Chapter 5 Advanced TensorFlow

> Artificial intelligence would be the ultimate version of Google. The ultimate search engine that would understand everything on the Web. It would understand exactly what you wanted, and it would give you the right thing.
>
> – Larry Page

After introducing the basic tensor operations, let's further explore the advanced operations, such as tensor merging and segmentation, norm statistics, tensor filling and clipping. We will also use the MNIST dataset again to enhance our understanding of tensor operations in TensorFlow.

5.1 Merge and split

5.1.1 Merge

Merging means combining multiple tensors into one tensor in a certain dimension. Taking the data of a school's gradebooks as an example, a tensor $\boldsymbol{A}$ is used to save the gradebooks of classes 1 to 4. There are 35 students in each class with a total of 8 subjects. The shape of the tensor $\boldsymbol{A}$ is: [4,35,8]. Similarly, tensor $\boldsymbol{B}$ keeps the gradebooks of the other 6 classes, with a shape of [6,35,8]. By merging these two gradebooks, you can get the gradebooks of all the classes in the school, recorded as tensor $\boldsymbol{C}$, and the corresponding shape should be [10,35,8], where 10 represents 10 classes, 35 represents 35 students, and 8 represents 8 subjects.

Tensors can be merged using concatenate and stack operations. The concatenate operation does not generate new dimensions. It only merges along existing dimensions. But the stack operation creates new dimensions. Whether to use the concatenate or stack operation to merge tensors depends on whether a new dimension needs to be created for a specific scene.

Concatenate In TensorFlow, tensors can be concatenated using tf.concat (tensors, axis) function, where the first parameter holds a list of tensors that need to be merged, and the second parameter specifies the dimensional index to be merged. Back to the above example, we merge the gradebooks in the class dimension. Here, the index number of the class dimension is 0, that is, axis = 0. The code for merging $\boldsymbol{A}$ and $\boldsymbol{B}$ is as follows:

```
In [1]:
a = tf.random.normal([4,35,8]) # Create gradebook A
b = tf.random.normal([6,35,8]) # Create gradebook B
tf.concat([a,b],axis=0) # Merge gradebooks
Out[1]:
<tf.Tensor: id=13, shape=(10, 35, 8), dtype=float32, numpy=
```

```
array([[[ 1.95299834e-01,  6.87859178e-01, -5.80048323e-01, ...,
         1.29430830e+00,  2.56610274e-01, -1.27798581e+00],
       [ 4.29753691e-01,  9.11329567e-01, -4.47975427e-01, ...,
```

In addition to the class dimension, we can also merge tensors in other dimensions. Considering that tensor $\boldsymbol{A}$ saves the first 4 subjects' scores of all students in all classes, with shape $[10,35,4]$, and tensor $\boldsymbol{B}$ saves the remaining 4 subjects' scores, with shape $[10,35,4]$. We can get the total grade book tensor by mergeing $\boldsymbol{A}$ and $\boldsymbol{B}$ as the following:

```
In [2]:
a = tf.random.normal([10,35,4])
b = tf.random.normal([10,35,4])
tf.concat([a,b],axis=2) # Merge along the last dimension
Out[2]:
<tf.Tensor: id=28, shape=(10, 35, 8), dtype=float32, numpy=
array([[[-5.13509691e-01, -1.79707789e+00,  6.50747120e-01, ...,
         2.58447856e-01,  8.47878829e-02,  4.13468748e-01],
       [-1.17108583e+00,  1.93961406e+00,  1.27830813e-02, ...,
```

Syntactically, the concatenate operation can be performed on any dimension. The only constraint is that the length of the non-merging dimension must be the same. For example, the tensors with shape $[4,32,8]$ and shape $[6,35,8]$ cannot be directly merged in the class dimension, because the length of the number of students dimension is not the same, one is 32 and the other is 35, for example:

```
In [3]:
a = tf.random.normal([4,32,8])
b = tf.random.normal([6,35,8])
tf.concat([a,b],axis=0) # Illegal merge. Second dimension is different.
Out[3]:
InvalidArgumentError: ConcatOp : Dimensions of inputs should match: shape[0]
= [4,32,8] vs. shape[1] = [6,35,8] [Op:ConcatV2] name: concat
```

Stack The concatenate operation merges data directly on existing dimensions and does not create new dimensions. If we want to create a new dimension when merging data, we need to use the tf.stack operation. Consider that tensor $\boldsymbol{A}$ saves the grade book of one class with the shape of $[35,8]$, and tensor $\boldsymbol{B}$ saves the grade book of another class with the shape of $[35,8]$. When merging the data of these two classes, we need to create a new dimension, defined as the class dimension. The new dimension can be placed in any position. Generally, the class dimension is placed before the student dimension, i.e. the new shape of the merged tensor should be $[2,35,8]$.

The tf.stack (tensors, axis) function can be used to combine multiple tensors. The first parameter represents the tensor list to be merged, and the second parameter specifies the position where the new dimension is inserted. The usage of axis is the same as that of tf.expand_dims

function. When $\text{axis} \geq 0$, a new dimension was inserted before axis. When $\text{axis} < 0$, we insert a new dimension after axis. Figure 5.1 shows the new dimension position corresponding to different axis parameter setting for a tensor with shape $[b, c, h, w]$.

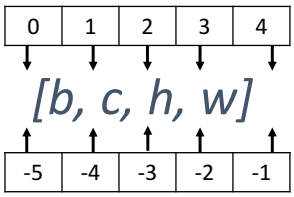

Figure 5.1 New dimension insertion position for stack operation with different axis value

Merge the two classes' gradebooks using stack operation and insert the class dimension at the axis = 0 position. The code is as follows:

```
In [4]:
a = tf.random.normal([35,8])
b = tf.random.normal([35,8])
tf.stack([a,b],axis=0) # Stack a and b and insert new dimension at axis=0
Out[4]:
<tf.Tensor: id=55, shape=(2, 35, 8), dtype=float32, numpy=
array([[[ 3.68728966e-01, -8.54765773e-01, -4.77824420e-01,
        -3.83714020e-01, -1.73216307e+00,  2.03872994e-02,
         2.63810277e+00, -1.12998331e+00],…
```

We can also choose to insert new dimensions elsewhere. For example, insert the class dimension at the end:

```
In [5]:
a = tf.random.normal([35,8])
b = tf.random.normal([35,8])
tf.stack([a,b],axis=-1) # Insert new dimension at the end
Out[5]:
<tf.Tensor: id=69, shape=(35, 8, 2), dtype=float32, numpy=
array([[[ 0.3456724 , -1.7037214 ],
        [ 0.41140947, -1.1554345 ],
        [ 1.8998919 ,  0.56994915],…
```

Now the class dimension is on axis = 2, and we need to understand the data according to the view represented by the latest dimension order. If we choose to use tf.concat to merge the above transcripts, then it would be:

```
In [6]:
```

```
a = tf.random.normal([35,8])
b = tf.random.normal([35,8])
tf.concat([a,b],axis=0) # No class dimension
```

```
Out[6]:
<tf.Tensor: id=108, shape=(70, 8), dtype=float32, numpy=
array([[-0.5516891 , -1.5031327 , -0.35369992,  0.31304857,  0.13965549,
        0.6696881 , -0.50115544,  0.15550546],
       [ 0.8622069 ,  1.0188094 ,  0.18977325,  0.6353301 ,  0.05809061,…
```

It can be seen that tf.concat can also merge data smoothly, but we need to understand the tensor data in the way that the first 35 students come from the first class and the last 35 students come from the second class, which is not very intuitive. For this example, it is obviously more reasonable to create a new dimension through the tf.stack method.

