Artificial Intelligence Technology and Application

Deep Learning Lab Guide

Student Version

Huawei Technologies CO., LTD.

Contents

1 TensorFlow 2 Basics

1.1 Introduction

1.1.1 About This Lab

This experiment helps trainees understand the basic syntax of TensorFlow 2 by introducing a series of tensor operations of TensorFlow 2, including tensor creation, slicing, and indexing, tensor dimension modification, tensor arithmetic operations, and tensor sorting.

1.1.2 Objectives

Upon completion of this experiment, you will be able to:

- Understand how to create a tensor.
- Master the tensor slicing and indexing methods.
- Master the syntax for tensor dimension modification.
- Master arithmetic operations of tensors.
- Master the tensor sorting method.
- Dive deeper into Eager Execution and AutoGraph based on code.

1.2 Procedure

1.2.1 Tensors

In TensorFlow, tensors are classified into constant tensors and variable tensors:

- A defined constant tensor has an unchangeable value and dimension, and a defined variable tensor has a changeable value and an unchangeable dimension.
- In neural networks, variable tensors are generally used as matrices for storing weights and other information, and are a type of trainable data. Constant tensors can be used as variables for storing hyperparameters or other structured data.

1.2.1.1 Importing TensorFlow

```
```

1.2.1.2 Tensor Creation

1.2.1.2.1 Creating a Constant Tensor

Common methods for creating a constant tensor include:

- **tf.constant()**: creates a constant tensor.
- **tf.zeros(), tf.zeros_like(), tf.ones(),** and **tf.ones_like()**: creates an all-zero or all-one constant tensor.
- **tf.fill()**: creates a tensor with a user-defined value.
- **tf.random**: creates a tensor with a known distribution.
- Create a list object by using NumPy, and then convert the list object into a tensor by using **tf.convert_to_tensor**.

Step 1 tf.constant()

tf.constant(value, dtype=None, shape=None, name='Const', verify_shape=False):

- **value**: value
- **dtype**: data type
- **shape**: tensor type
- **name**: constant name
- **verify_shape**: Boolean value, used to verify the shape of a value. The default value is **False**. If **verify_shape** is set to **True**, the system checks whether the shape of a value is consistent with the **shape**. If they are inconsistent, the system reports an error.

```
```

Output:

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

Code:

```
```

Output:

```
Value of the constant const_a: [[1. 2.] [3. 4.]]
Data type of the constant const_a: <dtype: 'float32'>
Shape of the constant const_a: (2, 2)
Name of the device that is to generate the constant const_a:
/job:localhost/replica:0/task:0/device:CPU:0
```

Step 2 tf.zeros(), tf.zeros_like(), tf.ones(), and tf.ones_like()

Usages of **tf.ones()** and **tf.ones_like()** are similar to those of **tf.zeros()** and **tf.zeros_like()**. Therefore, the following describes only the usages of **tf.ones()** and **tf.ones_like()**.

Create a constant with the value **0**. **tf.zeros(shape, dtype=tf.float32, name=None):**

- **shape**: tensor shape.
- **dtype**: data type.
- **name**: specifies the template name.

Code:

```
```

Create a tensor whose value is **0** based on the input tensor, with its shape being the same as that of the input tensor:

tf.zeros_like(input_tensor, dtype=None, name=None, optimize=True):

- **input_tensor**: tensor.
- **dtype**: data type.
- **name**: tensor name.
- **optimize**: indicates whether optimization is enabled.

Code:

```
```

Output:

```
array([[0., 0.], [0., 0.]], dtype=float32)
```

Step 3 tf.fill()

Create a tensor and fill it with a specific value. **tf.fill(dims, value, name=None)**:

- **dims**: tensor shape, which is the same as **shape** above.
- **value**: tensor value.
- **name**: tensor name.

Code:

```
```

Power output

```
array([[8, 8, 8], [8, 8, 8], [8, 8, 8]], dtype=int32)
```

Step 4 tf.random

This module is used to generate a tensor with a specific distribution. Common methods in this module include **tf.random.uniform()**, **tf.random.normal()**, and **tf.random.shuffle()**. The following describes how to use **tf.random.normal()**.

Create a tensor that conforms to a normal distribution. **tf.random.normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32,seed=None, name=None)**:

- **shape**: data shape.
- **mean**: mean value with a Gaussian distribution.
- **stddev**: standard deviation with a Gaussian distribution.
- **dtype**: data type.
- **seed**: random seed.
- **name**: tensor name.

Code:

```
```

output:

```
array([[-0.8521641 , 2.0672443 , -0.94127315, 1.7840577 , 2.9919195 ], [-0.8644102 , 0.41812655, -0.85865736, 1.0617154 , 1.0575105 ], [ 0.22457163, -0.02204755, 0.5084496 , -0.09113179, -1.3036906 ], [-1.1108295 , -0.24195422, 2.8516252 , -0.7503834 , 0.1267275 ], [ 0.9460202 , 0.12648873, -2.6540542 , 0.0853276 , 0.01731399]], dtype=float32)
```

Step 5 Creating a list object by using NumPy, and then convert the list object into a tensor by using **tf.convert_to_tensor**.

This method can convert a given value into a tensor. **tf.convert_to_tensor** can be used to convert a Python data type into a tensor data type available to TensorFlow.

tf.convert_to_tensor(value,dtype=None,dtype_hint=None,name=None):

- **value**: value to be converted.
- **dtype**: data type of the tensor.
- **dtype_hint**: optional element type for the returned tensor, used when **dtype** is set to **None**. In some cases, a caller may not consider **dtype** when calling **tf.convert_to_tensor**. Therefore, **dtype_hint** can be used as a preference.

