Huawei AI Certification Training

HCIA-AI

Mainstream Development Framework

Lab Guide

Issue: 3.0

Huawei Technologies Co., Ltd.

Huawei Technologies Co., Ltd.

Address:	Huawei Industrial Base Bantian, Longgang Shenzhen 518129 People's Republic of China
Website:	http://e.huawei.com

Huawei Certification System

Huawei Certification is an integral part of the company's "Platform + Ecosystem" strategy, it supports the ICT infrastructure featuring "Cloud-Pipe-Device". It evolves to reflect the latest trends of ICT development.

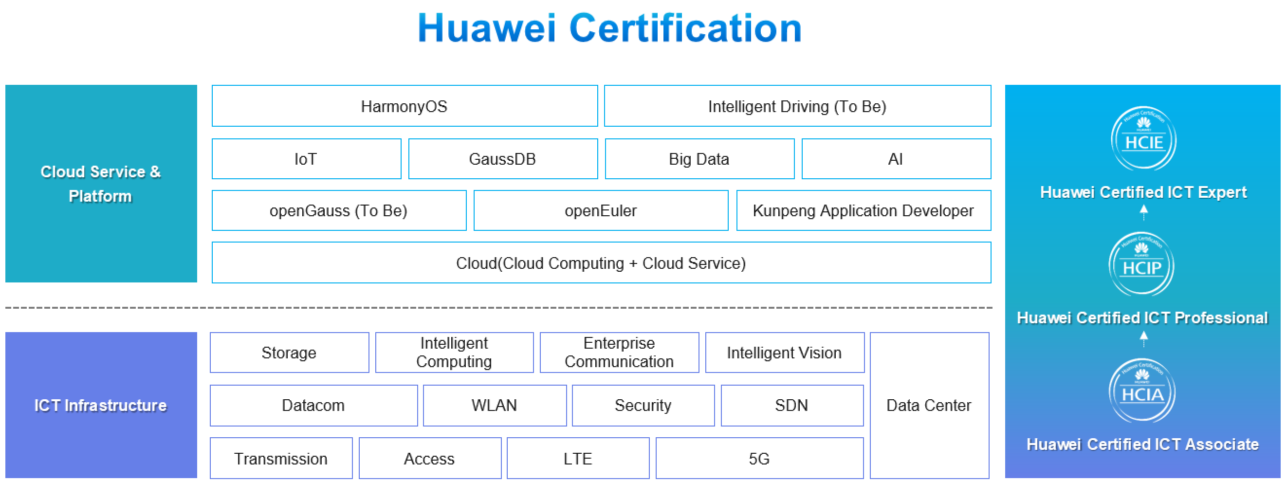
Huawei Certification consists of two categories: ICT Infrastructure, and Cloud Service & Platform. Huawei offers three levels of certification: Huawei Certified ICT Associate (HCIA), Huawei Certified ICT Professional (HCIP), and Huawei Certified ICT Expert (HCIE).

With its leading talent development system and certification standards, Huawei is committed to developing ICT professionals in the digital era, building a healthy ICT talent ecosystem.

HCIA-AI V3.0 certification is intended for cultivating and conducting qualification of engineers who are capable of creatively designing and developing AI products and solutions using machine learning and deep learning algorithms.

HCIA-AI V3.0 certified engineers understand the development history of AI, Huawei Ascend AI system, and Huawei full-stack AI strategy in all scenarios, master traditional machine learning and deep learning algorithms, and are able to use the TensorFlow and MindSpore frameworks to build, train, and deploy neural networks. With this certificate, you are qualified for positions including sales, marketing, product manager, project management, and technical support in the AI field.

Huawei Certification Portfolio



About This Document

Introduction

This document is intended for trainees who are preparing for the HCIA-AI certification examination or readers who want to learn AI basics and TensorFlow programming basics.

Description

This lab guide includes the following three exercises:

- Exercise 1 mainly introduces the basic syntax of TensorFlow 2.
- Exercise 2 introduces common modules of TensorFlow 2, especially the Keras API.
- Exercise 3 is a handwritten font image recognition exercise. It uses basic code to help learners understand how to recognize handwritten fonts using TensorFlow 2.