The tf.stack function also needs to meet a certain condition to use. It needs all the tensors to be merged to have the same shape. Let's what happens when stacking two tensors with different shapes.

```
In [7]:
a = tf.random.normal([35,4])
b = tf.random.normal([35,8])
tf.stack([a,b],axis=-1) # Illegal use of stack function. Different shapes.
```

```
Out[7]:
InvalidArgumentError: Shapes of all inputs must match: values[0].shape =
[35,4] != values[1].shape = [35,8] [Op:Pack] name: stack
```

The above operation attempts to merge two tensors whose shape is $[35,4]$ and $[35,8]$ respectively. Because the shapes of the two tensors are not the same, the merge operation cannot be completed.

5.1.2 Split

The inverse process of the merge operation is split, which splits a tensor into multiple tensors. Let's continue the gradebook example. We get the gradebook tensor of the entire school with shape of $[10,35,8]$, now we need to cut the data into 10 tensors in the class dimension, and each tensor holds the gradebook data of the corresponding class.

The tf.split(x, num_or_size_splits, axis) can be used to complete the tensor split operation. The meaning of the parameters in the function is as follows:

- x: the tensor to be split.
- num_or_size_splits: cutting scheme. When num_or_size_splits is a single value, such as 10, it means that the tensor x is cut into 10 parts with equal length. When num_or_size_splits is a list, each element of the list represents the length of each part. For example, num_or_size_splits=$[2,4,2,2]$ means that the tensor is cut into 4 parts, with the length of each

part as 2, 4, 2 and 2.

- axis: specify the dimension index of the split.

Now we cut the total grade book tensor into 10 pieces as follows:

```
In [8]:

x = tf.random.normal([10,35,8])

# Cut into 10 pieces with equal length

result = tf.split(x, num_or_size_splits=10, axis=0)

len(result)  # Return a list with 10 tensors of equal length

Out[8]: 10
```

We can view the shape of a tensor after cutting, and it should be all gradebook data of one class with shape of $[1,35,8]$.

```
In [9]: result[0] # Check the first class gradebook

Out[9]: <tf.Tensor: id=136, shape=(1, 35, 8), dtype=float32, numpy=

array([[[-1.7786729 ,  0.2970506 ,  0.02983334,  1.3970423 ,

         1.315918  , -0.79110134, -0.8501629 , -1.5549672 ],

       [ 0.5398711 ,  0.21478991, -0.08685189,  0.7730989 ,…
```

It can be seen that the shape of the first class tensor is $[1,35,8]$, which still has the class dimension. Let's perform unequal length cutting. For example, split the data into 4 parts with each length as $[4,2,2,2]$ for each part.

```
In [10]: x = tf.random.normal([10,35,8])

# Split tensor into 4 parts

result = tf.split(x, num_or_size_splits=[4,2,2,2] ,axis=0)

len(result)

Out[10]: 4
```

Check the shape of the first splitted tensor. According to our splitting scheme, it should contain the gradebooks of 4 classes. The shape should be $[4,35,8]$.

```
In [10]: result[0]

Out[10]: <tf.Tensor: id=155, shape=(4, 35, 8), dtype=float32, numpy=

array([[[-6.95693314e-01,  3.01393479e-01,  1.33964568e-01, ...,
```

In particular, if we want to divide one certain dimension by a length of 1, we can use the tf.unstack(x, axis) function. This method is a special case of tf.split. The splitting length is fixed as 1. We only need to specify the index number of the splitting dimension. For example, unstack the total grade book tensor in the class dimension:

```
In [11]: x = tf.random.normal([10,35,8])

result = tf.unstack(x,axis=0)

len(result) # Return a list with 10 tensors
```

```
Out[11]: 10
```

View the shape of the splitted tensor:

```
In [12]: result[0] # The first class tensor

Out[12]: <tf.Tensor: id=166, shape=(35, 8), dtype=float32, numpy=

array([[-0.2034383 ,  1.1851563 ,  0.25327438, -0.10160723,  2.094969  ,

       -0.8571669 , -0.48985648,  0.55798006],…
```

It can be seen that after splitting through tf.unstack, the splitted tensor shape becomes [35,8], that is, the class dimension disappears, which is different from tf.split.

5.2 Common statistics

During the neural networks calculations, various statistical attributes need to be computed, such as maximum, minimum, mean, and norm. Because tensors usually a lot of data, it is easier to infer the distribution of tensor values by obtaining the statistical information of these tensors.

5.2.1 Norm

Norm is a measure of the "length" of a vector. It can be generalized to tensors. In neural networks, it is often used to represent the tensor weight and the gradient magnitude. Commonly used norms are:

- L1 norm, defined as the sum of the absolute values of all the elements of the vector:

$$\|\boldsymbol{x}\|_1 = \sum_i |x_i|$$

- L2 norm, defined as the root sum of the squares of all the elements of the vector:

$$\|\boldsymbol{x}\|_2 = \sqrt{\sum_i |x_i|^2}$$

- ∞ norm, defined as the maximum of the absolute values of all elements of a vector:

$$\|\boldsymbol{x}\|_\infty = max_i(|x_i|)$$

For matrices and tensors, the above formulas can also be used after flattening the matrices and tensors into a vector. In TensorFlow, tf.norm(x, ord) function can be used to solve the L1, L2 and ∞ norms, where the parameter ord is specified as 1, 1 or np.inf for L1, L2 and ∞ norms respectively.

```
In [13]: x = tf.ones([2,2])

tf.norm(x,ord=1) # L1 norm

Out[13]: <tf.Tensor: id=183, shape=(), dtype=float32, numpy=4.0>

In [14]: tf.norm(x,ord=2) # L2 norm

Out[14]: <tf.Tensor: id=189, shape=(), dtype=float32, numpy=2.0>
```

```
In [15]: import numpy as np

tf.norm(x,ord=np.inf) # ∞ norm

Out[15]: <tf.Tensor: id=194, shape=(), dtype=float32, numpy=1.0>
```

5.2.2 Max, min, mean and sum

The tf.reduce_max, tf.reduce_min, tf.reduce_mean, and tf.reduce_sum functions can be used to get the maximum, minimum, mean, and sum of tensors in a certain dimension or in all dimensions.

Consider a tensor of shape [4,10], where the first dimension represents the number of samples and the second dimension represents the probability that the current sample belongs to each of the 10 categories. The maximum value of each sample's probability can be obtained through tf.reduce_max function:

```
In [16]: x = tf.random.normal([4,10])

tf.reduce_max(x,axis=1) # get maximum value at 2nd dimension

Out[16]:<tf.Tensor: id=203, shape=(4,), dtype=float32,
numpy=array([1.2410722 , 0.88495886, 1.4170984 , 0.9550192 ],
dtype=float32)>
```

The above code returns a vector of length 4, which represents the maximum probability value of each sample. Similarly, we can find the minimum value of the probability for each sample as follows:

```
In [17]: tf.reduce_min(x,axis=1) # get the minimum value at 2nd dimension

Out[17]:<tf.Tensor: id=206, shape=(4,), dtype=float32, numpy=array([-
0.27862206, -2.4480672 , -1.9983795 , -1.5287997 ], dtype=float32)>
```

Find the mean probabilities of each sample:

```
In [18]: tf.reduce_mean(x,axis=1)

Out[18]:<tf.Tensor: id=209, shape=(4,), dtype=float32,
numpy=array([ 0.39526337, -0.17684573, -0.148988  , -0.43544054],
dtype=float32)>
```

When the axis parameter is not specified, the tf.reduce_* functions will find the maximum, minimum, mean, and sum of all the data.

```
In [19]:x = tf.random.normal([4,10])

tf.reduce_max(x),tf.reduce_min(x),tf.reduce_mean(x)

Out [19]: (<tf.Tensor: id=218, shape=(), dtype=float32, numpy=1.8653786>,

 <tf.Tensor: id=220, shape=(), dtype=float32, numpy=-1.9751656>,

 <tf.Tensor: id=222, shape=(), dtype=float32, numpy=0.014772797>)
```