Code:

```
```

Output:

```
list
```

Code:

```
```

Output:

```
<tf.Tensor: shape=(6,), dtype=float32, numpy=array([1., 2., 3., 4., 5., 6.], dtype=float32)>
```

----End

1.2.1.2.2 Creating a Variable Tensor

In TensorFlow, variables are operated by using the **tf.Variable** class. **tf.Variable** indicates a tensor. The value of **tf.Variable** can be changed by running an arithmetic operation on **tf.Variable**. Variable values can be read and changed.

Code:

```
```

Output:

```
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 1., 1.],
  [1., 1., 1.]], dtype=float32)>
```

Code:

```
```

Output:

```
Value of the variable var_1:
tf.Tensor( [[1. 1. 1.] [1. 1. 1.]], shape=(2, 3), dtype=float32)
Value of the variable var_1 after the assignment: tf.Tensor( [[1. 2. 3.] [4. 5. 6.]], shape=(2, 3), dtype=float32)
```

Code:

```
```

Output:

```
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[2., 3., 4.],
  [5., 6., 7.]], dtype=float32)>
```

1.2.1.3 Tensor Slicing and Indexing

1.2.1.3.1 Slicing

Tensor slicing methods include:

- [start: End]: extracts a data slice from the start position to the end position of the tensor.
- [start:end:step] or [::step]: extracts a data slice at an interval of **step** from the start position to the end position of the tensor.
- [::-1]: slices data from the last element.
- '...': indicates a data slice of any length.

Code:

```
```

Output:

```
<tf.Tensor: shape=(4, 100, 100, 3), dtype=float32, numpy=
array([[[[ 1.68444023e-01, -7.46562362e-01, -4.34964240e-01],
  [-4.69263226e-01, 6.26460612e-01, 1.21065331e+00],
  [ 7.21675277e-01, 4.61057723e-01, -9.20868576e-01],
  ...,
```

Code:

```
```

Output:

```
<tf.Tensor: shape=(100, 100, 3), dtype=float32, numpy=
array([[[ 1.68444023e-01, -7.46562362e-01, -4.34964240e-01],
  [-4.69263226e-01, 6.26460612e-01, 1.21065331e+00],
  [ 7.21675277e-01, 4.61057723e-01, -9.20868576e-01],
  ...,
```

Code:

```
```

Output:

```
<tf.Tensor: shape=(2, 100, 100, 3), dtype=float32, numpy=
array([[[[ 1.68444023e-01, -7.46562362e-01, -4.34964240e-01],
  [-4.69263226e-01, 6.26460612e-01, 1.21065331e+00],
  [ 7.21675277e-01, 4.61057723e-01, -9.20868576e-01],
  ...,
```

Code:

```
```

Output:

```
<tf.Tensor: shape=(4, 100, 100, 3), dtype=float32, numpy=
array([[[[-1.70684665e-01, 1.52386248e+00, -1.91677585e-01],
  [-1.78917408e+00, -7.48436213e-01, 6.10363662e-01],
  [ 7.64770031e-01, 6.06725179e-02, 1.32704067e+00],
  ...,
```

1.2.1.3.2 Indexing

The basic format of an index is as follows: a[d1][d2][d3].

Code:

```

```

Output:

```
<tf.Tensor: shape=(), dtype=float32, numpy=0.38231283>
```

If the indexes of data to be extracted are nonconsecutive, **tf.gather** and **tf.gather_nd** are commonly used for data extraction in TensorFlow.

To extract data from a particular dimension:

tf.gather(params, indices,axis=None):

- **params**: input tensor.
- **indices**: index of the data to be extracted.
- **axis**: dimension of the data to be extracted.

Code:

```

```

Output:

```
<tf.Tensor: shape=(3, 100, 100, 3), dtype=float32, numpy=
array([[[[ 1.68444023e-01, -7.46562362e-01, -4.34964240e-01],
  [-4.69263226e-01, 6.26460612e-01, 1.21065331e+00],
  [ 7.21675277e-01, 4.61057723e-01, -9.20868576e-01],
  ...,
```

tf.gather_nd allows indexes to be performed on multiple dimensions: **tf.gather_nd(params,indices)**:

- **params**: input tensor.
- **indices**: index of the data to be extracted, which is generally a multidimensional list.

Code:

```

```

Output:

```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.5705869, 0.9735735], dtype=float32)>
```

1.2.1.4 Tensor Dimension Modification

1.2.1.4.1 Viewing

Code:

```

```

Output:

```
(2, 2) (2, 2) tf.Tensor([2 2], shape=(2,), dtype=int32)
```

As described above, **.shape** and **.get_shape()** return TensorShape objects, and **tf.shape(x)** returns Tensor objects.

1.2.1.4.2 Reshaping

tf.reshape(tensor,shape,name=None):

- **tensor**: input tensor
- **shape**: dimension of the reshaped tensor

Code:

```
```

Output:

```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[1, 2],
  [3, 4],
  [5, 6]], dtype=int32)>
```

1.2.1.4.3 Expanding

tf.expand_dims(input,axis,name=None):

- **input**: input tensor
- **axis**: adds a dimension after the **axis** dimension. When the number of dimensions of the input data is D, the **axis** must fall in the range of [–(D + 1), D] (included). A negative value indicates adding a dimension in reverse order.

Code:

```
```

Output:

```
Size of the original data: (100, 100, 3). Add a dimension before the first dimension (axis=0): (1, 100, 100, 3). Add a dimension before the second dimension (axis=1): (100, 1, 100, 3). Add a dimension after the last dimension (axis=-1): (100, 100, 3, 1).
```

1.2.1.4.4 Squeezing

tf.squeeze(input,axis=None,name=None):

- **input**: input tensor
- **axis**: If **axis** is set to **1**, dimension 1 needs to be deleted.

