63]: x[:,:,:,1]  # Read data on G channel
Out[63]:
<tf.Tensor: id=492, shape=(4, 32, 32), dtype=float32, numpy=
array([[[ 0.575703  ,  0.11028383, -0.9950867 , ...,  0.38083118,
        -0.11705163, -0.13746642],
       ...
```

In order to avoid the situation of too many colons like $x[:,:,:,1]$, we can use symbols "⋯ " to take all the data in multiple dimensions, where the number of dimensions needs to be automatically inferred according to the rules: when the symbol ⋯ appears in slice mode, the dimension to the left of "⋯" will be automatically aligned to the left maximum. The dimension to the right of the symbol "⋯" will be automatically aligned to the far right. The system will automatically infer the number of dimensions represented by the symbol "⋯". The details are summarized in Table 4.2:

Table 0.2 "…" sliding method summary

Method	Meaning
a,⋯,b	Select 0 to a for dimension a, b to end for dimension b, all elements for other dimensions.
a,⋯	Select 0 to a for dimension a, all elements for other dimensions.

⋯,b	Select b to end for dimension b, all elements for other dimensions.
⋯	Read all elements

We list more examples as follows:

- ❑ Read the G and B channel data of the first and second pictures:

```
In [64]: x[0:2,...,1:]
Out[64]:
<tf.Tensor: id=497, shape=(2, 32, 32, 2), dtype=float32, numpy=
array([[[[ 0.575703  ,  0.8872789 ],
         [ 0.11028383, -0.27128693],
         [-0.9950867 , -1.7737272 ],
         ...
```

- ❑ Read the last two pictures:

```
In [65]:x[2:,...]  # equivalent to x[2:]
Out[65]:
<tf.Tensor: id=502, shape=(2, 32, 32, 3), dtype=float32, numpy=
array([[[[-8.10753584e-01,  1.10984087e+00,  2.71821529e-01],
         [-6.10031188e-01, -6.47952318e-01, -4.07003373e-01],
         [ 4.62206364e-01, -1.03655539e-01, -1.18086267e+00],
         ...
```

- ❑ Read R and G channel data:

```
In [66]:x[...,:2]
Out[66]:
<tf.Tensor: id=507, shape=(4, 32, 32, 2), dtype=float32, numpy=
array([[[[-1.26881   ,  0.575703  ],
         [ 0.98697686,  0.11028383],
         [-0.66420585, -0.9950867 ],
         ...
```

4.6.3 Slicing summary

Tensor indexing and slicing methods are various, especially the slicing operation, which is easy for beginners to get confused. In essence, the slicing operation has only this basic form of start: end: step. Through this basic form, some default parameters are purposefully omitted, and multiple abbreviated methods are derived. So it is easier and faster to write. Since the number of dimensions that deep learning generally deals with is within four dimensions, you will find that the tensor slice operation is not that complicated in deep learning.

4.7 Dimension transformation

In neural networks, dimensional transformation is the core tensor operation. Through dimensional transformation, the data can be arbitrarily switched to meet the computing needs of different situations. So why do we need dimensional transformation? Consider the batch form of the linear layer:

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

It is assumed that 2 samples are included in $\boldsymbol{X}$, each of which has a feature length of 4 and a shape of $[2,4]$. The output nodes of the linear layer is 3, that is, the shape of $\boldsymbol{W}$ is $[4,3]$, and the shape of $\boldsymbol{b}$ is defined $[3]$. Then the result of $\boldsymbol{X}@\boldsymbol{W}$ has shape of $[2,3]$. Note that we also need to add $\boldsymbol{b}$ with shape $[3]$. How to add 2 tensors of different shapes directly?

Recall that what we want to do is adding a bias to each output node of each layer. This bias is shared by all samples at each node. In other words, each sample should add the same bias at each node as shown in Figure 0.5:

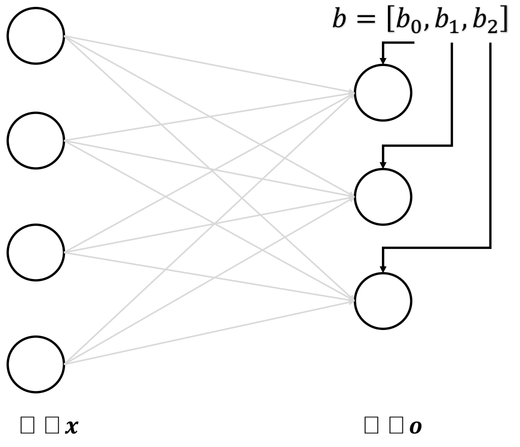

Figure 0.5 Bias of a linear layer

Therefore, for the input $\boldsymbol{X}$ of 2 samples, we need to copy the bias

$$\boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

to the number of samples into the following matrix form:

$$\boldsymbol{B}' = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_1 & b_2 & b_3 \end{bmatrix}$$

and then add $\mathbf{X}' = \mathbf{X}@\mathbf{W}$

$$\mathbf{X}' = \begin{bmatrix} x'_{11} & x'_{12} & x'_{13} \\ x'_{21} & x'_{22} & x'_{23} \end{bmatrix}$$

Because they have the same shape at this time, which satisfies the mathematical conditions of matrix addition:

$$\mathbf{Y} = \mathbf{X}' + \mathbf{B}' = \begin{bmatrix} x'_{11} & x'_{12} & x'_{13} \\ x'_{21} & x'_{22} & x'_{23} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 & b_3 \\ b_1 & b_2 & b_3 \end{bmatrix}$$

In this way, it not only satisfies the mathematical condition that the matrix addition needs to be consistent in shape, but also achieves the logic of sharing the bias vector to the output nodes of each input sample. In order to achieve this, we insert a new dimension—Batch, to the bias vector ***b***, then copy the data in the Batch dimension to get a transformed version $\mathbf{B}'$ with shape of [2,3]. This series of operations is called dimensional transformation.

Each algorithm has different logical requirements for tensor format. When the existing tensor format does not meet the algorithm requirements, the tensor needs to be adjusted to the correct format through dimensional transformation. Basic dimensional transformation includes functions such as changing the view (reshape()), inserting new dimensions (expand_dims()), deleting dimensions (squeeze()), and exchanging dimensions (transpose()).

4.7.1 Reshape

Before introducing the reshape operation, let's first understand the concept of tensor storage and view. The view of the tensor is the way we understand the tensor. For example, the tensor of shape [2,4,4,3] is logically understood as 2 pictures, each picture has 4 rows and 4 columns, and each pixel has 3 channels of RGB data. The storage of a tensor is reflected in that the tensor is stored in the memory as a continuous area. For the same storage, we can have different ways of view. For the [2,4,4,3] tensor, we can consider it as 2 samples, each of which is characterized by a vector of length 48. The same tensor can produce different views. This is the relationship between storage and view. View generation is very flexible, but needs to be reasonable.

We can generate a vector through tf.range (), and generate different views through tf.reshape() function, for example:

```
In [67]: x=tf.range(96)
x=tf.reshape(x,[2,4,4,3])  # Change view to [2,4,4,3] without change storage
Out[67]:  # Data is not changed, only view is changed.
<tf.Tensor: id=11, shape=(2, 4, 4, 3), dtype=int32, numpy=
array([[[[ 0,  1,  2],
         [ 3,  4,  5],
         [ 6,  7,  8],
         [ 9, 10, 11]],…
```

When storing data, memory does not support this dimensional hierarchy concept, and can only be written to memory in a tiled and sequential manner. Therefore, this hierarchical relationship needs to be managed manually, that is, the storage order of each tensor needs to be manually tracked. For ease of expression, we refer to the dimension on the left side of the tensor shape list as the large dimension, and the dimension on the right side of the shape list as the small dimension. For example, in a tensor of shape [2,4,4,3], the number of images 2 is called the large dimension and the number of channels 3 is called the small dimension. Under the setting of priority to write in small dimension first, the memory layout of the above tensor **x** is

1	2	3	4	5	6	7	8	9	…	…	…	93	94	95

Changing the view of the tensor only changes the way the tensor is understood. It does not change the storage order. Because the writing of a large amount of data consumes more computing resources, this is done to increase the computation efficiency. Because the data has only a flat structure when stored and it is separate from the logical structure, the new logical structure (view) does not need to change the data storage mode, which can save a lot of computing resources. While changing the view operation provides convenience, it also brings a lot of logical dangers. The main reason is that the default premise of changing the view operation is that the storage does not change, otherwise changing the view operation is illegal. We first introduce legal view transformation operations, and then introduce some illegal view transformations.

For example, tensor $\boldsymbol{A}$ is written into the memory according to the initial view of $[b, h, w, c]$. If we change the way of understanding, it can have the following understanding:

- Tensor $[b, h \cdot w, c]$ represents b pictures with $h \cdot w$ pixels and c channels.
- Tensor $[b, h, w \cdot c]$ represents b pictures with h lines and the feature length of each line is $w \cdot c$.
- Tensor $[b, h \cdot w \cdot c]$ represents b pictures, and the feature length of each picture is $h \cdot w \cdot c$.

The storage of the above views does not need to be changed, so it is all correct.

Syntactically, the view transformation only needs to make sure the total number of elements of the new view and the size of the storage area are equal, that is, the element number of the new view is equal to

$$b \cdot h \cdot w \cdot c$$

It is precisely because the view design has very few grammatical constraints and is completely defined by the user, which makes it prone to logical risks when changing the view.

Now let's consider illegal view transformations. For example, if the new view is defined as $[b, w, h, c]$, $[b, c, h * w]$ or $[b, c, h, w]$, the storage order of the tensor needs to be changed. If the storage order is not updated synchronously, the recovered data will be inconsistent with the new view, resulting in data disorder. This requires the user to understand the data in order to determine whether the operation is legal. We will show how to change the storage of tensors in the "Swap Dimensions" section.

One technique for using view transform operations correctly is to track the order of the stored dimensions. For example, tensors saved in the initial view of "number of pictures-rows-columns-channels", the storage is also written in the order of "number of pictures-rows-columns-channels". If the view is restored in the "number of pictures-pixels-channels" method, it does not conflict with the "number of pictures-rows-columns-channels", so correct data can be obtained. However, if the data is restored in the "number of pictures-channels-pixels" method, because the memory layout is in the order of "number of pictures-rows-columns-channels", the order of the view dimensions is inconsistent with the order of the storage dimensions, which leads to disordered data.

Changing views is a very common operation in neural networks. You can implement complex logic by concatenating multiple reshape operations. However, when changing views through reshape, you must always remember the storage order of the tensor. The dimensional order of the new view must be the same as the storage order. Otherwise you need to synchronize the storage order through the swap dimension operation. For example, for image data with shape $[4,32,32,3]$, shape can be adjusted to $[4,1024,3]$ by reshape operations. The view's dimensional order is $b-\text{pixel}-c$ and the tensor's storage order is $[b,h,w,c]$. The tensor with shape $[4,1024,3]$ can be restored to

- When $[b,h,w,c]=[4,32,32,3]$, the dimensional order of the new view and the storage order are consistent, and data can be recovered without disorders.
- When $[b,w,h,c]=[4,32,32,3]$, the dimensional order of the new view conflicts with the storage order.
- When $[h\cdot w\cdot c,b]=[3072,4]$, the dimensional order of the new view conflicts with the storage order.

In TensorFlow, we can obtain the number of dimensions and shape of a tensor through the tensor's ndim and shape member attributes:

```
In [68]:x.ndim,x.shape # Get the tensor’s dimension and shape
Out[68]:(4, TensorShape([2, 4, 4, 3]))
```

With tf.reshape (x, new_shape), we can legally change the view of the tensor arbitrarily, for example:

```
In [69]:tf.reshape(x,[2,-1])
Out[69]:<tf.Tensor: id=520, shape=(2, 48), dtype=int32, numpy=
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
        16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,…
        80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95]])>
```

The parameter -1 indicates that the length on the current axis needs to be automatically derived according to the rule that the total elements of the tensor are not changed. For example, the above -1 can be derived as

$$\frac{2\cdot 4\cdot 4\cdot 3}{2}=48$$

Change the view of the data again to $[2,4,12]$ as follows:

```
In [70]:tf.reshape(x,[2,4,12])
Out[70]:<tf.Tensor: id=523, shape=(2, 4, 12), dtype=int32, numpy=
array([[[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11],…
        [36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47]],
       [[48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59], …
        [84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95]]])>
```

Change the view of the data to $[2,16,3]$ again as follows:

```
In [71]:tf.reshape(x,[2,-1,3])
Out[71]:<tf.Tensor: id=526, shape=(2, 16, 3), dtype=int32, numpy=
array([[[ 0,  1,  2], …
        [45, 46, 47]],
      [[48, 49, 50],…
       [93, 94, 95]]])>
```

Through the above series of continuous view transformation operations, we need to be aware that the storage order of the tensor has not changed, and the data is still stored in the order of the initial order of $0,1,2,\cdots,95$ in memory.

4.7.2 Add and delete dimensions

Add a dimension Adding a dimension with a length of 1 is equivalent to adding the concept of a new dimension to the original data. The dimension length is 1, so the data does not need to be changed, it is only a change of view.

Consider a specific example. The data of a large-scale grayscale image is saved as a tensor of shape 28×28. At the end, a new dimension is added to the tensor, which is defined as the number of channels. Then the shape of the tensor becomes $[28,28,1]$ as follows:

```
In [72]:  # Generate a 28x28 matrix
x = tf.random.uniform([28,28],maxval=10,dtype=tf.int32)
Out[72]:
<tf.Tensor: id=11, shape=(28, 28), dtype=int32, numpy=
array([[6, 2, 0, 0, 6, 7, 3, 3, 6, 2, 6, 2, 9, 3, 0, 3, 2, 8, 1, 3, 6, 2,
        3, 9, 3, 6, 1, 7],…
```

With tf.expand_dims (x, axis), we can insert a new dimension before the specified axis:

```
In [73]:  x = tf.expand_dims(x,axis=2)
Out[73]:
<tf.Tensor: id=13, shape=(28, 28, 1), dtype=int32, numpy=
array([[[6],
        [2],
        [0],
        [0],
        [6],
        [7],
        [3],…
```

It can be seen that after inserting a new dimension, the storage order of the data has not changed. Only the view of the data is changed after inserting a new dimension.

In the same way, we can insert a new dimension at the front and name it the number of images dimension with a length of 1. At this time, the shape of the tensor becomes $[1,28,28,1]$.

```
In [74]: x = tf.expand_dims(x,axis=0)  # Insert a dimension at the beginning
Out[74]:
<tf.Tensor: id=15, shape=(1, 28, 28, 1), dtype=int32, numpy=
array([[[[6],

         [2],

         [0],

         [0],

         [6],

         [7],

         [3],…
```

Note that when the axis of tf.expand_dims is positive, it means that a new dimension is inserted before the current dimension; when it is negative, it means that a new dimension is inserted after the current dimension. Taking tensor $[b, h, w, c]$ as an example, the actual insertion position of different axis parameters is shown in Figure 0.6:

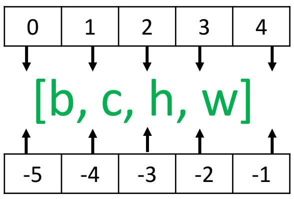

Figure 0.6 Insertion position of different axis parameters

Delete a dimension Deleting a dimension is the inverse operation of adding a dimension. As with adding a dimension, deleting a dimension can only delete a dimension of length 1, and it does not change the storage order of the tensor. Continue to consider the example of the shape [1,28,28,1]. If we want to delete the number of pictures dimension, we can use the tf.squeeze (x, axis) function. The axis parameter is the index number of the dimension to be deleted.