When solving the error function, the error of each sample can be obtained through the MSE function, and the average error of the sample needs to be calculated. Here we can use

tf.reduce_mean function as follows:

```
In [20]:
out = tf.random.normal([4,10]) # Simulate output
y = tf.constant([1,2,2,0]) # Real labels
y = tf.one_hot(y,depth=10) # One-hot encoding
loss = keras.losses.mse(y,out) # Calculate loss of each sample
loss = tf.reduce_mean(loss) # Calculate mean loss
loss
Out[20]:
<tf.Tensor: id=241, shape=(), dtype=float32, numpy=1.1921183>
```

Similar to the tf.reduce_mean function, the sum function tf.reduce_sum(x, axis) can calculate the sum of all features of the tensor on the corresponding axis:

```
In [21]:out = tf.random.normal([4,10])
tf.reduce_sum(out,axis=-1) # Calculate sum along the last dimension
Out[21]:<tf.Tensor: id=303, shape=(4,), dtype=float32, numpy=array([-
0.588144 ,  2.2382064,  2.1582587,  4.962141 ], dtype=float32)>
```

In addition to obtain the maximum or minimum value of the tensor, we sometimes also want to obtain the corresponding position index. For example, for the classification tasks, we need to know the position index of the maximum probability, which is ususally used as the prediction category. Considering the classification problem with 10 categories, we get the output tensor with shape [2,10], where 2 represents 2 samples and 10 indicates the probability of belonging to 10 categories. Since the position index of the element represents the probability that the current sample belongs to this category, we often use the index corresponding to the largest probability as the predicted category.

```
In [22]:out = tf.random.normal([2,10])
out = tf.nn.softmax(out, axis=1) # Use softmax to convert to probability
out
Out[22]:<tf.Tensor: id=257, shape=(2, 10), dtype=float32, numpy=
array([[0.18773547, 0.1510464 , 0.09431915, 0.13652141, 0.06579739,
        0.02033597, 0.06067333, 0.0666793 , 0.14594753, 0.07094406],
       [0.5092072 , 0.03887136, 0.0390687 , 0.01911005, 0.03850609,
        0.03442522, 0.08060656, 0.10171875, 0.08244187, 0.05604421]],
      dtype=float32)>
```

Taking the first sample as an example, it can be seen that the index with the highest probability (0.1877) is 0. Because the probability on each index represents the probability that the sample belongs to this category, the probability that the first sample belongs to class 0 is the largest. Therefore, the first sample should most likely belong to class 0. This is a typical

application where the index number of the maximum needs to be solved.

We can use tf.argmax(x, axis) and tf.argmin(x, axis) to find the index of the maximum and minimum values of x on the axis parameter. For example:

```
In [23]:pred = tf.argmax(out, axis=1)
pred
Out[23]:<tf.Tensor: id=262, shape=(2,), dtype=int64, numpy=array([0, 0],
dtype=int64)>
```

It can be seen that the maximum probability of the two samples appears on index 0, so it is most likely that they both belong to category 0. We can use category 0 as the predicted category for the two samples.

5.3 Tensor comparison

In order to get the classification metrics such as accuracy, it is generally necessary to compare the prediction result with the real label. Considering the prediction results of 100 samples, the predicted category can be obtained through tf.argmax.

```
In [24]:out = tf.random.normal([100,10])
out = tf.nn.softmax(out, axis=1) # Convert to probability
pred = tf.argmax(out, axis=1) # Find corresponding category
Out[24]:<tf.Tensor: id=272, shape=(100,), dtype=int64, numpy=
array([0, 6, 4, 3, 6, 8, 6, 3, 7, 9, 5, 7, 3, 7, 1, 5, 6, 1, 2, 9, 0, 6,
       5, 4, 9, 5, 6, 4, 6, 0, 8, 4, 7, 3, 4, 7, 4, 1, 2, 4, 9, 4,…
```

The pred variable holds the predicted category of the 100 samples. We compare them with the true labels to get a boolean tensor representing whether each sample predicts the correct one. The tf.equal(a, b) (or tf.math.equal(a, b), which is equivalent) function can compare whether the two tensors are equal, for example:

```
In [25]: # Simiulate the true labels
y = tf.random.uniform([100],dtype=tf.int64,maxval=10)
Out[25]:<tf.Tensor: id=281, shape=(100,), dtype=int64, numpy=
array([0, 9, 8, 4, 9, 7, 2, 7, 6, 7, 3, 4, 2, 6, 5, 0, 9, 4, 5, 8, 4, 2,
       5, 5, 5, 3, 8, 5, 2, 0, 3, 6, 0, 7, 1, 1, 7, 0, 6, 1, 2, 1, 3, …
In [26]:out = tf.equal(pred,y) # Compare true and prediction
Out[26]:<tf.Tensor: id=288, shape=(100,), dtype=bool, numpy=
array([False, False, False, False,  True, False, False, False, False,
       False, False, False, False, False,  True, False, False,  True,…
```

The tf.equal function returns the comparison result as a boolean tensor. We only need to count the number of True elements to get the correct number of predictions. In order to achieve this, we first convert the boolean type to an integer tensor, that is, True corresponds to 1, and False corresponds to 0, and then sum the number of 1 to get the number of True elements in the

comparison result:

```
In [27]:out = tf.cast(out, dtype=tf.float32) # convert to int type
correct = tf.reduce_sum(out) # get the number of True elements
Out[27]:<tf.Tensor: id=293, shape=(), dtype=float32, numpy=12.0>
```

It can be seen that the number of correct predictions in our randomly generated prediction data is 12, so its accuracy is

$$\text{accuracy} = \frac{12}{100} = 12\%$$

This is the normal level of random prediction models.

Except for the tf.equal function, other commonly used comparison functions are shown in Table 5.1:

Table 5.1 Common comparison functions

Function	Comparison logic
tf.math.greater	$a > b$
tf.math.less	$a < b$
tf.math.greater_equal	$a \geq b$
tf.math.less_equal	$a \leq b$
tf.math.not_equal	$a \neq b$
tf.math.is_nan	$a = \text{nan}$

5.4 Fill and copy

5.4.1 Fill

The height and width of images and the length of the sequence signals may not be the same. In order to facilitate parallel computing of the network, it is necessary to expand data of different lengths to the same. We previously introduced that the length of data can be increased by copying. However, repeatedly copying data will destroy the original data structure and is not suitable for some situations. A common practice is to fill in a sufficient number of specific values at the beginning or end of the data. These specific values (e.g. 0) generally represent invalid meanings. This operation is called Padding.

Consider a two sentence tensor that each word is represented by a digital code, such as 1 for I, 2 for like, and so on. The first sentence is “I like the weather today.” We assume that the sentence number is encoded as $[1,2,3,4,5,6]$. The second sentence is “So do I.” with encoding as $[7,8,1,6]$. In order to store the two sentence in one tensor, we need to keep the length of these two sentences consistent, that is, we need to expand the length of the second sentence to 6. A common padding scheme is to pad a number of zeros at the end of the second sentence, i.e. $[7,8,1,6,0,0]$.

Now the two sentences can be stacked and combined into a tensor of shape $[2,6]$.