Background Knowledge Required

This course is a basic course for Huawei certification. Before beginning this course, you should:

- Have basic Python knowledge
- Be familiar with basic concepts of TensorFlow
- Understand basic Python programming knowledge

Contents

1 TensorFlow 2 Basics

1.1 Introduction

1.1.1 About This Exercise

This exercise introduces tensor operations of TensorFlow 2, including tensor creation, slicing, indexing, tensor dimension modification, tensor arithmetic operations, and tensor sorting, to help you understand the basic syntax of TensorFlow 2.

1.1.2 Objectives

- Learn how to create tensors.
- Learn how to slice and index tensors.
- Master the syntax of tensor dimension changes.
- Master arithmetic operations of tensors.
- Know how to sort tensors.
- Understand eager execution and AutoGraph based on code.

1.2 Tasks

1.2.1 Introduction to Tensors

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

- A defined constant tensor has an immutable value and dimension while a defined variable tensor has a variable value and an immutable dimension.
- In a neural network, a variable tensor is generally used as a matrix for storing weights and other information, and is a trainable data type. A constant tensor can be used as a variable for storing hyperparameters or other structural information.

1.2.1.1 Tensor Creation

1.2.1.1.1 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(),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.
- **tf.convert_to_tensor:** creates a list object by using NumPy and then converts it into a tensor.

Step 1 tf.constant()

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

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

Code:

```
import tensorflow as tf
print(tf.__version__)
const_a = tf.constant([[1, 2, 3, 4]],shape=[2,2], dtype=tf.float32) # Create a 2x2 matrix with values 1, 2, 3, and 4.
const_a
```

Output:

```
2.0.0-beta1
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[1., 2.],
            [3., 4.]], dtype=float32)>
```

Code:

```
# View common attributes.
print("value of const_a: ", const_a.numpy())
print("data type of const_a: ", const_a.dtype)
print("shape of const_a: ", const_a.shape)
print("device that const_a will be generated: ", const_a.device)
```

Output:

```
The value of const_a is [[1. 2.]
  [3. 4.]]
The data type of const_a is <dtype: 'float32'>.
The shape of const_a is (2, 2).
const_a will be generated on /job:localhost/replica:0/task:0/device:CPU:0.
```

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

The usage of **tf.ones()** and **tf.ones_like()** is similar to that of **tf.zeros()** and **tf.zeros_like()**. Therefore, the following describes how to use **tf.ones()** and **tf.ones_like()**.

Create a tensor with all elements set to zero.

tf.zeros(shape, dtype=tf.float32, name=None):

- **shape**: tensor shape
- **dtype**: type
- **name**: name for the operation

Code:

```
zeros_b = tf.zeros(shape=[2, 3], dtype=tf.int32) # Create a 2x3 matrix with all element values being 0.
```

Create a tensor with all elements set to zero 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**: type
- **name**: name for the operation
- **optimize**: optimize or not

Code:

```
zeros_like_c = tf.zeros_like(const_a)
# View generated data.
zeros_like_c.numpy()
```

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 the preceding shape
- **value**: tensor value
- **name**: name of the output

Code:

```
fill_d = tf.fill([3,3], 8) # 3x3 matrix with all element values being 8
# View data.
fill_d.numpy()
```

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. The common methods in this module include **tf.random.uniform()**, **tf.random.normal()**, and **tf.random.shuffle()**. The following demonstrates how to use **tf.random.normal()**.

Create a tensor that conforms to the 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 of Gaussian distribution
- **stddev**: standard deviation of Gaussian distribution
- **dtype**: data type
- **seed**: random seed
- **name**: name for the operation

Code:

```
random_e = tf.random.normal([5,5],mean=0,stddev=1.0, seed = 1)
# View the created data.
random_e.numpy()
```

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 Create a list object by using NumPy and then convert it into a tensor by using **tf.convert_to_tensor**.

This method can convert the given value to a tensor. It converts Python objects of various types to Tensor objects.