```
In [75]:x = tf.squeeze(x, axis=0)  # Delete the image number dimension

Out[75]:

<tf.Tensor: id=586, shape=(28, 28, 1), dtype=int32, numpy=

array([[[8],

        [2],

        [2],

        [0],…
```

Continue to delete the channel number dimension. Since the image number dimension has been deleted, the shape of x at this time is [28,28,1], so when deleting the channel number dimension, we should specify axis = 2, the implementation is as follows:

```
In [76]: x = tf.squeeze(x, axis=2)  # Delete channel dimension

Out[76]:

<tf.Tensor: id=588, shape=(28, 28), dtype=int32, numpy=

array([[8, 2, 2, 0, 7, 0, 1, 4, 9, 1, 7, 4, 8, 2, 7, 4, 8, 2, 9, 8, 8, 0,

        9, 9, 7, 5, 9, 7],
      [3, 4, 9, 9, 0, 6, 5, 7, 1, 9, 9, 1, 2, 7, 2, 7, 5, 3, 3, 7, 2, 4,
       5, 2, 7, 3, 8, 0],…
```

If we do not specify the dimension parameter axis, that is, tf.squeeze (x), it will delete all dimensions with a length of 1 by default, for example:

```
In [77]:
x = tf.random.uniform([1,28,28,1],maxval=10,dtype=tf.int32)
tf.squeeze(x)  # Delete all dimensions with length 1
Out[77]:
<tf.Tensor: id=594, shape=(28, 28), dtype=int32, numpy=
array([[9, 1, 4, 6, 4, 9, 0, 0, 1, 4, 0, 8, 5, 2, 5, 0, 0, 8, 9, 4, 5, 0,
       1, 1, 4, 3, 9, 9],…
```

It is recommended to specify the dimension parameters to be deleted one by one, in order to prevent TensorFlow from accidentally deleting certain dimensions with a length of 1, resulting in invalid calculation results.

4.7.3 Swap dimensions

Changing the view, adding or deleting dimensions will not affect the storage of the tensor. Sometimes it is not enough to change the understanding of the tensor without changing the order of the dimensions. That is, the storage order needs to be adjusted directly. By swapping the dimensions, both the storage order and the view of the tensor are changed.

Swapping imension operations are very common. For example, in TensorFlow, the default storage format of image tensor is channel after line format $[b,h,w,c]$, but the image format of some libraries is channel before line format $[b,c,h,w]$. So we need to complete the dimension swap operation. We take transformation from $[b,h,w,c]$ to $[b,c,h,w]$ as an example to introduce how to use the tf.transpose(x, perm) function to complete the dimension swap operation, where the parameter perm represents the order of the new dimension. Considering the image tensor with shape [2,32,32,3], the dimensional indexes of "number of pictures, rows, columns, and channels" are 0, 1, 2, and 3 respectively. If the order of the new dimension is "number of pictures, number of channels, row and column ", the corresponding index number becomes [0,3,1,2], so the parameter perm needs to be set to [0,3,1,2]. The implementation is as follows:

```
In [78]: x = tf.random.normal([2,32,32,3])
tf.transpose(x,perm=[0,3,1,2])  # Swap dimension
Out[78]:
<tf.Tensor: id=603, shape=(2, 3, 32, 32), dtype=float32, numpy=
array([[[[-1.93072677e+00, -4.80163872e-01, -8.85614634e-01, ...,
          1.49124235e-01,  1.16427064e+00, -1.47740364e+00],
        [-1.94761145e+00,  7.26879001e-01, -4.41877693e-01, ...
```

If we want to change $[b,h,w,c]$ to $[b,w,h,c]$, i.e. exchange the high and wide dimensions, the new dimension index becomes [0,2,1,3] as follows:

```
In [79]:
x = tf.random.normal([2,32,32,3])
tf.transpose(x,perm=[0,2,1,3]) # Swap dimension
Out[79]:
<tf.Tensor: id=612, shape=(2, 32, 32, 3), dtype=float32, numpy=
array([[[[ 2.1266546 , -0.64206547,  0.01311932],
        [ 0.918484  ,  0.9528751 ,  1.1346699 ],
        ...,
```

It should be noted that after the dimension swap is completed through tf.transpose, the storage order of the tensor has changed, and the view has changed accordingly. All subsequent operations must be based on the new order and view. Compared to the change view operation, the dimension swap operation is more computationally expensive.

4.7.4 Copy data

After inserting a new dimension, we may want to copy data on the new dimension to meet the format requirements of subsequent algorithms. Consider the example $\boldsymbol{Y} = \boldsymbol{X}@\boldsymbol{W} + \boldsymbol{b}$. After inserting a new dimension with the number of samples for $\boldsymbol{b}$, we need to copy the batch size data in the new dimension and change the shape of $\boldsymbol{b}$ to be consistent with $\boldsymbol{X}@\boldsymbol{W}$ to complete the tensor addition operation.

We can use the tf.tile(x, multiples) function to complete the data replication operation in the specified dimensions. Multiples parameter specifies the replication multiples for each dimension respectively. 1 indicates that the data will not be copied, and 2 indicates that the new length is twice of the original length, that is, a copy of the data, and so on.

Taking the input $[2,4]$ and 3-node output linear transformation layer as an example, the bias $\boldsymbol{b}$ is defined as:

$$\boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Insert new dimensions through tf.expand_dims(b, axis = 0) and turn it into a matrix:

$$\boldsymbol{B} = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

Now the shape of $\boldsymbol{B}$ becomes $[1,3]$, we need to copy data in the dimension of axis = 0 according to the number of input samples. The batch size here is 2, that is, a copy is made and becomes:

$$\boldsymbol{B} = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_1 & b_2 & b_3 \end{bmatrix}$$

Through tf.tile(b, multiples = [2,1]), it can be copied once in the axis = 0 dimension and not copied in the axis = 1 dimension. First, insert a new dimension as follows:

```
In [80]:
```

```
b = tf.constant([1,2])  # Create tensor b

b = tf.expand_dims(b, axis=0)  # Insert new dimension

b

Out[80]:

<tf.Tensor: id=645, shape=(1, 2), dtype=int32, numpy=array([[1, 2]])>
```

Copy 1 copy of the data in the Batch dimension to achieve the following:

```
In [81]:b = tf.tile(b, multiples=[2,1])

Out[81]:

<tf.Tensor: id=648, shape=(2, 2), dtype=int32, numpy=

array([[1, 2],

       [1, 2]])>
```

Now the shape of $\boldsymbol{B}$ becomes $[2,3]$ and $\boldsymbol{B}$ can be directly added to $\boldsymbol{X}@\boldsymbol{W}$. In fact, the above steps of inserting dimensions and copying data do not need to be performed by us manually, TensorFlow will automatically complete this, which is known as the automatic extension function.

Consider another example with 2x2 matrix. The implementation is as follows:

```
In [82]: x = tf.range(4)

x=tf.reshape(x,[2,2])  # Create 2x2 matrix

Out[82]:

<tf.Tensor: id=655, shape=(2, 2), dtype=int32, numpy=

array([[0, 1],

       [2, 3]])>
```

First copy one piece of data in the column dimension as follows:

```
In [83]:x = tf.tile(x,multiples=[1,2])

Out[83]:

<tf.Tensor: id=658, shape=(2, 4), dtype=int32, numpy=

array([[0, 1, 0, 1],

       [2, 3, 2, 3]])>
```

Then copy 1 piece of data in the row dimension:

```
In [84]:x = tf.tile(x,multiples=[2,1])

Out[84]:

<tf.Tensor: id=672, shape=(4, 4), dtype=int32, numpy=

array([[0, 1, 0, 1],

       [2, 3, 2, 3],

       [0, 1, 0, 1],

       [2, 3, 2, 3]])>
```

After the replication operation in 2 dimensions, we can see the shape of the data has doubled.

This example helps us understand the process of data replication more intuitively.

It should be noted that tf.tile will create a new tensor to save the copied tensor. Since the copy operation involves a large amount of data reading and writing operations, the calculation cost is relatively high. The tensor operations between different shapes in the neural network are very common, so is there a lightweight copy operation? This is the broadcasting operation to be introduced next.

4.8 Broadcasting

Broadcasting (or automatic expansion mechanism) is a lightweight tensor copying method, which logically expands the shape of the tensor data, but only performs the actual storage copy operation when needed. For most scenarios, the broadcasting mechanism can complete logical operations by avoiding the actual data copying, thereby reducing a large amount of computational cost compared to the tf.tile function.

For all dimensions of length 1, broadcasting has the same effect as tf.tile. The difference is that tf.tile creates a new tensor, performs the copy IO operation, and saves the copy. Broadcasting does not immediately copy the data, instead it will logically change the shape of the tensor, so that the view becomes the copied shape. Broadcasting will use the optimization methods of the deep learning framework to avoid the actual copying of data and complete the logical operations. For the user, the final effect of broadcasting and tf.tile copy is the same, the operation is transparent to the user, but the broadcasting mechanism saves a lot of computing resources. It is recommended to use broadcasting as much as possible in the calculation process to improve computing efficiency.

Continuing to consider the above example $\boldsymbol{Y} = \boldsymbol{X}@\boldsymbol{W} + \boldsymbol{b}$, the shape of $\boldsymbol{X}@\boldsymbol{W}$ is $[2,3]$, and the shape of $\boldsymbol{b}$ is $[3]$, we can manually complete the copy data operation by combining tf.expand_dims and tf.tile, transform $\boldsymbol{b}$ to shape $[2,3]$, and then add it to $\boldsymbol{X}@\boldsymbol{W}$. But in fact, it is also correct to add $\boldsymbol{X}@\boldsymbol{W}$ directly to $\boldsymbol{b}$ with $[3]$, for example:

```
x = tf.random.normal([2,4])

w = tf.random.normal([4,3])

b = tf.random.normal([3])

y = x@w+b # Add tensors with different shapes directly
```

The above addition does not occur a logical error, so how does it work? This is because it automatically calls the broadcasting function tf.broadcast_to(x, new_shape), expanding the two shapes to the same [2,3], that is, the above operation is equivalent to:

```
y = x@w + tf.broadcast_to(b,[2,3])
```

In other words, when the operator + encounters two tensors with inconsistent shapes, it will automatically consider expanding the two tensors to a consistent shape, and then call tf.add to complete the tensor addition operation. By automatically calling tf.broadcast_to(b, [2,3]), it not only achieves the purpose of increasing dimension and copying the data, but also avoids the expensive calculation cost of actually copying the data. At the same time, the code is more concise and efficient.

So with the broadcasting mechanism, can all tensors with inconsistent shapes complete the corresponding calculation directly? Obviously, all calculations need to be performed under the correct logic. The broadcasting mechanism does not disturb the normal calculation logic. It only automatically completes the function of adding dimensions and copying data for the most common scenarios, improving development and operating efficiency. What is this most common scenario? This comes to the core idea of broadcasting design.

The core idea of the broadcasting mechanism is universality. That is, the same data can be generally suitable for other locations. Before verifying universality, we need to align the tensor shape to the right first, and then perform universality check: for a dimension of length 1, by default this data is generally suitable for other positions in the current dimension; for dimensions that do not exist, then after adding a new dimension, the default current data is also universally applicable to the new dimension, so that it can be expanded into a tensor shape of any number of dimensions and any length.

Considering the tensor $\boldsymbol{A}$ with shape $[w,1]$, it needs to be extended to shape $[b,h,w,c]$, as shown in Figure 0.7, the first line is the shape to be expanded, and the second line is the existing shape:

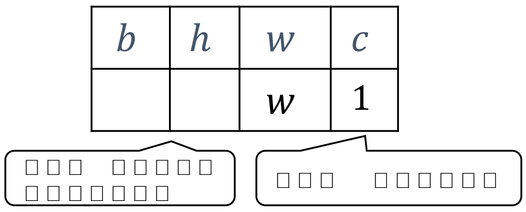

Figure 0.8 Broadcasting example 1

First align the two shapes to the right. For the channel dimension c, the current length of the tensor is 1. By default, this data is also suitable for other positions in the current dimension. The data is logically copied and the length becomes c; for the non-existing dimensions b and h, a new dimension is automatically inserted, the length of the new dimension is 1, and at the same time, the current data is generally suitable for other positions in the new dimension, that is, for other pictures and other rows, it is completely consistent with the data of the current row. This automatically expands the length of b and h to b and h, as shown in Figure 0.9.

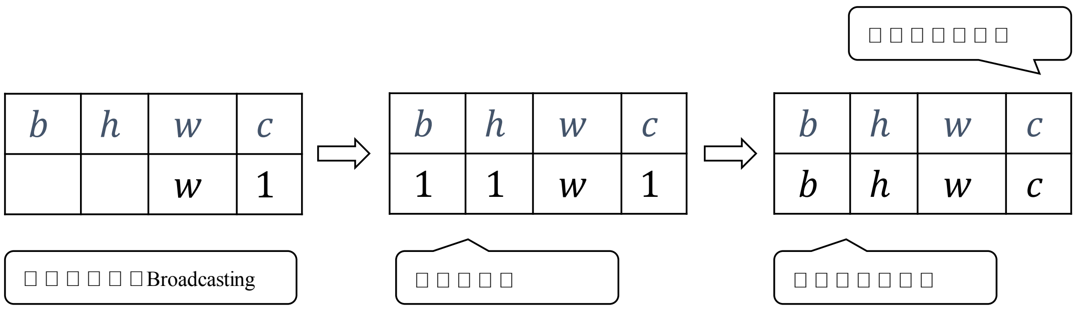

Figure 0.10 Broadcasting example 2

The tf.broadcast_to(x, new_shape) function can be used to explicitly perform the automatic expansion function to expand the existing shape to new_shape. The implementation is as follows:

```
In [87]:
A = tf.random.normal([32,1])  # Create a matrix
tf.broadcast_to(A, [2,32,32,3])  # Expand to 4 dimensions
Out[87]:
<tf.Tensor: id=13, shape=(2, 32, 32, 3), dtype=float32, numpy=
array([[[[-1.7571245 , -1.7571245 , -1.7571245 ],
        [ 1.580159  ,  1.580159  ,  1.580159  ],
        [-1.5324328 , -1.5324328 , -1.5324328 ],...
```

It can be seen that, under the guidance of the universality principle, the broadcasting mechanism has become intuitive and easy to understand, and its design is in line with human thinking patterns.

Let us consider an example that does not satisfy the principle of universality, as shown in Figure 0.11 below.

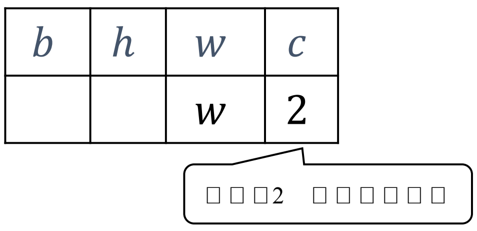

Figure 0.12 Broadcasting bad example

In the c dimension, the tensor already has 2 features, and the length of the corresponding dimension of the new shape is $c(c \neq 2$, such as = 3), then these 2 features in the current dimension cannot be universally applied to other positions, so it does not meet the universality principle. If we apply broadcasting, it will trigger errors, such as:

```
In [88]:
A = tf.random.normal([32,2])
tf.broadcast_to(A, [2,32,32,4])
Out[88]:
InvalidArgumentError: Incompatible shapes: [32,2] vs. [2,32,32,4]
[Op:BroadcastTo]
```

When performing tensor operations, some operations will automatically call the broadcasting mechanism when processing tensors of different shapes, such as +,-, *, /, etc., to broadcast the participating tensors into a common shape and then calculate accordingly. As shown in Figure 0.13, examples of tensor addition in three different shapes are demonstrated:

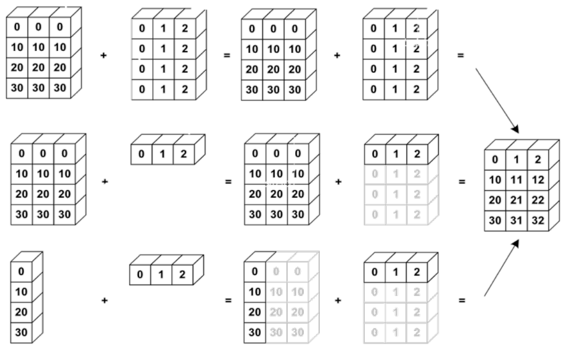

Figure 0.14 Automatic broadcasting example

Let’s simply test the automatic broadcasting mechanism of basic operators, for example:

```
a = tf.random.normal([2,32,32,1])

b = tf.random.normal([32,32])

a+b,a-b,a*b,a/b # Test automatic broadcasting for operations +, -, *, and /
```

These operations can be broadcasted into a common shape before the actual caculation. Mastering and using the broadcasting mechanism can make code more concise and efficient.

4.9 Mathematical operations

Previous chapters have used basic mathematical operations such as addition, subtraction, multiplication, and division. In this section, we will systematically introduce the common mathematical operations in TensorFlow.

4.9.1 Addition, subtraction, multiplication and division

Addition, subtraction, multiplication, and division are the most basic mathematical operations. They are implemented by tf.add, tf.subtract, tf.multiply, and tf.divide functions respectively. TensorFlow has overloaded operators $+, -, *,$ and $/$. It is generally recommended to use those operators directly. Floor dividing and remainder dividing are other two common operations, implemented by the $//$ and $\%$ operators, respectively. Let's demonstrate the division operations, for example:

```
In [89]:

a = tf.range(5)

b = tf.constant(2)

a//b # Floor dividing

Out[89]:

<tf.Tensor: id=115, shape=(5,), dtype=int32, numpy=array([0, 0, 1, 1, 2])>

In [90]:a%b # Remainder dividing
```

```
Out[90]:

<tf.Tensor: id=117, shape=(5,), dtype=int32, numpy=array([0, 1, 0, 1, 0])>
```

4.9.2 Power operations

The power operation can be conveniently completed through tf.pow(x, a) function, or the operator ** as x**a.

```
In [91]:

x = tf.range(4)

tf.pow(x,3)

Out[91]:

<tf.Tensor: id=124, shape=(4,), dtype=int32, numpy=array([ 0,  1,  8, 27])>

In [92]:x**2

Out[92]:

<tf.Tensor: id=127, shape=(4,), dtype=int32, numpy=array([0, 1, 4, 9])>
```

Set the exponent to the form of $\frac{1}{a}$ to implement the root operation $\sqrt[a]{x}$, for example:

```
In [93]: x=tf.constant([1.,4.,9.])

x**(0.5)  # square root

Out[93]:

<tf.Tensor: id=139, shape=(3,), dtype=float32, numpy=array([1., 2., 3.],
dtype=float32)>
```

In particular, for common square and square root operations, tf.square(x) and tf.sqrt(x) can be used. The square operation is implemented as follows:

```
In [94]:x = tf.range(5)

x = tf.cast(x, dtype=tf.float32)  # convert to float type

x = tf.square(x)

Out[94]:

<tf.Tensor: id=159, shape=(5,), dtype=float32, numpy=array([ 0.,  1.,  4.,
9., 16.], dtype=float32)>
```

The square root operation is implemented as follows:

```
In [95]:tf.sqrt(x)

Out[95]:

<tf.Tensor: id=161, shape=(5,), dtype=float32, numpy=array([0., 1., 2., 3.,
4.], dtype=float32)>
```

4.9.3 Exponential and logarithmic operations

Exponential operations can also be easily implemented using tf.pow(a, x) or the ** operator, for example:

```
In [96]: x = tf.constant([1.,2.,3.])

2**x

Out[96]:

<tf.Tensor: id=179, shape=(3,), dtype=float32, numpy=array([2., 4., 8.],
dtype=float32)>
```

In particular, for natural exponents e^x, this can be achieved with tf.exp(x), for example:

```
In [97]: tf.exp(1.)

Out[97]:

<tf.Tensor: id=182, shape=(), dtype=float32, numpy=2.7182817>
```

In TensorFlow, natural logarithms $\log_\mathrm{e} x$ can be implemented with tf.math.log(x), for example:

```
In [98]: x=tf.exp(3.)

tf.math.log(x)

Out[98]:

<tf.Tensor: id=186, shape=(), dtype=float32, numpy=3.0>
```

If you want to calculate the logarithm of other bases, you can use the logarithmic base-changing formula:

$$\log_a x = \frac{\log_\mathrm{e} x}{\log_\mathrm{e} a}$$

For example the calculation of $\frac{\log_\mathrm{e} \mathrm{x}}{\log_\mathrm{e} 10}$ can be achieved by:

```
In [99]: x = tf.constant([1.,2.])

x = 10**x

tf.math.log(x)/tf.math.log(10.)

Out[99]:

<tf.Tensor: id=6, shape=(2,), dtype=float32, numpy=array([1., 2.],
dtype=float32)>
```

It is relatively cumbersome to implement. Maybe TensorFlow will launch log functions with arbitrary bases in the future.

4.9.4 Matrix multiplication

The neural network contains a large number of matrix multiplication operations. We have previously introduced that the matrix multiplication can be easily implemented by the @ operator, and the tf.matmul(a, b) function. It should be noted that the matrix multiplication in TensorFlow can use the batch method, that is, the tensors $\boldsymbol{A}$ and $\boldsymbol{B}$ can have dimensions greater than 2. When the dimensions are greater than 2, TensorFlow selects the last two dimensions of $\boldsymbol{A}$ and $\boldsymbol{B}$ to perform matrix multiplication, and all the previous dimensions are considered as batch dimensions.

According to the definition of matrix multiplication, the condition of $\boldsymbol{A}$ being able to multiply a matrix $\boldsymbol{B}$ is that the length of the penultimate dimension (column) of $\boldsymbol{A}$ and the length of the penultimate dimension (row) of $\boldsymbol{B}$ must be equal. For example, tensor a with shape $[4,3,28,32]$ can be multiplied by tensor b with shape $[4,3,32,2]$. The code is as follows:

```
In [100]:
a = tf.random.normal([4,3,28,32])
b = tf.random.normal([4,3,32,2])
a@b
Out[100]:
<tf.Tensor: id=236, shape=(4, 3, 28, 2), dtype=float32, numpy=
array([[[[-1.66706240e+00, -8.32602978e+00],
        [ 9.83304405e+00,  8.15909767e+00],
        [ 6.31014729e+00,  9.26124632e-01],…
```

Matrix multiplication also supports the automatic broadcasting mechanism, for example:

```
In [101]:
a = tf.random.normal([4,28,32])
b = tf.random.normal([32,16])
tf.matmul(a,b)  # First brocast b to shape [4, 32, 16] and then multiply a
Out[101]:
<tf.Tensor: id=264, shape=(4, 28, 16), dtype=float32, numpy=
array([[[-1.11323869e+00, -9.48194981e+00,  6.48123884e+00, ...,
          6.53280640e+00, -3.10894990e+00,  1.53050375e+00],
        [ 4.35898495e+00, -1.03704405e+01,  8.90656471e+00, ...,
```

The above operation automatically expands the variable b to a common shape $[4,32,16]$ and then multiplies the variable a in batch form to obtain the resulting shape as $[4,28,16]$.

4.10 Hands-on forward propagation

So far, we have introduced how to create tensors, index slicing of tensors, dimensional transformations, and common mathematical operations. Finally, we will use the knowledge we have learned to complete the implementation of the three-layer neural network:

$$\text{out} = ReLU\{ReLU\{ReLU[\boldsymbol{X}@\boldsymbol{W}_1 + \boldsymbol{b}_1]@\boldsymbol{W}_2 + \boldsymbol{b}_2\}@\boldsymbol{W}_3 + \boldsymbol{b}_3\}$$

The data set we use is the MNIST handwritten digital picture data set. The number of input nodes is 784. The output node numbers of the first, second and third layer are 256, 128 and 10, respectively. First create the tensor parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ for each nonlinear layer as follows:

```
# Every layer's tensor needs to be optimized. Set initial bias to be 0s.
# w and b for first layer
w1 = tf.Variable(tf.random.truncated_normal([784, 256], stddev=0.1))
b1 = tf.Variable(tf.zeros([256]))
# w and b for second layer
w2 = tf.Variable(tf.random.truncated_normal([256, 128], stddev=0.1))
b2 = tf.Variable(tf.zeros([128]))
# w and b for third layer
w3 = tf.Variable(tf.random.truncated_normal([128, 10], stddev=0.1))
b3 = tf.Variable(tf.zeros([10]))
```

In forward calculation, the view of the input tensor with shape $[b, 28,28]$ is first adjusted to matrix with shape $[b, 784]$, so that it is suitable for the input format of the network:

```
    # Chanve view[b, 28, 28] => [b, 28*28]
    x = tf.reshape(x, [-1, 28*28])
```

Next finish the calculation of the first layer, we perform the automatic expansion operation here:

```
          # First layer calculation, [b, 784]@[784, 256] + [256] => [b, 256]
+ [256] => [b, 256] + [b, 256]
          h1 = x@w1 + tf.broadcast_to(b1, [x.shape[0], 256])
          h1 = tf.nn.relu(h1) # apply activation function
```

Use the same method for the second and third nonlinear function layers. The output layer can use the ReLU activation function:

```
          # Second layer calculation, [b, 256] => [b, 128]
          h2 = h1@w2 + b2
          h2 = tf.nn.relu(h2)
          # Output layer calculation, [b, 128] => [b, 10]
          out = h2@w3 + b3
```

Transform the real labeled tensor into One-hot encoding and calculate the mean square error from out. The code is as follows:

```
          # Calculate mean square error, mse = mean(sum(y-out)^2)
          # [b, 10]
          loss = tf.square(y_onehot - out)
          # Error metrics, mean: scalar
```

```
        loss = tf.reduce_mean(loss)
```

The above forward calculation process needs to be wrapped in the context of "with tf.GradientTape () as tape", so that the computing graph information can be saved during forward calculation for the automatic derivative operation.

Use the tape.gradient() function to get the gradient information of the network parameters. The result is stored in the grads list variable as follows:

```
    # Calculate gradients for [w1, b1, w2, b2, w3, b3]
    grads = tape.gradient(loss, [w1, b1, w2, b2, w3, b3])
```

Then we need to update the parameters by

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

```
    # Update parameters using assign_sub (subtract the update and assign back
to the original parameter)
    w1.assign_sub(lr * grads[0])
    b1.assign_sub(lr * grads[1])
    w2.assign_sub(lr * grads[2])
    b2.assign_sub(lr * grads[3])
    w3.assign_sub(lr * grads[4])
    b3.assign_sub(lr * grads[5])
```

Among them, assign_sub() subtracts itself from a given parameter value to implement an in-place update operation. The variation of the network training error is shown in Figure 4.11.

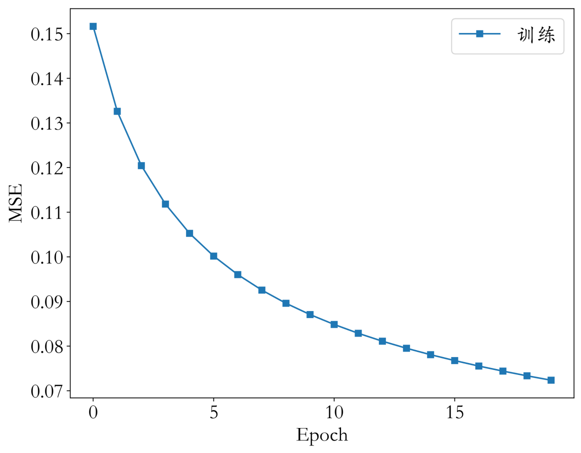

Figure 4.15 Training error of the forward calculation