The padding operation can be implemented by the tf.pad(x, paddings) function. The parameter paddings is a list of multiple nested schemes with the format of $[\text{Left Padding}, \text{Right Padding}]$. For example, $\text{paddings} = [[0,0],[2,1],[1,2]]$ indicates that the first dimension is not filled, and the left (the beginning) of the second dimension is filled with two units, and fill one unit on the right (end) of the second dimension, fill one unit on the left of the third dimension, and fill two units on the right of the third dimension. Considering the example of the above two sentences, two units need to be filled to the right of the first dimension of the second sentence, and the paddings scheme is $[[0,2]]$:

```
In [28]:a = tf.constant([1,2,3,4,5,6]) # 1st sentence
b = tf.constant([7,8,1,6]) # 2nd sentence
b = tf.pad(b, [[0,2]]) # Pad two 0s in the end of 2nd sentence
b
Out[28]:<tf.Tensor: id=3, shape=(6,), dtype=int32, numpy=array([7, 8, 1, 6,
0, 0])>
```

After filling, the shape of the two tensors is consistent, and we can stack them together, The code is as follows:

```
In [29]:tf.stack([a,b],axis=0) # Stack a and b
Out[29]:<tf.Tensor: id=5, shape=(2, 6), dtype=int32, numpy=
array([[1, 2, 3, 4, 5, 6],
       [7, 8, 1, 6, 0, 0]])>
```

In natural language processing, sentences with different length need to be loaded. Some sentences are shorter, such as only 10 words, and some sentences are longer, such as more than 100 words. In order to be able to save in the same tensor, a threshold that can cover most of the sentence length is generally selected, such as 80 words. For sentences with less than 80 words, we fill with 0s at the end of those sentences. For sentences with more than 80 words, we truncate the sentence to 80 words by removing some words at the end. We will use the IMDB dataset as an example to demonstrate how to transform sentences of unequal length into a structure of equal length. The code is as follows:

```
In [30]:total_words = 10000 # Set word number
max_review_len = 80 # Maximum length for each sentence
embedding_len = 100 # Word vector length
# Load IMDB dataset
(x_train, y_train), (x_test, y_test) =
keras.datasets.imdb.load_data(num_words=total_words)
# Pad or truncate sentences to the same length with end padding and
truncation
x_train = keras.preprocessing.sequence.pad_sequences(x_train,
maxlen=max_review_len,truncating='post',padding='post')
```

```
x_test = keras.preprocessing.sequence.pad_sequences(x_test,
maxlen=max_review_len,truncating='post',padding='post')

print(x_train.shape, x_test.shape)

Out[30]: (25000, 80) (25000, 80)
```

In the above code, we set the maximum length of the sentence max_review_len to 80 words. Through the keras.preprocessing.sequence.pad_sequences function, we can quickly complete the padding and truncation implementation. Take one of the sentences as an example and the transformed vector is like this:

```
[   1  778  128   74   12  630  163   15    4 1766 7982 1051    2   32
   85  156   45   40  148  139  121  664  665   10   10 1361  173    4
  749    2   16 3804    8    4  226   65   12   43  127   24    2   10
   10    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0]
```

We can see that the final part of the sentence is filled with 0s so that the length of the sentence is exactly 80. In fact, we can also choose to fill the beginning part of the sentence when the length of the sentence is not enough. After processing, all sentence length becomes 80, so that the training set can be uniformly stored in the tensor of shape $[25000,80]$ and the test set can be stored in the tensor of shape $[25000,80]$.

Let's introduce an example of filling in multiple dimensions at the same time. Consider padding the height and width dimensions of images. If we have pictures with dimension 28×28 and the input layer shape of neural network is 32×32, we need to fill the images to get the shape of 32×32. We can choose to fill 2 units each in the upper, lower, left, and right of the image matrix as shown in Figure 5.2.

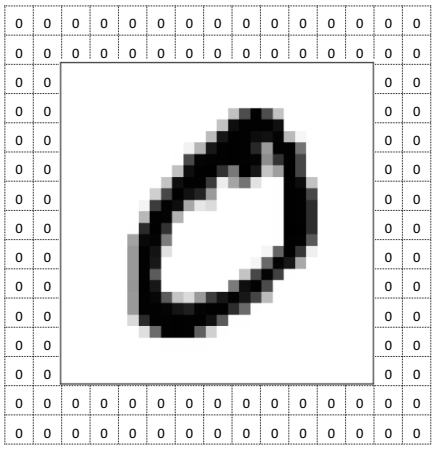

Figure 5.2 Image padding example

The above padding scheme can be implemented as follow:

```
In [31]:

x = tf.random.normal([4,28,28,1])
```

```
# Pad 2 units at each edge of the image
tf.pad(x,[[0,0],[2,2],[2,2],[0,0]])
Out[31]:
<tf.Tensor: id=16, shape=(4, 32, 32, 1), dtype=float32, numpy=
array([[[[ 0.       ],
        [ 0.       ],
        [ 0.       ],…
```

After the padding operation, the size of the picture becomes 32×32, which meets the input requirements of the neural network.

5.4.2 Copy

In the dimension transformation section, we introduced the tf.tile function of copying the dimension of length 1. Actually, the tf.tile function can be used to repeatedly copy multiple copies of data in any dimension. For example, for image data with shape $[4,32,32,3]$, if the copy scheme is multiples=[2,3,3,1], that means the channel dimension is not copied, 3 copies in the height and width dimensions, and 2 copy in the image number dimension. The implementation is as follows:

```
In [32]:x = tf.random.normal([4,32,32,3])
tf.tile(x,[2,3,3,1])
Out[32]:<tf.Tensor: id=25, shape=(8, 96, 96, 3), dtype=float32, numpy=
array([[[[ 1.20957184e+00,  2.82766962e+00,  1.65782201e+00],
        [ 3.85402292e-01,  2.00732923e+00, -2.79068202e-01],
        [-2.52583921e-01,  7.82584965e-01,  7.56870627e-01],...
```

5.5 Data limiting

Consider how to implement the nonlinear activation function ReLU. In fact, it can be implemented by simple data limiting operations with the range of elements being limited to $x \in [0,+\infty)$.

In TensorFlow, the lower limit of the data can be set through tf.maximum (x, a), that is, the upper limit of the data can be set through tf.minimum (x, a).

```
In [33]:x = tf.range(9)
tf.maximum(x,2) # Set lower limit of x to 2
Out[33]:<tf.Tensor: id=48, shape=(9,), dtype=int32, numpy=array([2, 2, 2, 3,
4, 5, 6, 7, 8])>
In [34]:tf.minimum(x,7) # Set x upper limit to 7
Out[34]:<tf.Tensor: id=41, shape=(9,), dtype=int32, numpy=array([0, 1, 2, 3,
4, 5, 6, 7, 7])>
```

Based on tf.maximum function, we can implement ReLU as follows:

```
def relu(x): # ReLU function
    return tf.maximum(x,0.) # Set lower limit of x to be 0
```

By combining tf.maximum(x, a) and tf.minimum(x, b), you can limit the upper and lower boundaries of the data at the same time, i.e. $x \in [a, b]$.

```
In [35]:x = tf.range(9)
tf.minimum(tf.maximum(x,2),7) # Set x range to be [2, 7]
Out[35]:<tf.Tensor: id=57, shape=(9,), dtype=int32, numpy=array([2, 2, 2, 3,
4, 5, 6, 7, 7])>
```

More conveniently, we can use the tf.clip_by_value function to achieve upper and lower clipping:

```
In [36]:x = tf.range(9)
tf.clip_by_value(x,2,7) # Set x range to be [2, 7]
Out[36]:<tf.Tensor: id=66, shape=(9,), dtype=int32, numpy=array([2, 2, 2, 3,
4, 5, 6, 7, 7])>
```

5.6 Advanced operations

Most of the above functions are common and easy to understand. Next, we will introduce some commonly used but slightly more complicated functions.