Code:

```
```

Output:

```
Size of the original data: (1, 100, 100, 3) Data size after the dimension compression: (100, 100, 3)
```

1.2.1.4.5 Transposing

tf.transpose(a,perm=None,conjugate=False,name='transpose'):

- **a**: input tensor
- **perm**: tensor size sequence, generally used to transpose high-dimensional arrays
- **conjugate**: indicates complex number transpose.
- **name**: tensor name

Code:

```
```

Output:

```
Size of the original data: (2, 3) Data size after transposition: (3, 2)
```

Code:

```
```

Output:

```
Size of the original data: (4, 100, 200, 3) Data size after transposition:l (4, 200, 100, 3)
```

1.2.1.4.6 Broadcasting

broadcast_to is used to broadcast data from a low dimension to a high dimension.

tf.broadcast_to(input,shape,name=None):

- **input**: input tensor
- **shape**: size of the output tensor

Code:

```
```

Output:

```
Original data: [1 2 3 4 5 6] Data after broadcasting: [[1 2 3 4 5 6] [1 2 3 4 5 6] [1 2 3 4 5 6] [1 2 3 4 5 6]]
```

Code:

```
```

Output:

```
tf.Tensor( [[ 1 2 3] [11 12 13] [21 22 23] [31 32 33]], shape=(4, 3), dtype=int32)
```

1.2.1.5 Arithmetic Operations on Tensors

1.2.1.5.1 Arithmetic Operators

Main arithmetic operations include addition (**tf.add**), subtraction (**tf.subtract**), multiplication (**tf.multiply**), division (**tf.divide**), logarithm (**tf.math.log**), and powers (**tf.pow**). The following describes only one addition example.

Code:

```
```

Output:

```
tf.Tensor( [[ 4 11] [ 6 17]], shape=(2, 2), dtype=int32)
```

1.2.1.5.2 Matrix Multiplication

Matrix multiplication is implemented by calling **tf.matmul**.

Code:

```
```

Output:

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[13, 63],
  [20, 96]], dtype=int32)>
```

1.2.1.5.3 Tensor Statistics Collection

Methods for collecting tensor statistics include:

- **tf.reduce_min/max/mean()**: calculates the minimum, maximum, and average values.
- **tf.argmax()/tf.argmin()**: calculates the positions of the maximum and minimum values.
- **tf.equal()**: checks whether two tensors are equal by element.
- **tf.unique()**: removes duplicate elements from tensors.
- **tf.nn.in_top_k(prediction, target, K)**: calculates whether the predicted value is equal to the actual value, and returns a Boolean tensor.

The following describes how to use **tf.argmax()**: Return the position of the maximum value.

- **tf.argmax(input,axis)**
- **input**: input tensor
- **axis**: maximum output value in the axis dimension

Code:

```
```

Output:

```
Input tensor: [[1 3 2] [2 5 8] [7 5 9]] Searches for the position of the maximum value by column: [2 1 2] Search for the position of the maximum value by row: [1 2 2]
```

1.2.1.6 Dimension-based Arithmetic Operations

In TensorFlow, a series of operations of **tf.reduce_*** reduce tensor dimensions. The series of operations can be performed on dimensional elements of a tensor, for example, calculating the mean value by row and calculating a product of all elements in the tensor. Common operations include **tf.reduce_sum** (addition), **tf.reduce_prod** (multiplication), **tf.reduce_min** (minimum), **tf.reduce_max** (maximum), **tf.reduce_mean** (mean value)., **tf.reduce_all** (logical AND), **tf.reduce_any** (logical OR), and **tf.reduce_logsumexp** (log(sum(exp))).

The methods for using these operations are similar. The following describes how to use **tf.reduce_sum**. Calculate the sum of elements in each dimension of a tensor (**tf.reduce_sum(input_tensor, axis=None, keepdims=False,name=None)**):

- **input_tensor**: input tensor.
- **axis**: specifies axis to be calculated. If this parameter is not specified, the mean value of all elements is calculated.
- **keepdims**: specifies whether to keep dimensions. If this parameter is set to **True**, the output result retains the shape of the input tensor. If this parameter is set to **False**, dimensions of the output result decrease.
- **name**: operation name.

Code:

```

```

Output:

```
Raw data [[1 2 3] [4 5 6]. Calculate the sum (axis=None) of all elements in the tensor: 21. Calculate the sum (axis=0) of each column by row: 5 7 9]. Calculate the sum (axis=0) of each column by column: [6 15].
```

1.2.1.7 Tensor Concatenation and Splitting

1.2.1.7.1 Tensor Concatenation

In TensorFlow, tensor concatenation operations include:

- **tf.contact()**: concatenates vectors based on the specified dimension, while keeping other dimensions unchanged.
- **tf.stack()**: changes a group of R dimensional tensors to R+1 dimensional tensors, with the dimensions changed after the concatenation.

tf.concat(values, axis, name='concat'):

- **tf.concat(values, axis, name='concat')**:
- **axis**: dimension to concatenate
- **name**: operation name

Code:

```
```

Output:

```
Sizes of the original data: (4, 100, 100, 3) (40, 100, 100, 3) Size of the data after concatenation: (44, 100, 100, 3)
```

A dimension is added to the original matrix. In the same way, the parameter **axis** determines the position of the dimension. **tf.stack(values, axis=0, name='stack')**:

- **values**: input tensors, a group of tensors with the same shape and data type.
- **axis**: dimension to concatenate.
- **name**: operation name.

Code:

```
```

Output:

```
Sizes of the original data: (100, 100, 3) (100, 100, 3) Data size after concatenation: (2, 100, 100, 3)
```

1.2.1.7.2 Tensor Splitting

In TensorFlow, tensor splitting operations include:

- **tf.unstack()**: splits a tensor by a specific dimension.
- **tf.split()**: splits a tensor into a specified number of sub tensors based on a specific dimension.

tf.split() is more flexible than **tf.unstack()**.

tf.unstack(value,num=None,axis=0,name='unstack'):

- **value**: input tensor
- **num**: indicates that a list containing **num** elements is output. The value of **num** must be the same as the number of elements in the specified dimension. This parameter can generally be ignored.
- **axis**: indicates the dimension based on which the tensor is split.
- **name**: operation name

Code:

```
```

Output:

```
[<tf.Tensor: shape=(100, 100, 3), dtype=float32, numpy=
array([[[ 0.0665694 , 0.7110351 , 1.907618 ],
  [ 0.84416866, 1.5470593 , -0.5084871 ],
  [-1.9480026 , -0.9899087 , -0.09975405],
  ...,
```

tf.split(value, num_or_size_splits, axis=0):

- **value**: input tensor
- **num_or_size_splits**: number of sub tensors
- **axis**: indicates the dimension based on which the tensor is split.

tf.split() splits a tensor in either of the following ways:

1. If the value of **num_or_size_splits** is an integer, the tensor is evenly split into sub tensors in the specified dimension (**axis** = D).
2. If the value of **num_or_size_splits** is a vector, the tensor is split into sub tensors based on the element value of the vector in the specified dimension (**axis** = D).

Code:

```
```

Output:

```
```

1.2.1.8 Tensor Sorting

In TensorFlow, tensor sorting operations include:

- **tf.sort()**: sorts tensors in ascending or descending order and returns the sorted tensors.
- **tf.argsort()**: sorts tensors in ascending or descending order, and returns tensor indexes.
- **tf.nn.top_k()**: returns the first k maximum values.

tf.sort/argsort(input, direction, axis):

- **input**: input tensor
- **direction**: sorting order, which can be set to **DESCENDING** (descending order) or **ASCENDING** (ascending order). The default value is **ASCENDING**.
- **axis**: sorting by the dimension specified by **axis**. The default value of **axis** is **–1**, indicating the last dimension.

Code:

```
```

Output:

```
Input tensor: [1 8 7 9 6 5 4 2 3 0] Tensors after sorting: [0 1 2 3 4 5 6 7 8 9] The indexes of the elements after sorting are as follows: [9 0 7 8 6 5 4 2 1 3]
```

tf.nn.top_k(input,K,sorted=TRUE):

- **input**: input tensor
- **K**: the first k values to be output and their indexes

- **sorted**: When **sorted** is set to **TRUE**, the tensor is sorted in ascending order. When **sorted** is set to **FALSE**, the tensor is sorted in descending order.

Return two tensors:

- **values**: k maximum values in each row.
- **indices**: positions of elements in the last dimension of the input tensor.

Code:

```
```

Output:

```
Input tensor: [1 8 7 9 6 5 4 2 3 0] The first five values in ascending order are as follows: [9 8 7 6 5]
the indexes of the first five values in ascending order: [3 1 2 4 5]
```

1.2.2 Eager Execution Mode of TensorFlow 2

Eager Execution mode:

The Eager Execution mode of TensorFlow is a type of imperative programming, which is the same as native Python. When you perform a particular operation, the system immediately returns a result.

Graph mode:

TensorFlow 1.0 adopts the Graph mode to first build a computational graph, enable a Session, and then feed actual data to obtain a result. In Eager Execution mode, code debugging is easier, but the code execution efficiency is lower. The following implements simple multiplication by using TensorFlow to compare the differences between the Eager Execution mode and the Graph mode.

Code:

```
```

Output:

```
tf.Tensor( [[4. 6.] [4. 6.]], shape=(2, 2), dtype=float32)
```

Code:

```
#Use the syntax of TensorFlow 1.x in TensorFlow 2. You can install the v1 compatibility package in TensorFlow 2.0 to inherit the TensorFlow 1.x code and disable the eager execution mode.

#Create a graph and define it as a computational graph.

#Enable the drawing function, and perform the multiplication operation to obtain data.
```

Output:

```
[[4. 6.] [4. 6.]]
```

Restart the kernel to restore TensorFlow 2.0 and enable the eager execution mode. Another advantage of the Eager Execution mode lies in availability of native Python functions, for example, following condition statement:

Code:

```
```

Output:

```
tf.Tensor(0.11304152, shape=(), dtype=float32)
```

Owing to eager execution, this dynamic control flow can generate a NumPy value extractable by the Tensor, without using operators such as **tf.cond** and **tf.while** provided in Graph mode.

1.2.3 TensorFlow2 AutoGraph

When being used to comment out a function, the decorator **tf.function** can be called like any other function. **tf.function** will be compiled into a graph, so that it can run more efficiently on a GPU or a TPU. In this case, the function becomes an operation in TensorFlow. The function can be directly called to output a return value. However, the function is executed in graph mode and intermediate variable values cannot be directly viewed.

Code:

```
```

Output:

```
Tensor("b:0", shape=(), dtype=float32)
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[1.4121541 , 1.1626956 , 1.2527422 ],
  [1.2903953 , 1.0956903 , 1.1309073 ],
  [1.1039395 , 0.92851776, 1.0752096 ]], dtype=float32)>
```

According to the command output, the value of **b** in the function cannot be viewed directly, but the return value can be viewed using **.numpy()**. The following compares the performance of the graph mode and eager execution mode by performing the same operation (computation of one LSTM layer).

Code:

```
#Use the timeit module to measure the execution time of a small code segment.
import timeit
#Create a convolutional layer.