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

- **value**: value to be converted
- **dtype**: tensor data type
- **dtype_hint**: optional element type for the returned tensor, used when **dtype** is **None**. In some cases, a caller may not have a dtype in mind when converting to a tensor, so dtype_hint can be used as a soft preference.

Code:

```
# Create a list.
list_f = [1,2,3,4,5,6]
# View the data type.
type(list_f)
```

Output:

```
list
```

Code:

```
tensor_f = tf.convert_to_tensor(list_f, dtype=tf.float32)
tensor_f
```

Output:

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

1.2.1.1.2 Creating a Variable Tensor

In TensorFlow, variables are created and tracked via the **tf.Variable** class. A **tf.Variable** represents a tensor whose value can be changed by running ops on it. Specific ops allow you to read and modify the values of this tensor.

Code:

```
# To create a variable, provide an initial value.
var_1 = tf.Variable(tf.ones([2,3]))
var_1
```

Output:

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

Code:

```
# Read the variable value.
Print("value of var_1: ",var_1.read_value())
# Assign a new value to the variable.
var_value_1=[[1,2,3],[4,5,6]]
var_1.assign(var_value_1)
Print("new value for var_1: ",var_1.read_value())
```

Output:

```
Value of var_1: tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]], shape=(2, 3), dtype=float32)
New value for var_1: tf.Tensor(
[[1. 2. 3.]
 [4. 5. 6.]], shape=(2, 3), dtype=float32)
```

Code:

```
# Add a value to this variable.
var_1.assign_add(tf.ones([2,3]))
var_1
```

Output:

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

1.2.1.2 Tensor Slicing and Indexing

1.2.1.2.1 Slicing

Major slicing methods include:

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

Code:

```
# Create a 4-dimensional tensor. The tensor contains four images. The size of each image is 100 x 100 x 3.
tensor_h = tf.random.normal([4,100,100,3])
tensor_h
```

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:

```
# Extract the first image.
tensor_h[0,:,:,:]
```

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:

```
# Extract one slice every two images.
tensor_h[::2,...]
```

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:

```
# Slice data from the last element.
tensor_h[::-1]
```

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],
           ...,
```

1.2.1.2.2 Indexing

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

Code:

```
# Obtain the pixel in the [20,40] position in the second channel of the first image.
tensor_h[0][19][39][1]
```

Output:

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

If the indexes 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:

```
# Extract the first, second, and fourth images from tensor_h ([4,100,100,3]).
indices = [0,1,3]
tf.gather(tensor_h,axis=0,indices=indices)
```

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 data extraction from multiple dimensions:

tf.gather_nd(params,indices):

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

Code:

```
# Extract the pixel in [1,1] in the first dimension of the first image and pixel in [2,2] in the first dimension of the second image in tensot_h ([4,100,100,3]).
indices = [[0,1,1,0],[1,2,2,0]]
tf.gather_nd(tensor_h,indices=indices)
```

Output:

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

1.2.1.3 Tensor Dimension Modification

1.2.1.3.1 Dimension Display

Code:

```
const_d_1 = tf.constant([[1, 2, 3, 4]],shape=[2,2], dtype=tf.float32)
# Three common methods for displaying a dimension:
print(const_d_1.shape)
print(const_d_1.get_shape())
print(tf.shape(const_d_1))# The output is a tensor. The value of the tensor indicates the size of the tensor dimension to be displayed.
```

Output:

```

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

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

1.2.1.3.2 Dimension Reshaping

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

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

Code:

```
reshape_1 = tf.constant([[1,2,3],[4,5,6]])
print(reshape_1)
tf.reshape(reshape_1, (3,2))
```

Output:

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

1.2.1.3.3 Dimension Expansion

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

- **input**: input tensor

- **axis**: adds a dimension after the axis dimension. Given an input of D dimensions, **axis** must be in range [-(D+1), D] (inclusive). A negative value indicates the reverse order.