5.6.1 tf.gather

The tf.gather function can collect data according to the index number. Consider the example of grade books. Assume that there are 4 classes, 35 students in each class, 8 subjects in total, and the tensor shape of the grade books is [4,35,8].

```
x = tf.random.uniform([4,35,8],maxval=100,dtype=tf.int32)
```

Now we need to collect the grade books of the 1st and 2nd classes. We can give the index number of the class we want to collect (e.g. [0,1]) and specify the dimension of the class (e.g. axis = 0). And then collect the data through the tf.gather function.

```
In [38]:tf.gather(x,[0,1],axis=0) # Collect data for 1st and 2nd classes
Out[38]:<tf.Tensor: id=83, shape=(2, 35, 8), dtype=int32, numpy=
array([[[43, 10, 93, 85, 75, 87, 28, 19],
        [52, 17, 44, 88, 82, 54, 16, 65],
        [98, 26,  1, 47, 59,  3, 59, 70],…
```

In fact, the above requirements can be more conveniently achieved through slicing. However, for irregular indexing methods, such as the need to spot check the grade data of students 1, 4, 9, 12, 13, and 27, the slicing method is not suitable. The tf.gather function is designed for this situation and is more convenient to use. The implementation is as follows:

```
In [39]: # Collect the grade of students 1,4,9,12,13 and 27
tf.gather(x,[0,3,8,11,12,26],axis=1)
Out[39]:<tf.Tensor: id=87, shape=(4, 6, 8), dtype=int32, numpy=
array([[[43, 10, 93, 85, 75, 87, 28, 19],
        [74, 11, 25, 64, 84, 89, 79, 85],…
```

If we need to collect the grades of the 3rd and 5th subjects of all students, we can specify the subject dimension axis = 2 to achieve the following:

```
# Collect the grades of the 3rd and 5th subjects of all students
In [40]:tf.gather(x,[2,4],axis=2)
Out[40]:<tf.Tensor: id=91, shape=(4, 35, 2), dtype=int32, numpy=
array([[[93, 75],
        [44, 82],
        [ 1, 59],…
```

It can be seen that tf.gather is very suitable for situations where the index numbers are not regular. The index numbers can be arranged out of order, and the data collected will also be in the corresponding order. For example:

```
In [41]:a=tf.range(8)
a=tf.reshape(a,[4,2])
Out[41]:<tf.Tensor: id=115, shape=(4, 2), dtype=int32, numpy=
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])>
In [42]:tf.gather(a,[3,1,0,2],axis=0) # Collect element 4,2,1,3
Out[42]:<tf.Tensor: id=119, shape=(4, 2), dtype=int32, numpy=
array([[6, 7],
       [2, 3],
       [0, 1],
       [4, 5]])>
```

We will make the problem a little more complicated. If we want to check the subject scores of students $[3,4,6,27]$ in class $[2,3]$, we can do this by combining multiple tf.gather operations. First extract data for class $[2,3]$:

```
In [43]:
students=tf.gather(x,[1,2],axis=0) # Collect data for class 2 and 3
Out[43]:<tf.Tensor: id=227, shape=(2, 35, 8), dtype=int32, numpy=
array([[[ 0, 62, 99,  7, 66, 56, 95, 98],…
```

Then we extract the corresponding data for selected students:

```
In [44]:

tf.gather(students,[2,3,5,26],axis=1) # Collect data for students 3,4,6,27

Out[44]:<tf.Tensor: id=231, shape=(2, 4, 8), dtype=int32, numpy=

array([[[69, 67, 93,  2, 31,  5, 66, 65], …
```

Now we get the selected tensor with shape $[2,4,8]$.

This time we want to spot check all subjects of the second classmate of the second class, all subjects of the third classmate of the third class, and all subjects of the fourth classmate of the fourth class. So how does it work? Data can be manually extracted one by one in a clumsy way. First extract the data of the first sampling point: $x[1,1]$.

```
In [45]: x[1,1]

Out[45]:<tf.Tensor: id=236, shape=(8,), dtype=int32, numpy=array([45, 34,

99, 17,  3,  1, 43, 86])>
```

Then extract the data of the second sampling point $x[2,2]$ and the data of the third sampling point $x[3,3]$, and finally combine the sampling results together.

```
In [46]: tf.stack([x[1,1],x[2,2],x[3,3]],axis=0)

Out[46]:<tf.Tensor: id=250, shape=(3, 8), dtype=int32, numpy=

array([[45, 34, 99, 17,  3,  1, 43, 86],

       [11, 25, 84, 95, 97, 95, 69, 69],

       [ 0, 89, 52, 29, 76,  7,  2, 98]])>
```

Using the above method, we can correctly obtain the result of shape $[3,8]$, where 3 represents the number of sampling points, and 4 represents the data of each sampling point. The biggest problem is that the sampling is performed manually and serially, and the calculation efficiency is extremely low. Is there a better way to achieve this?

5.6.2 tf.gather_nd

With the tf.gather_nd function, we can sample multiple points by specifying the multidimensional coordinates of each sampling point. Going back to the challenge above, we want to spot check all the subjects of the second classmate of the second class, all the subjects of the third classmate of the third class, and all the subjects of the fourth classmate of the fourth class. Then the index coordinates of the three sampling points can be recorded as: $[1,1]$, $[2,2]$, and $[3,3]$, and we can combine this sampling scheme into a list $[[1,1],[2,2],[3,3]]$.

```
In [47]:

tf.gather_nd(x,[[1,1],[2,2],[3,3]])

Out[47]:<tf.Tensor: id=256, shape=(3, 8), dtype=int32, numpy=

array([[45, 34, 99, 17,  3,  1, 43, 86],

       [11, 25, 84, 95, 97, 95, 69, 69],

       [ 0, 89, 52, 29, 76,  7,  2, 98]])>
```

The result is consistent with the serial sampling method, and the implementation is more concise and efficient.

Generally, when using tf.gather_nd to sample multiple samples, for example, if we want to sample class i, student j, and subject k, we can use the expression $[\ldots,[i,j,k],\ldots]$. The inner list contains the corresponding index coordinates of each sampling point, for example:

```
In [48]:

tf.gather_nd(x,[[1,1,2],[2,2,3],[3,3,4]])

Out[48]:<tf.Tensor: id=259, shape=(3,), dtype=int32, numpy=array([99, 95,
76])>
```

In the above code, we extracted the grades of subject 1 of class 1 student 2, subject 2 of class 2 student 3, and class 3 of student 3 subject 4. There are a total of 3 grade data, and the results are summarized into a tensor with shape of $[3]$.

5.6.3 tf.boolean_mask

In addition to sampling by a given index number, sampling can also be performed by a given mask. Continue to take the gradebook tensor with shape $[4,35,8]$ as an example, this time we use the mask method for data extraction.

Consider sampling in the class dimension and set the corresponding mask as

$$\text{mask} = [\text{True}, \text{False}, \text{False}, \text{True}]$$

That is, the first and fourth classes are sampled. Using the function tf.boolean_mask(x, mask, axis), the sampling can be performed on the corresponding axis according to the mask scheme, which is realized as:

```
In [49]:

tf.boolean_mask(x,mask=[True, False,False,True],axis=0)

Out[49]:<tf.Tensor: id=288, shape=(2, 35, 8), dtype=int32, numpy=

array([[[43, 10, 93, 85, 75, 87, 28, 19],…
```

Note that the length of the mask must be the same as the length of the corresponding dimension. If we are sampling in the class dimension, we must specify the mask with length 4 to specify whether the 4 classes are sampling.

If mask sampling is performed on 8 subjects, we need to set the mask sampling scheme to

$$\text{mask} = [\text{True}, \text{False}, \text{False}, \text{True}, \text{True}, \text{False}, \text{False}, \text{True}]$$

That is, sample the first, fourth, fifth, and eighth subjects:

```
In [50]:

tf.boolean_mask(x,mask=[True,False,False,True,True,False,False,True],axis=2)

Out[50]:<tf.Tensor: id=318, shape=(4, 35, 4), dtype=int32, numpy=

array([[[43, 85, 75, 19],…
```

It is not difficult to find that the usage of tf.boolean_mask here is actually very similar to

tf.gather, except that one is sampled by the mask method, and the other is directly given the index number.

Now let's consider a multi-dimensional mask sampling method similar to tf.gather_nd. In order to facilitate the demonstration, we reduced the number of classes to two and the number of students to three. That is, a class has only three students and the tensor shape is $[2,3,8]$. If we want to sample students 1 to 2 of the first class and students 2 to 3 of the second class, we can achieve it using tf.gather_nd:

```
In [51]:x = tf.random.uniform([2,3,8],maxval=100,dtype=tf.int32)
tf.gather_nd(x,[[0,0],[0,1],[1,1],[1,2]])
Out[51]:<tf.Tensor: id=325, shape=(4, 8), dtype=int32, numpy=
array([[52, 81, 78, 21, 50,  6, 68, 19],
       [53, 70, 62, 12,  7, 68, 36, 84],
       [62, 30, 52, 60, 10, 93, 33,  6],
       [97, 92, 59, 87, 86, 49, 47, 11]])>
```

A total of 4 students' results were sampled with a shape of $[4,8]$.