#Use @tf.function to convert the operation into a graph.

#Compare the execution time of the two modes.

#Call timeit.timeit to measure the time required for executing the code 10 times.
```

Output:

```
Time required for running the operation at one CNN layer in eager execution mode:
18.26327505100926 Time required for running the operation at one CNN layer in graph mode:
6.740775318001397
```

The comparison shows that the code execution efficiency in graph mode is much higher. Therefore, the **@tf.function** function can be used to improve the code execution efficiency.

2 Common Modules of TensorFlow 2

2.1 Introduction

2.1.1 About This Lab

This section describes the common modules of TensorFlow 2, including:

- **tf.data**: performs operations on data sets, including reading data sets from the memory, reading CSV files, reading TFRecord files, and enhancing data.
- **tf.image**: processes images, including adjusting image brightness and saturation, converting image sizes, rotating images, and detecting edges.
- **tf.gfile**: implements operations on files, including reading, writing, and renaming files, and operating folders.
- **tf.keras**: a high-level API that is used to build and train deep learning models.
- **tf.distributions** and other modules.

This section focuses on the **tf.keras** module to lay a foundation for deep learning modeling.

2.1.2 Objectives

Upon completion of this experiment, you will be able to:
Master the common deep learning modeling interfaces in **tf.keras**.

2.2 Procedure

2.2.1 Model Building

2.2.1.1 Building a Model by Stacking (tf.keras.Sequential)

The most common way to build a model is to stack layers by using **tf.keras.Sequential**.

2.2.1.2 Building a Functional Model

Functional models are built mainly by using **tf.keras.Input** and **tf.keras.Model**, which are more complex than **tf.keras.Sequential** but have a good effect. Variables can be input at the same time or in different phases, and data can be output in different phases. Functional models are preferred if more than one model output is needed.

Model stacking (.Sequential) vs Functional model (Model):

The **tf.keras.Sequential** model is a simple stack of layers and cannot represent any model. You can use the Keras functional model to build complex model topologies such as:

- Multi-input models.
- Multi-output models.
- Models with shared layers.
- Models with non-sequential data flows (for example, residual connections).

Code:

```
#Use the output of the previous layer as the input of the next layer.

#Print model information.
```

Output:

```
Model: "model"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
input_1 (InputLayer) [(None, 32)] 0
_________________________________________________________________
dense_3 (Dense) (None, 32) 1056
_________________________________________________________________
dense_4 (Dense) (None, 32) 1056
_________________________________________________________________
dense_5 (Dense) (None, 10) 330
=================================================================
Total params: 2,442
Trainable params: 2,442
Non-trainable params: 0
_________________________________________________________________
```

2.2.1.3 Building a Network Layer (tf.keras.layers)

The **tf.keras.layers** module is used to configure neural network layers. Common classes include:

- **tf.keras.layers.Dense**: builds a fully connected layer.
- **tf.keras.layers.Conv2D**: builds a two-dimensional convolutional layer.
- **tf.keras.layers.MaxPooling2D/AveragePooling2D**: builds a maximum/average pooling layer.
- **tf.keras.layers.RNN**: builds a recurrent neural network layer.
- **tf.keras.layers.LSTM/tf.keras.layers.LSTMCell**: builds an LSTM network layer/LSTM unit.
- **tf.keras.layers.GRU/tf.keras.layers.GRUCell**: builds a GRU unit/GRU network layer.
- **tf.keras.layers.Embedding**: converts a positive integer (subscript) into a vector of a fixed size, for example, [[4],[20]]->[[0.25,0.1],[0.6,-0.2]]. The embedding layer can only be used as the first model layer.
- **tf.keras.layers.Dropout**: builds the dropout layer.
- **tf.keras.layers.MaxPooling2D/AveragePooling2D**.

- **tf.keras.layers.LSTM/tf.keras.layers.LSTMCell**.

Main network configuration parameters in **tf.keras.layers** include:

- **activation**: sets the activation function for the layer. By default, the system applies no activation function.
- **kernel_initializer** and **bias_initializer**: initialization schemes that create the layer's weights (kernel and bias). The default initializer is **Glorot_uniform**.
- **kernel_regularizer** and **bias_regularizer**: regularization schemes that apply to the layer's weights (kernel and bias), such as L1 or L2 regularization. By default, the system applies no regularization function.

2.2.1.3.1 tf.keras.layers.Dense

Main configuration parameters in **tf.keras.layers.Dense** include:

- **units**: number of neurons.
- **activation**: sets the activation function.
- **use_bias**: indicates whether to use bias terms. Bias terms are used by default.
- **kernel_initializer**: initialization schemes that create the layer's weight (kernel).
- **bias_initializer**: initialization schemes that create the layer's weight (bias).
- **kernel_regularizer**: regularization schemes that apply to the layer's weight (kernel).
- **bias_regularizer**: regularization schemes that apply to the layer's weight (bias).
- **activity_regularizer**: regular item applied to the output, a Regularizer object.
- **kernel_constraint**: a constraint applied to a weight.
- **bias_constraint**: a constraint applied to a weight.

Code:

```
#Create a fully connected layer that contains 32 neurons. Set the activation function to sigmoid.
#The activation parameter can be set to a function name string, for example, sigmoid or a function object, for example, tf.sigmoid. layers.Dense(32, activation='sigmoid') layers.Dense(32, activation=tf.sigmoid)
#Set kernel_initializer

#Set kernel_regularizer to L2 regularization
```

Output:

```
<TensorFlow.python.keras.layers.core.Dense at 0x130c519e8>
```

2.2.1.3.2 tf.keras.layers.Conv2D

Main configurable parameters in **tf.keras.layers.Conv2D** include:

- **filters**: number of convolution kernels (output dimensions).
- **kernel_size**: width and length of a convolution kernel.
- **strides**: convolution step.
- **padding**: zero padding policy.

- When **padding** is set to **valid**, only valid convolution is performed, that is, boundary data is not processed. When **padding** is set to **same**, the convolution result at the boundary is reserved, and consequently, the output shape is usually the same as the input shape.
- **activation**: sets the activation function.
- **data_format**: data format, set to **channels_first** or **channels_last**. For example, for a 128 x 128 RGB image, data is organized as (3, 128, 128) if the value is **channels_first**, and (128, 128, 3) if the value is **channels_last**. The default value of this parameter is the value set in **~/.keras/keras.json**. If the value is not set, the value is **channels_last**.
- Other parameters include **use_bias. kernel_initializer**, **bias_initializer**, **kernel_regularizer**,

bias_regularizer, **activity_regularizer**, **kernel_constraints**, and **bias_constraints**.

Code:

```
```

Output:

```
<TensorFlow.python.keras.layers.convolutional.Conv2D at 0x106c510f0>
```

2.2.1.3.3 tf.keras.layers.MaxPooling2D/AveragePooling2D

Main configurable parameters in **tf.keras.layers.MaxPooling2D/AveragePooling2D** include:

- **pool_size**: size of the pooled kernel. For example, if the matrix (2, 2) is used, the picture becomes half of the original length in both dimensions. If this parameter is set to an integer, the integer is the values of all dimensions.
- **strides**: step.
- Other parameters include **padding** and **data_format**.

Code:

```
```

Output:

```
<TensorFlow.python.keras.layers.pooling.MaxPooling2D at 0x132ce1f98>
```

2.2.1.3.4 tf.keras.layers.LSTM/tf.keras.layers.LSTMCell

Main configuration parameters in **tf.keras.layers.LSTM/tf.keras.layers.LSTMCell** include:

- **units**: output dimension
- **input_shape** (**timestep** and **input_dim**): **timestep** can be set to **None**, and **input_dim** indicates the input data dimension.
- **activation**: sets the activation function.
- **recurrent_activation**: activation function to use for the recurrent step.

- **return_sequences**: If the value is **True**, the system returns the full sequence. If the value is **False**, the system returns the output of the last cell in the output sequence.
- **return_state**: Boolean value, indicates whether to return the last state in addition to the output.
- **dropout**: float between 0 and 1, fraction of the neurons to drop for the linear transformation of the inputs.
- **recurrent_dropout**: float between 0 and 1, fraction of the neurons to drop for the linear transformation of the recurrent state.

Code:

```
#Sequences t1, t2, and t3
```

Output:

```
Output when return_sequences is set to True: [[[-0.0106758]
  [-0.02711176]
  [-0.04583194]]]
Output when return_sequences is set to False: [0.05914127].
```

LSTMcell is the implementation unit of the LSTM layer.

- LSTM is an LSTM network layer.
- LSTMcell is a single-step computing unit, that is, an LSTM UNIT.

```
#LSTMCell
```

2.2.2 Model Training and Evaluation

2.2.2.1 Compiling

After a model is built, you can call **compile** to configure the learning process of the model:

- **compile(optimizer='rmsprop', loss=None, metrics=None, loss_weights=None)**
- **optimizer**: optimizer
- **loss**: loss function, cross entropy for binary tasks and MSE for regression tasks
- **metrics**: model evaluation criteria during training and testing For example, **metrics** can be set to **['accuracy']**. To specify multiple evaluation criteria, set a dictionary, for example, set **metrics** to **{'output_a':'accuracy'}**.
- **loss_weights**: If the model has multiple task outputs, you need to specify a weight for each output when optimizing the global loss.

Code:

```
#Determine the optimizer (optimizer), loss function (loss), and model evaluation method (metrics).
```

2.2.2.2 Training

fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None, validation_steps=None):

- **x**: Input training data
- **y**: Target (labeled) data
- **batch_size**: Number of samples for each gradient update The default value is **32**.
- **epochs**: number of iteration rounds of the training model
- **verbose**: log display mode, set to **0**, **1**, or **2**. 0 = Not displayed, 1 = Progress bar, 2 = One line is displayed for each round.
- **callbacks**: callback function used during training
- **validation_split**: fraction of the training data to be used as validation data
- **validation_data**: validation set. This parameter will overwrite **validation_split**.
- **shuffle**: indicates whether to shuffle data before each round of iteration. This parameter is invalid when **steps_per_epoch** is not **None**.
- **initial_epoch**: epoch at which to start training (useful for resuming a previous training weight)
- **steps_per_epoch**: set to the dataset size or **batch_size**
- **validation_steps**: This parameter is valid only when **steps_per_epoch** is specified. Total number of steps (batches of samples) to validate before stopping.

Code:

```
```

Output:

```
Train on 1000 samples, validate on 200 samples
Epoch 1/10
1000/1000 [==============================] - 0s 488us/sample - loss: 12.6024 -
categorical_accuracy: 0.0960 - val_loss: 12.5787 - val_categorical_accuracy: 0.0850
Epoch 2/10
1000/1000 [==============================] - 0s 23us/sample - loss: 12.6007 -
categorical_accuracy: 0.0960 - val_loss: 12.5776 - val_categorical_accuracy: 0.0850
Epoch 3/10
1000/1000 [==============================] - 0s 31us/sample - loss: 12.6002 -
categorical_accuracy: 0.0960 - val_loss: 12.5771 - val_categorical_accuracy: 0.0850

Epoch 10/10
1000/1000 [==============================] - 0s 24us/sample - loss: 12.5972 -
categorical_accuracy: 0.0960 - val_loss: 12.5738 - val_categorical_accuracy: 0.0850
<TensorFlow.python.keras.callbacks.History at 0x130ab5518>
```

You can use **tf.data** to build training input pipelines for large datasets.

Code:

```

```

Output:

```
Train for 30 steps, validate for 3 steps
Epoch 1/10
30/30 [==============================] - 0s 15ms/step - loss: 12.6243 - categorical_accuracy:
0.0948 - val_loss: 12.3128 - val_categorical_accuracy: 0.0833
...
30/30 [==============================] - 0s 2ms/step - loss: 12.5797 - categorical_accuracy:
0.0951 - val_loss: 12.3067 - val_categorical_accuracy: 0.0833
<TensorFlow.python.keras.callbacks.History at 0x132ab48d0>
```

Callback function.

Code:

A callback function is an object passed to the model to customize and extend the model's behavior during training. We can write our own custom callbacks, or use

built-in functions in **tf.keras.callbacks**. Common built-in callback functions are as follows:

tf.keras.callbacks.ModelCheckpoint: periodically save models.

tf.keras.callbacks.LearningRateScheduler: dynamically changes the learning rate.

tf.keras.callbacks.EarlyStopping: stops the training in advance.

tf.keras.callbacks.TensorBoard: uses TensorBoard.

#Set hyperparameters.

```

```

Output:

```
Train on 1000 samples, validate on 200 samples
0
0.1
Epoch 1/10
1000/1000 [==============================] - 0s 155us/sample - loss: 12.7907 -
categorical_accuracy: 0.0920 - val_loss: 12.7285 - val_categorical_accuracy: 0.0750
1
0.1
Epoch 2/10
1000/1000 [==============================] - 0s 145us/sample - loss: 12.6756 -
categorical_accuracy: 0.0940 - val_loss: 12.8673 - val_categorical_accuracy: 0.0950
...
0.001
Epoch 10/10
1000/1000 [==============================] - 0s 134us/sample - loss: 12.3627 -
categorical_accuracy: 0.1020 - val_loss: 12.3451 - val_categorical_accuracy: 0.0900
<TensorFlow.python.keras.callbacks.History at 0x133d35438>
```

2.2.2.3 Evaluation and Prediction

Evaluation and prediction functions: **tf.keras.Model.evaluate** and **tf.keras.Model.predict**.

Code:

```
#Model evaluation
```

Output:

```
1000/1000 [==============================] - 0s 45us/sample - loss: 12.2881 -
categorical_accuracy: 0.0770
[12.288104843139648, 0.077]
```

Code:

```
#Model prediction
```

Output:

```
[[0.04431767 0.24562006 0.05260926 ... 0.1016549 0.13826898 0.15511878]
[0.06296062 0.12550288 0.07593573 ... 0.06219672 0.21190381 0.12361749]
[0.07203944 0.19570401 0.11178136 ... 0.05625525 0.20609994 0.13041474]
...
[0.09224506 0.09908539 0.13944311 ... 0.08630784 0.15009451 0.17172746]
[0.08499582 0.17338121 0.0804626 ... 0.04409525 0.27150458 0.07133815]
[0.05191234 0.11740112 0.08346355 ... 0.0842929 0.20141983 0.19982798]]
```

2.2.3 Model Saving and Restoration

2.2.3.1 Saving and Restoring an Entire Model

Code:

```

```

After a model is saved, you can find the corresponding weight file in the corresponding folder.

2.2.3.2 Saving and Loading Network Weights Only

If the weight name is suffixed with **.h5** or **.keras**, save the weight as an HDF5 file, or otherwise, save the weight as a TensorFlow Checkpoint file by default.

Code:

3 Image Recognition

3.1 Introduction

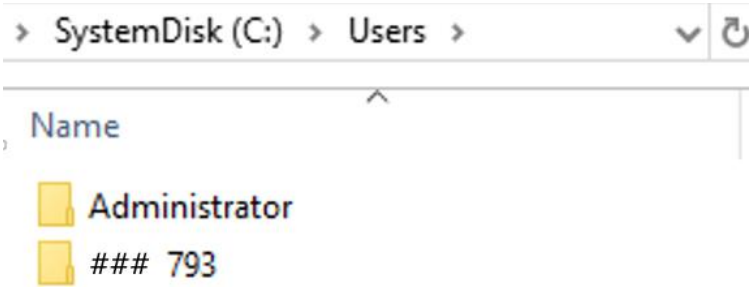
This experiment classifies images based on the **mnist_fashion** dataset.

3.2 Procedure

3.2.1 Importing the Dataset

Step 1 Place the downloaded dataset in a correct location.

In this example, place the **.keras** file in the folder named after the current user in the **users** directory of drive C.



The **.keras** file is placed in **###793**, which is the folder named after the current user.

Step 2 Load the dataset.

Import the **tensorflow** library and then load the dataset.

If the dataset is incorrectly stored, the **tensorflow** library will proactively load the dataset at a very low speed. In this case, consider accelerating the speed by using science-online tools. In addition, the dataset has been divided into the training set and test set. No additional division is required.

Step 3 View the number of classes.

In this example, **collections** in the standard library **python** is used to count the number of classes.

Ten classes are obtained and returned.

Step 4 View images.

Import the **matplotlib** library to view images.

----End

3.2.2 Preprocessing Data

Before writing neural networks correctly, preprocess the data so that the data complies with the input and output formats of the neural networks.

Step 1 Perform one_hot processing.

Perform one_hot processing on the **Target** data (results) of the dataset.

The parameter **num_classes=10** indicates that the dataset consists of 10 classes.

Step 2 Stitch samples in batches and classify the samples into small batches.

Map independent variables to dependent variables of the dataset, and then classify the samples into small batches to improve the gradient descent performance. In this example, 50 samples are classified to one group.

----End

3.2.3 Building a Neural Network

Step 1 Define an input layer.

Specify the number of neurons at the input layer. However, because the data is loaded in the matrix format at the beginning, the input layer is represented in the matrix format.

Step 2 Flatten the data.

Flatten the data to represent the data in the vector format.

Step 3 Define a middle layer (hidden layer).

In this example, 100 neurons are defined for each layer, and the ReLU activation function is used.

Step 4 Define an output layer.

As ten classes are specified, ten neurons are required at the output layer, and the softmax activation is used.

Step 5 Define the model sequence.

The overall model building process is complete so far. Afterwards, define the input and output sequences of the model.

Step 6 Define the loss function, optimizer, and training output information.

Specify the Adam optimizer, and set the learning rate to 0.001 and the loss function to the cross-entropy loss function.

Afterwards, output the accuracy rate of the current model after each round of training.

Step 7 View the model.

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28)]          0
_________________________________________________________________
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 100)               78500
_________________________________________________________________
dense_1 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_2 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_3 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1010
=================================================================
Total params: 109,810
Trainable params: 109,810
Non-trainable params: 0
```