If we use a mask, how do we express it? Table 5.2 expresses the sampling of the corresponding position:

Table 5.2 Sampling using mask method

	Student 0	Student 1	Student 2
Class 0	True	True	False
Class 1	False	True	True

Therefore, through this table, the sampling scheme using the mask method can be well expressed. The code is implemented as follows:

```
In [52]:
tf.boolean_mask(x,[[True,True,False],[False,True,True]])
Out[52]:<tf.Tensor: id=354, shape=(4, 8), dtype=int32, numpy=
array([[52, 81, 78, 21, 50,  6, 68, 19],
       [53, 70, 62, 12,  7, 68, 36, 84],
       [62, 30, 52, 60, 10, 93, 33,  6],
       [97, 92, 59, 87, 86, 49, 47, 11]])>
```

The result is exactly the same as tf.gather_nd method. It can be seen that tf.boolean_mask method can be used for both one and multi dimensional sampling.

The above three operations are more commonly used, especially tf.gather and tf.gather_nd. Three additional advanced operations are added below.

5.6.4 tf.where

Through the tf.where(cond, a, b) function, we can read data from the parameter a or b according to the true and false conditions of the cond condition. The condition determination rule is as follows:

$$o_i = \begin{cases} a_i & \text{cond}_i\text{为 True} \\ b_i & \text{cond}_i\text{为 False} \end{cases}$$

Among them i is the element index of the tensor. The size of the returned tensor is consistent with a and b. When the corresponding position of cond_i is True, the data is copied from a_i to o_i. Otherwise, the data is copied from b_i to o_i. Consider extracting data from 2 tensors $\boldsymbol{A}$ and $\boldsymbol{B}$ of all 1's and 0's, where the position of True in 中cond_i extracts element 1 from the corresponding position of $\boldsymbol{A}$, otherwise extracts 0 from the corresponding position of $\boldsymbol{B}$. The code is as follows:

```
In [53]:

a = tf.ones([3,3])  # Tensor A

b = tf.zeros([3,3]) # Tensor B

# Create condition matrix

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

tf.where(cond,a,b)

Out[53]:<tf.Tensor: id=384, shape=(3, 3), dtype=float32, numpy=

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

       [0., 1., 0.],

       [1., 1., 0.]], dtype=float32)>
```

It can be seen that the positions of 1 in the returned tensor are all from tensor $\boldsymbol{A}$, and the positions of 0 in the returned tensor are from tensor $\boldsymbol{B}$.

When the parameter a=b=None, that is, the a and b parameters are not specified, tf.where returns the index coordinates of all True elements in the cond tensor. Consider the following cond tensor:

```
In [54]: cond

Out[54]:<tf.Tensor: id=383, shape=(3, 3), dtype=bool, numpy=

array([[ True, False, False],

       [False,  True, False],

       [ True,  True, False]])>
```

True appears 4 times in total, and the index at the position of each True element is $[0,0]$, $[1,1]$, $[2,0]$ and $[2,1]$ respectively. The index coordinates of these elements can be obtained directly through the form of tf.where(cond) as follows:

```
In [55]:tf.where(cond)
Out[55]:<tf.Tensor: id=387, shape=(4, 2), dtype=int64, numpy=
array([[0, 0],
       [1, 1],
       [2, 0],
       [2, 1]], dtype=int64)>
```

So what's the use of this? Consider a scenario where we need to extract all the positive data and indexes in a tensor. First construct the tensor a and obtain the position masks of all positive numbers through comparison operations:

```
In [56]:x = tf.random.normal([3,3]) # Create tensor a
Out[56]:<tf.Tensor: id=403, shape=(3, 3), dtype=float32, numpy=
array([[-2.2946844 ,  0.6708417 , -0.5222212 ],
       [-0.6919401 , -1.9418817 ,  0.3559235 ],
       [-0.8005251 ,  1.0603906 , -0.68819374]], dtype=float32)>
```

By comparison operation, we get the mask of all positive numbers:

```
In [57]:mask=x>0 # equivalent to tf.math.greater()
mask
Out[57]:<tf.Tensor: id=405, shape=(3, 3), dtype=bool, numpy=
array([[False,  True, False],
       [False, False,  True],
       [False,  True, False]])>
```

Extract the index coordinates of the True element in the mask tensor via tf.where:

```
In [58]:indices=tf.where(mask) # Extract all element greater than 0
Out[58]:<tf.Tensor: id=407, shape=(3, 2), dtype=int64, numpy=
array([[0, 1],
       [1, 2],
       [2, 1]], dtype=int64)>
```

After getting the index, we can restore all positive elements through tf.gather_nd:

```
In [59]:tf.gather_nd(x,indices) # Extract all positive elements
Out[59]:<tf.Tensor: id=410, shape=(3,), dtype=float32,
numpy=array([0.6708417, 0.3559235, 1.0603906], dtype=float32)>
```

In fact, after we get the mask, we can also get all the positive elements directly through tf.boolean_mask:

```
In [60]:tf.boolean_mask(x,mask) # Extract all positive elements
Out[60]:<tf.Tensor: id=439, shape=(3,), dtype=float32,
numpy=array([0.6708417, 0.3559235, 1.0603906], dtype=float32)>
```

Through the above series of comparisons, we can intuitively feel that this function has great practical applications, and also get a deep understanding of their nature to be able to achieve our purpose in a more flexible, simple, and efficient way.

5.6.5 tf.scatter_nd

The tf.scatter_nd(indices, updates, shape) function can efficiently refresh part of the tensor data, but this function can only perform refresh operations on all 0 tensors, so it may be necessary to combine other operations to implement the data refresh function for non-zero tensors.

Figure 5.3 shows the refresh calculation principle of the one-dimensional all-zero tensor. The shape of the whiteboard is represented by the shape parameter, the index number of the data to be refreshed is represented by indices, and updates parameter contains the new data. The tf.scatter_nd(indices, updates, shape) function writes the new data to the all-zero tensor according to the index position given by indices, and return the updated result tensor.

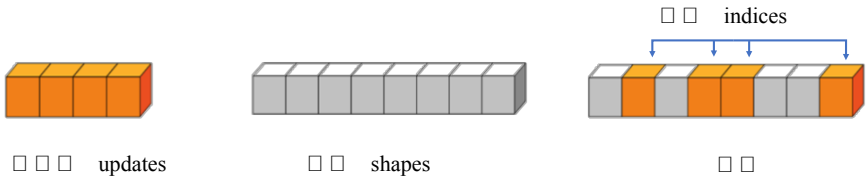

Figure 5.3 scatter_nd function for refreshing data

We implement a refresh example of the tensor in Figure 5.3 as follows:

```
In [61]: # Create indices for refreshing data
indices = tf.constant([[4], [3], [1], [7]])
# Create data for filling the indices
updates = tf.constant([4.4, 3.3, 1.1, 7.7])
# Refresh data for all 0 vector of length 8
tf.scatter_nd(indices, updates, [8])
Out[61]:<tf.Tensor: id=467, shape=(8,), dtype=float32, numpy=array([0. ,
1.1, 0. , 3.3, 4.4, 0. , 0. , 7.7], dtype=float32)>
```

It can be seen that on the all-zero tensor of length 8, the data of the corresponding positions are filled in with values from updates.

Consider an example of a 3-dimensional tensor. As shown in Figure 5.4, the shape of the all-zero tensor is a feature map with 4 channels in total, and each channel has size 4×4. New data updates has shape $[2,4,4]$, which needs to be written in indices $[1,3]$.

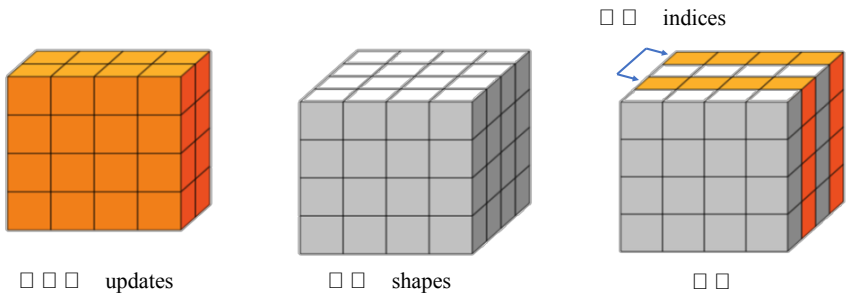

Figure 5.4 3D tensor data refreshing

We write the new feature map into the existing tensor as follows:

```
In [62]:
indices = tf.constant([[1],[3]])
updates = tf.constant([
    [[5,5,5,5],[6,6,6,6],[7,7,7,7],[8,8,8,8]],
    [[1,1,1,1],[2,2,2,2],[3,3,3,3],[4,4,4,4]]
])
tf.scatter_nd(indices,updates,[4,4,4])
Out[62]:<tf.Tensor: id=477, shape=(4, 4, 4), dtype=int32, numpy=
array([[[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],
       [[5, 5, 5, 5], # New data 1
        [6, 6, 6, 6],
        [7, 7, 7, 7],
        [8, 8, 8, 8]],
       [[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],
       [[1, 1, 1, 1], # New data 2
        [2, 2, 2, 2],
        [3, 3, 3, 3],
        [4, 4, 4, 4]]])>
```

It can be seen that the data is refreshed onto the second and fourth channel feature maps.

5.6.6 tf.meshgrid

The tf.meshgrid function can easily generate the coordinates of the sampling points of the two-dimensional grid, which is convenient for applications such as visualization. Consider the Sinc function with two independent variables x and y as:

$$z = \frac{\sin(x^2 + y^2)}{x^2 + y^2}$$

If we need to draw a 3D surface of the Sinc function in the interval $x \in [-8,8], y \in [-8,8]$, as shown in Figure 5.5, we first need to generate the grid point coordinate set of the x and y axes, so that the output value of the function at each position can be calculated by the expression of the Sinc function z. We can generate 10,000 coordinate sampling points by:

```
points = []
for x in range(-8,8,100): # Loop to generate 100 sampling point for x-axis
for y in range(-8,8,100): # Loop to generate 100 sampling point for y-axis
    z = sinc(x,y)
    points.append([x,y,z])
```

Obviously, this serial sampling method is extremely inefficient. Is there a simple and efficient way to generate grid coordinates? The answer is the tf.meshgrid function.

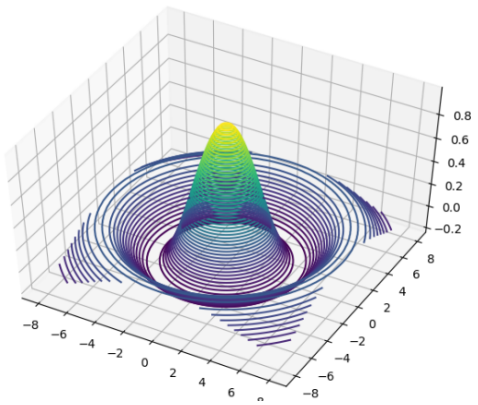

Figure 5.5 Sinc function

By sampling 100 data points on the x-axis and y-axis respectively, the tf.meshgrid(x, y) can be used to generate tensor data of these 10,000 data points and save them in a tensor of shape [100,100,2]. For the convenience of calculation, tf.meshgrid will return 2 tensors after cutting in the axis = 2 dimension, where the tensor $\boldsymbol{A}$ contains the x-coordinates of all points and the tensor $\boldsymbol{B}$ contains the y-coordinates of all points.

```
In [63]:
x = tf.linspace(-8.,8,100) # x-axis
y = tf.linspace(-8.,8,100) # y-axis
x,y = tf.meshgrid(x,y)
x.shape,y.shape
Out[63]: (TensorShape([100, 100]), TensorShape([100, 100]))
```

Using the generated grid point coordinate tensors, the Sinc function is implemented in TensorFlow as follows:

```
z = tf.sqrt(x**2+y**2)
z = tf.sin(z)/z  # sinc function
```

The matplotlib library can be used to draw the 3D surface of the function as shown in Figure 5.5.

```
import matplotlib
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


fig = plt.figure()
ax = Axes3D(fig)
# Plot Sinc function
ax.contour3D(x.numpy(), y.numpy(), z.numpy(), 50)
plt.show()
```

5.7 Load classic datasets

So far, we have learned the common tensor operations, and are ready to implement most of the deep networks. Finally, we will complete this chapter with a classification network model implemented in a tensor format. Before that, we first formally introduce how to use the tools provided by TensorFlow to load datasets conveniently for commonly used classic datasets. For loading custom datasets, we will introduce in the subsequent chapters.

In TensorFlow, the keras.datasets module provides automatic download, management, loading, and conversion functions of commonly used classic datasets, as well as the corresponding dataset objects, which facilitates multi-threading, preprocessing, shuffling, and batch-training.

Some commonly used classic datasets:

- Boston Housing: the Boston housing price trend dataset, used for training and testing of regression models.
- CIFAR10/100: a real picture dataset for picture classification tasks.
- MNIST/Fashion_MNIST: a handwritten digital picture dataset, used for picture classification tasks.
- IMDB: sentiment classification task dataset, for text classification tasks.

These datasets are used very frequently in machine learning or deep learning. For the newly proposed algorithms, it is generally preferred to test on classic datasets, and then try to migrate to larger and more complex data sets.

We can use the datasets.xxx.load_data() function to automatically load classic datasets, where xxx represents the specific dataset name, such as "CIFAR10" and "MNIST". TensorFlow will cache the data in the .keras/datasets folder in the user directory by default, as shown in Figure 5.6.

Users do not need to care about how the dataset is saved. If the current dataset is not in the cache, it will be downloaded, decompressed, and loaded automatically from the network. If it is already in the cache, the load is automatically completed. For example, to automatically load the MNIST dataset:

```
In [66]:
import  tensorflow as tf
from    tensorflow import keras
from    tensorflow.keras import datasets # Load dataset loading module
# Load MNIST dataset
(x, y), (x_test, y_test) = datasets.mnist.load_data()
print('x:', x.shape, 'y:', y.shape, 'x test:', x_test.shape, 'y test:',
y_test)
Out [66]:
x: (60000, 28, 28) y: (60000,) x test: (10000, 28, 28) y test: [7 2 1 ... 4
5 6]
```

The load_data() function will return data in the corresponding format. For the image datasets MNIST and CIFAR10, 2 tuples will be returned. The first tuple holds the training data x and y objects; the second tuple is the test data x_test and y_test objects. All data is stored in a Numpy array container.

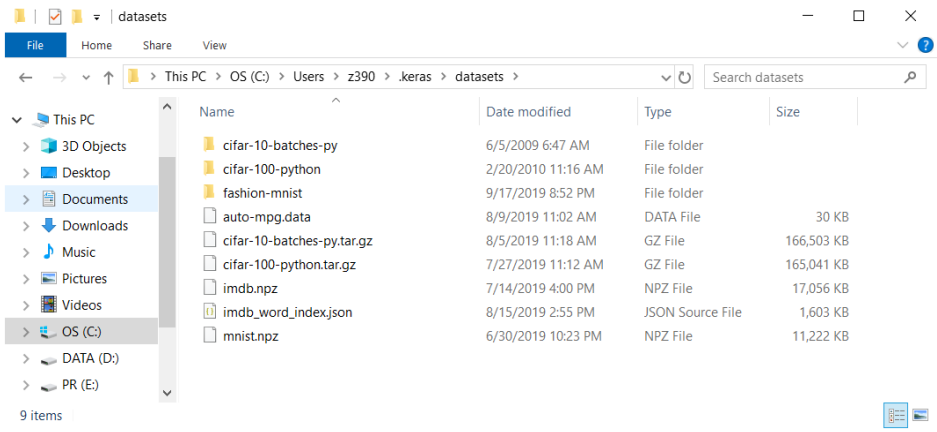

Figure 5.6 TensorFlow classic dataset saving directory

After data is loaded into memory, it needs to be converted into a Dataset object in order to take advantage of the various convenient functions provided by TensorFlow. Dataset.from_tensor_slices can be used to convert the training data image x and label y into Dataset objects:

```
# Convert to Dataset objects
train_db = tf.data.Dataset.from_tensor_slices((x, y))
```

After converting data into a Dataset object, we generally need to add a series of standard processing steps for the dataset, such as random shuffling, preprocessing, and batch loading.