View the declared model by checking the number of parameters to be trained at each layer, the number of parameters to be trained on the entire network, and the number of neurons at each layer.

Step 8 Train the model.

Perform ten rounds of training.

```
Train for 1200 steps
Epoch 1/10
1200/1200 [==============================] - 6s 5ms/step - loss: 1.0290 - accuracy: 0.7558
Epoch 2/10
1200/1200 [==============================] - 5s 5ms/step - loss: 0.4982 - accuracy: 0.8221
Epoch 3/10
1200/1200 [==============================] - 5s 4ms/step - loss: 0.4448 - accuracy: 0.8386
Epoch 4/10
1200/1200 [==============================] - 5s 4ms/step - loss: 0.4099 - accuracy: 0.8509
Epoch 5/10
1200/1200 [==============================] - 4s 4ms/step - loss: 0.3888 - accuracy: 0.8575
Epoch 6/10
1200/1200 [==============================] - 5s 4ms/step - loss: 0.3729 - accuracy: 0.8632
Epoch 7/10
1200/1200 [==============================] - 5s 4ms/step - loss: 0.3589 - accuracy: 0.8689
Epoch 8/10
1200/1200 [==============================] - 5s 4ms/step - loss: 0.3484 - accuracy: 0.8712
Epoch 9/10
1200/1200 [==============================] - 5s 4ms/step - loss: 0.3426 - accuracy: 0.8741
Epoch 10/10
1200/1200 [==============================] - 5s 4ms/step - loss: 0.3283 - accuracy: 0.8777

<tensorflow.python.keras.callbacks.History at 0x2547f1d3948>
```

The accuracy rate of the model reaches 87% after the training is complete.

Step 9 Evaluate the model by using the test set.

Perform prediction evaluation by using the test set to determine whether the model is appropriate.

```

```

```
200/200 [==============================] - 1s 3ms/step - loss: 0.4393 - accuracy: 0.8506

[0.43925349552184345, 0.8506]
```

According to the preceding figure, the accuracy rate is 85%, indicating that the model is not overfitted, but the fitting degree is low. In this case, more neurons, a deeper network, and more datasets can be used to improve the model performance. A convolutional neural network (CNN) can be used alternatively. Due to the computer performance, convolution is performed.

----End

3.2.4 Building a CNN

During CNN building, data preprocessing takes a relatively long time because black-and-white images are used (there is only one color channel). However, the dataset is not defined. Therefore, you need to proactively add a dimension.

```

```

Then, perform stitching and small-batch processing.

```

```

Step 1 Build an input layer.

In this example, single-channel gray images are used. However, convolutional input requires a multi-dimensional color channel for displaying the images.

Therefore, a dimension needs to be added for output. If no dimension is added for the dataset, an error is reported because the input format of the dataset is different from that of the model.

Step 2 Build a convolutional layer.

Each of the 30 convolution kernels is used for 5 x 5 convolution, and the SAME operation is performed on the images. This operation expands the images before reprocessing to ensure that the image scale is not reduced due to convolution. The ReLU activation function is used.

Step 3 Build a pooling layer.

A 2 x 2 max pooling operation is performed, followed by a convolution operation.

Step 4 Build a fully connected layer.

After the convolution is complete, flatten the neurons into vectors for subsequent classification, or continue to add a fully connected network layer.

Statistics show that the model accuracy rate will not increase on a CNN due to more fully connected layers or neurons. Therefore, no more fully connected layer is added to reduce the training time.

Step 5 Determine the model and optimize the loss function.

Determine the input and output positions of the model as before, and then specify the loss function and optimization method. Finally, view the model.

```
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_7 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 28, 28, 30)        780
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 28, 28, 30)        22530
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 30)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 14, 14, 30)        22530
_________________________________________________________________
flatten_6 (Flatten)          (None, 5880)              0
_________________________________________________________________
dense_17 (Dense)             (None, 10)                58810
=================================================================
Total params: 104,650
Trainable params: 104,650
Non-trainable params: 0
_________________________________________________________________
```

The preceding figure shows the input shape of each layer. Due to the SAME operation, no image scale is reduced during convolution.

Step 6 Train and test the model.

Only one round of training is performed as the CNN training is slow without being accelerated by a GPU.

Step 7 Save the model.

Step 8 Read the model.

----End