5.7.1 Shuffling

Using the Dataset.shuffle(buffer_size) function, we can randomly shuffle the Dataset objects to prevent the data from being generated in a fixed order during each training, so that the model will not "remember" the label information. The code is implemented as follows:

```
train_db = train_db.shuffle(10000)
```

Here the buffer_size parameter specifies the size of the buffer pool, which is generally set to a larger constant. Calling these utility functions provided by the Dataset will return a new Dataset object.

$$\mathrm{db} = \mathrm{db}.\mathrm{step1}().\mathrm{step2}().\mathrm{step3}.()$$

This method completes all data processing steps in order, which is very convenient to implement.

5.7.2 Batch training

In order to take advantage of the parallel computing capabilities of GPUs, multiple samples are generally calculated simultaneously during the network calculation process. We call this training method batch training, and the number of samples in one batch is called batch size. In order to generate batch size samples from the Dataset at one time, the Dataset needs to be set to batch training mode. The implementation is as follows:

```
train_db = train_db.batch(128) # batch size is 128
```

Here 128 is the Batch Size parameter, that is, 128 samples are calculated at one time in parallel. Batch Size is generally set according to the user's GPU memory resources. When the GPU memory is insufficient, the Batch Size can be appropriately reduced.

5.7.3 Preprocessing

The format of the dataset loaded from keras.datasets cannot meet the model input requirements in most cases, so it is necessary to implement the preprocessing step according to the user's logic. The Dataset object can call the user-defined preprocessing logic very conveniently by providing the map(func) utility function, while the preprocessing logic is implemented in the func function. For example, the following code calls a function named preprocess to complete the preprocessing of each sample:

```
# Preprocessing is implemented in the preprocess function
train_db = train_db.map(preprocess)
```

Considering the MNIST handwritten digital picture dataset, image x loaded from keras.datasets after .batch () operation has shape $[b, 28,28]$, where the pixels are represented by integers from 0 to 255 and the label shape is $[b]$ with digital encoding. The actual neural network input generally needs to normalize the image data to the interval $[0,1]$ or $[-1,1]$ around 0. At the same time, according to the network settings, the input view of shape $[28,28]$ needs to be adjusted to a appropriate format. For label information, we can choose one-hot encoding during

preprocessing or the error calculation.

Here we map the MNIST image data to interval $[0,1]$ and adjust the view to $[b, 28 * 28]$. For label data, we choose to perform one-hot encoding in the preprocessing function. The preprocess function is implemented as follows:

```
def preprocess(x, y): # Customized preprocessing function
    x = tf.cast(x, dtype=tf.float32) / 255.
    x = tf.reshape(x, [-1, 28*28])     # flatten
    y = tf.cast(y, dtype=tf.int32)    # convert to int
    y = tf.one_hot(y, depth=10)    # one-hot encoding
    return x,y
```

5.7.4 Epoch training

For the Dataset object, we can iterate through the following ways:

```
  for step, (x,y) in enumerate(train_db): # Iterate with step
```

or

```
    for x,y in train_db: # Iterate without step
```

The x and y objects returned each time are batch samples and labels. When one iteration is completed for all samples of train_db, the for loop terminates. Compelting a batch of data training is called a Step, and completing an iteration of the entire training set through multiple steps is called an Epoch. In training, it is usually necessary to iterate multiple Epochs on the data set to obtain better training results. For example, fixed training of 20 Epoch is implemented as follows:

```
    for epoch in range(20): # Epoch number
        for step, (x,y) in enumerate(train_db): # Iteration step number
            # training...
```

In addition, we can also set a Dataset object so that the dataset will traverse multiple times before exiting such as:

```
    train_db = train_db.repeat(20) # Dataset iteration 20 times
```

The above code makes the for x, y in train_db iterates 20 epochs before exiting. No matter which of these methods is used, the same effect can be achieved. Since the previous chapter has completed the actual calculation of forward calculation, we skip it here.

5.8 Hands-on MNIST dataset

We have already introduced and implemented the forward propagation and dataset. Now let's finish the remaining classification task logic. In the training process, the error data can be effectively monitored by printing out after several steps. The code is as follows:

```
        # Print training error every 100 steps
```

```
        if step % 100 == 0:
            print(step, 'loss:', float(loss))
```

Since loss is a tensor type of TensorFlow, it can be converted to a standard Python floating-point number through the float() function. After several Steps or several Epoch trainings, a test (verification) can be performed to obtain the current performance of the model, for example:

```
        if step % 500 == 0: # Do a test every 500 steps
            # evaluate/test
```

Now let's use the tensor operation functions to complete the actual calculation of accuracy. First consider a batch sample x. The network's predicted value can be obtained through forward calculation as follows:

```
        for x, y in test_db: # Iterate through test dataset
            h1 = x @ w1 + b1 # 1st layer
            h1 = tf.nn.relu(h1) # Activation function
            h2 = h1 @ w2 + b2 # 2nd layer
            h2 = tf.nn.relu(h2) # Activation function
            out = h2 @ w3 + b3 # Output layer
```

The shape of the predicted value is $[b, 10]$. It represents the probability that the sample belongs to each category. We select the index number where the maximum probability occurs according to the tf.argmax function, which is the most likely category number of the sample:

```
            # Select the max probability category
            pred = tf.argmax(out, axis=1)
```

Since y has already been one-hot encoded in preprocessing, we can get the category number for y similarly:

```
            y = tf.argmax(y, axis=1)
```

With tf.equal, we can compare whether the two results are equal:

```
            correct = tf.equal(pred, y)
```

Sum the number of all True (converted to 1) element in the result, which is the correct number of predictions:

```
            total_correct += tf.reduce_sum(tf.cast(correct,
dtype=tf.int32)).numpy()
```

Divide the correct number of predictions by the total number of tests to get the accuracy, and print it out as follows:

```
         # Calcualte accuracy
        print(step, 'Evaluate Acc:', total_correct/total)
```

After training a simple 3-layer neural network with 20 Epochs, we achieved an accuracy of 87.25% on the test set. If we use complex neural network models and fine-tune network hyperparameters, we can get better accuracy. The training error curve is shown in Figure 5.7, and the test accuracy curve is shown in Figure 5.8.

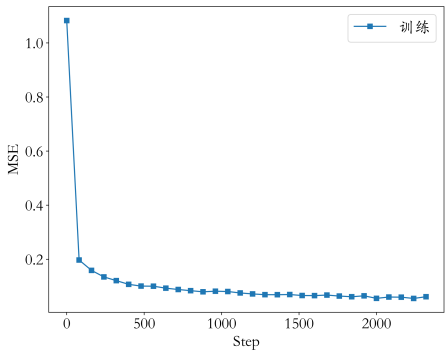

Figure 5.7 MNIST training loss

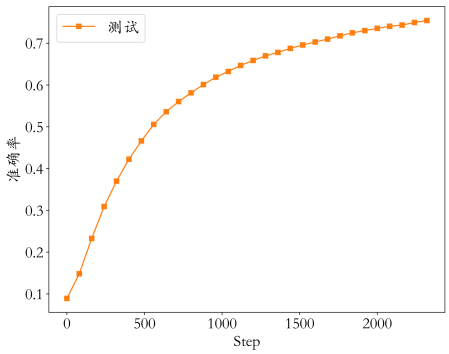

Figure 5.8 MNIST testing accuracy