Code:

```
# Generate a 100 x 100 x 3 tensor to represent a 100 x 100 three-channel color image.
expand_sample_1 = tf.random.normal([100,100,3], seed=1)
print("original data size: ",expand_sample_1.shape)
Print("add a dimension (axis=0) before the first dimension: ",tf.expand_dims(expand_sample_1, axis=0).shape)
Print("add a dimension (axis=1) before the second dimension: ",tf.expand_dims(expand_sample_1, axis=1).shape)
Print("add a dimension (axis=-1) after the last dimension: ",tf.expand_dims(expand_sample_1, axis=-1).shape)
```

Output:

```
Original data size: (100, 100, 3)
Add a dimension (axis=0) before the first dimension: (1, 100, 100, 3)
Add a dimension (axis=1) before the second dimension: (100, 1, 100, 3)
Add a dimension (axis=-1) after the last dimension: (100, 100, 3, 1)
```

1.2.1.3.4 Dimension Squeezing

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

This method is used to remove dimensions of size 1 from the shape of a tensor.

- **input**: input tensor
- **axis**: If you don not want to remove all size 1 dimensions, remove specific size 1 dimensions by specifying **axis**.

Code:

```
# Generate a 100 x 100 x 3 tensor.
orig_sample_1 = tf.random.normal([1,100,100,3])
print("original data size: ",orig_sample_1.shape)
squeezed_sample_1 = tf.squeeze(orig_sample_1)
print("squeezed data size: ",squeezed_sample_1.shape)

# The dimension of 'squeeze_sample_2' is [1, 2, 1, 3, 1, 1].
squeeze_sample_2 = tf.random.normal([1, 2, 1, 3, 1, 1])
t_1 = tf.squeeze(squeeze_sample_2) # Dimensions of size 1 are removed.
print('t_1.shape:', t_1.shape)
# Remove a specific dimension:
# 't' is a tensor of shape [1, 2, 1, 3, 1, 1]
t_1_new = tf.squeeze(squeeze_sample_2, [2, 4])
print('t_1_new.shape:', t_1_new.shape)
```

```
Output:
Original data size: (1, 100, 100, 3)
Squeezed data size: (100, 100, 3)
t_1.shape: (2, 3)
t_1_new.shape: (1, 2, 3, 1)
```

1.2.1.3.5 Transpose

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

- **a**: input tensor
- **perm**: permutation of the dimensions of **a**, generally used to transpose high-dimensional arrays
- **conjugate**: conjugate transpose
- **name**: name for the operation

Code:

```
# Low-dimensional transposition is simple. Input the tensor to be transposed by calling tf.transpose.
trans_sample_1 = tf.constant([1,2,3,4,5,6],shape=[2,3])
print("original data size: ",trans_sample_1.shape)
transposed_sample_1 = tf.transpose(trans_sample_1)
print("transposed data size: ",transposed_sample_1.shape)
```

Output:

```
Original data size: (2, 3)
Transposed data size: (3, 2)
```

Code:

```
The perm parameter is required for transposing high-dimensional data. perm indicates the permutation of the
dimensions of the input tensor.
For a three-dimensional tensor, its original dimension permutation is [0, 1, 2] (perm), indicating the length, width,
and height of the high-dimensional data, respectively.
By changing the value sequence in perm, you can transpose the corresponding dimension of the data.
# Generate a 4 x 100 x 200 x 3 tensor to represent four 100 x 200 three-channel color images.
trans_sample_2 = tf.random.normal([4,100,200,3])
print("original data size: ",trans_sample_2.shape)
# Exchange the length and width of the four images. The value range of perm is changed from [0,1,2,3] to
[0,2,1,3].
transposed_sample_2 = tf.transpose(trans_sample_2,[0,2,1,3])
print("transposed data size: ",transposed_sample_2.shape)
```

Output:

```
Original data size: (4, 100, 200, 3)
Transposed data size: (4, 200, 100, 3)
```

1.2.1.3.6 Broadcast (broadcast_to)

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:

```
broadcast_sample_1 = tf.constant([1,2,3,4,5,6])
print("original data: ",broadcast_sample_1.numpy())
broadcasted_sample_1 = tf.broadcast_to(broadcast_sample_1,shape=[4,6])
```

```
print("broadcast data: ",broadcasted_sample_1.numpy())
```

Output:

```
Original data: [1 2 3 4 5 6]
Broadcast data: [[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:

```
# During the operation, if two arrays have different shapes, TensorFlow automatically triggers the broadcast mechanism as NumPy does.
a = tf.constant([[ 0, 0, 0],
                 [10,10,10],
                 [20,20,20],
                 [30,30,30]])
b = tf.constant([1,2,3])
print(a + b)
```

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

1.2.1.4 Arithmetic Operations on Tensors

1.2.1.4.1 Arithmetic Operators

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 is an example of addition.

Code:

```
a = tf.constant([[3, 5], [4, 8]])
b = tf.constant([[1, 6], [2, 9]])
print(tf.add(a, b))
```

Output:

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

1.2.1.4.2 Matrix Multiplication

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

Code:

```
tf.matmul(a,b)
```

Output:

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

1.2.1.4.3 Tensor Statistics Collection

Methods for collecting tensor statistics include:

- **tf.reduce_min/max/mean()**: calculates the minimum, maximum, and mean 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 a tensor.
- **tf.nn.in_top_k(prediction, target, K)**: calculates whether the predicted value is equal to the actual value and returns a tensor of the Boolean type.

The following demonstrates how to use **tf.argmax()**.

Return the subscript of the maximum value.

tf.argmax(input,axis):

- **input**: input tensor
- **axis**: The maximum value is output based on the axis dimension.

Code:

```
argmax_sample_1 = tf.constant([[1,3,2],[2,5,8],[7,5,9]])
print("input tensor: ",argmax_sample_1.numpy())
max_sample_1 = tf.argmax(argmax_sample_1, axis=0)
max_sample_2 = tf.argmax(argmax_sample_1, axis=1)
print("locate the maximum value by column: ",max_sample_1.numpy())
print("locate the maximum value by row: ",max_sample_2.numpy())
```

Output:

```
Input tensor: [1 3 2]
 [2 5 8]
 [7 5 9]]
Locate the maximum value by column: [2 1 2].
Locate the maximum value by row: [1 2 2].
```

1.2.1.5 Dimension-based Arithmetic Operations

In TensorFlow, operations such as **tf.reduce_*** reduce tensor dimensions. These operations can be performed on the dimension 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), **tf.reduce_all** (logical AND), **tf.reduce_any** (logical OR), and **tf.reduce_logsumexp** (**log(sum(exp))**).

The methods of using these operations are similar. The following uses the **tf.reduce_sum** operation as an example.

Compute the sum of elements across dimensions of a tensor.

tf.reduce_sum(input_tensor, axis=None, keepdims=False,name=None):

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

Code:

```
reduce_sample_1 = tf.constant([1,2,3,4,5,6],shape=[2,3])
print("original data",reduce_sample_1.numpy())
print("compute the sum of all elements in a tensor (axis=None):
",tf.reduce_sum(reduce_sample_1,axis=None).numpy())
print("compute the sum of each column by column (axis=0): ",tf.reduce_sum(reduce_sample_1,axis=0).numpy())
print("compute the sum of each column by row (axis=1): ",tf.reduce_sum(reduce_sample_1,axis=1).numpy())
```

Output:

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

1.2.1.6 Tensor Concatenation and Splitting

1.2.1.6.1 Tensor Concatenation

In TensorFlow, tensor concatenation operations include:

- **tf.contact()**: concatenates tensors along one dimension. Other dimensions remain unchanged.
- **tf.stack()**: stacks the tensor list of rank R into a tensor of rank (R+1). Dimensions are changed after stacking.

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

- **values**: input tensor
- **axis**: dimension along which to concatenate
- **name**: name for the operation

Code:

```
concat_sample_1 = tf.random.normal([4,100,100,3])
concat_sample_2 = tf.random.normal([40,100,100,3])
Print("original data size: ",concat_sample_1.shape,concat_sample_2.shape)
concated_sample_1 = tf.concat([concat_sample_1,concat_sample_2],axis=0)
print("concatenated data size: ",concated_sample_1.shape)
```

Output:

```
Original data size: (4, 100, 100, 3) (40, 100, 100, 3)
```

```
Concatenated data size: (44, 100, 100, 3)
```

A dimension is added to an original matrix in the same way. **axis** determines the position where the dimension is added.

tf.stack(values, axis=0, name='stack'):

- **values**: a list of tensor objects with the same shape and type
- **axis**: axis to stack along
- **name**: name for the operation

Code:

```
stack_sample_1 = tf.random.normal([100,100,3])
stack_sample_2 = tf.random.normal([100,100,3])
Print("original data size: ",stack_sample_1.shape, stack_sample_2.shape)
# Dimension addition after concatenating. If axis is set to 0, a dimension is added before the first dimension.
stacked_sample_1 = tf.stack([stack_sample_1, stack_sample_2],axis=0)
print("concatenated data size: ",stacked_sample_1.shape)
```

Output:

```
Original data size: (100, 100, 3) (100, 100, 3)
Concatenated data size: (2, 100, 100, 3)
```

1.2.1.6.2 Tensor Splitting

In TensorFlow, tensor splitting operations include:

- **tf.unstack()**: unpacks tensors along the specific dimension.
- **tf.split()**: splits a tensor into a list of sub tensors based on specific dimensions.

Compared with **tf.unstack()**, **tf.split()** is more flexible.

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

- **value**: input tensor
- **num**: outputs a list containing **num** elements. **num** must be equal to the number of elements in the specified dimension. Generally, this parameter is ignored.
- **axis**: axis to unstack along
- **name**: name for the operation

Code:

```
# Unpack data along the first dimension and output the unpacked data in a list.
tf.unstack(stacked_sample_1,axis=0)
```

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 splits
- **axis**: dimension along which to split

tf.split() can be split in either of the following ways:

1. If **num_or_size_splits** is an integer, the tensor is evenly split into several small tensors along the **axis=D** dimension.
2. If **num_or_size_splits** is a vector, the tensor is split into several smaller tensors based on the element values of the vector along the **axis=D** dimension.

Code:

```
import numpy as np
split_sample_1 = tf.random.normal([10,100,100,3])
print("original data size: ",split_sample_1.shape)
splited_sample_1 = tf.split(split_sample_1, num_or_size_splits=5,axis=0)
print("If m_or_size_splits is 5, the size of the split data is: ",np.shape(splited_sample_1))
splited_sample_2 = tf.split(split_sample_1, num_or_size_splits=[3,5,2],axis=0)
print("If num_or_size_splits is [3,5,2], the sizes of the split data are:",
        np.shape(splited_sample_2[0]),
        np.shape(splited_sample_2[1]),
        np.shape(splited_sample_2[2]))
```

Output:

```
Original data size: (10, 100, 100, 3)
If m_or_size_splits is 5, the size of the split data is (5, 2, 100, 100, 3).
If num_or_size_splits is [3,5,2], the sizes of the split data are (3, 100, 100, 3) (5, 100, 100, 3) (2, 100, 100, 3).
```

1.2.1.7 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 the indices.
- **tf.nn.top_k()**: returns the k largest values.

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

- **input**: input tensor
- **direction**: direction in which to sort the values. The value can be **DESCENDING** or **ASCENDING**. The default value is **ASCENDING**.
- **axis**: axis along which to sort The default value is **-1**, which sorts the last axis.

Code:

```
sort_sample_1 = tf.random.shuffle(tf.range(10))
print("input tensor: ",sort_sample_1.numpy())
sorted_sample_1 = tf.sort(sort_sample_1, direction="ASCENDING")
print("tensor sorted in ascending order: ",sorted_sample_1.numpy())
sorted_sample_2 = tf.argsort(sort_sample_1,direction="ASCENDING")
print("index of elements in ascending order: ",sorted_sample_2.numpy())
```

Output:

```
Input tensor: [1 8 7 9 6 5 4 2 3 0]
Tensor sorted in ascending order: [0 1 2 3 4 5 6 7 8 9]
Index of elements in ascending order: [9 0 7 8 6 5 4 2 1 3]
```

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

- **input**: input tensor
- **K**: k largest values to be output and their indices
- **sorted**: **sorted=TRUE** indicates in ascending order. **sorted=FALSE** indicates in descending order.

Two tensors are returned:

- **values**: k largest values in each row
- **indices**: indices of **values** within the last dimension of **input**

Code:

```
values, index = tf.nn.top_k(sort_sample_1,5)
print("input tensor: ",sort_sample_1.numpy())
print("k largest values in ascending order: ", values.numpy())
print("indices of the k largest values in ascending order: ", index.numpy())
```

Output:

```
Input tensor: [1 8 7 9 6 5 4 2 3 0]
The k largest values in ascending order: [9 8 7 6 5]
Indices of the k largest 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 the native Python. When you perform a particular operation, the system immediately returns a result.

Graph mode:

TensorFlow 1 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:

```
x = tf.ones((2, 2), dtype=tf.dtypes.float32)
y = tf.constant([[1, 2],
                 [3, 4]], dtype=tf.dtypes.float32)
z = tf.matmul(x, y)
print(z)
```

Output:

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

Code:

```
# Use the syntax of TensorFlow 1.x in TensorFlow 2.x. You can install the v1 compatibility package in TensorFlow 2
to inherit the TensorFlow 1.x code and disable the eager execution mode.
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()
# Create a graph and define it as a computational graph.
a = tf.ones((2, 2), dtype=tf.dtypes.float32)
b = tf.constant([[1, 2],
                 [3, 4]], dtype=tf.dtypes.float32)
c = tf.matmul(a, b)
# Start a session and perform the multiplication operation to obtain data.
with tf.Session() as sess:
    print(sess.run(c))
```

Output:

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

Restart the kernel to restore TensorFlow to version 2 and enable the eager execution mode. Another advantage of the eager execution mode lies in availability of native Python functions, such as the following condition statement:

Code:

```
import tensorflow as tf
thre_1 = tf.random.uniform([], 0, 1)
x = tf.reshape(tf.range(0, 4), [2, 2])
print(thre_1)
if thre_1.numpy() > 0.5:
    y = tf.matmul(x, x)
else:
    y = tf.add(x, x)
```

Output:

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

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

1.2.3 AutoGraph of TensorFlow 2

When used to comment out a function, the **tf.function** decorator 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 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 the intermediate variable values cannot be directly viewed.

Code:

```
@tf.function
def simple_nn_layer(w,x,b):
    print(b)
```

```
        return tf.nn.relu(tf.matmul(w, x)+b)


w = tf.random.uniform((3, 3))
x = tf.random.uniform((3, 3))
b = tf.constant(0.5, dtype='float32')

simple_nn_layer(w,x,b)
```

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 output result, the value of **b** in the function cannot be directly viewed, but the return value can be viewed using **.numpy()**.

The following compares the performance of the graph and eager execution modes by performing the same operation (LSTM computation of one layer).

Code:

```
# Use the timeit module to measure the execution time of a small code segment.
import timeit
# Create a convolutional layer.
CNN_cell = tf.keras.layers.Conv2D(filters=100,kernel_size=2,strides=(1,1))

# Use @tf.function to convert the operation into a graph.
@tf.function
def CNN_fn(image):
        return CNN_cell(image)

image = tf.zeros([100, 200, 200, 3])

# Compare the execution time of the two modes.
CNN_cell(image)
CNN_fn(image)
# Call timeit.timeit to measure the time required for executing the code 10 times.
print("time required for performing the computation of one convolutional neural network (CNN) layer in eager execution mode:", timeit.timeit(lambda: CNN_cell(image), number=10))
print("time required for performing the computation of one CNN layer in graph mode:", timeit.timeit(lambda: CNN_fn(image), number=10))
```

Output:

```
Time required for performing the computation of one CNN layer in eager execution mode: 18.26327505100926
Time required for performing the computation of 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** can be used to improve the code execution efficiency.

2 Common Modules of TensorFlow 2

2.1 Introduction

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

- **tf.data**: implements operations on datasets.

  These operations include reading datasets from the memory, reading CSV files, reading TFRecord files, and augmenting data.
- **tf.image**: implements image processing.

  These operations include image luminance transformation, saturation transformation, image size transformation, image rotation, and edge detection.
- **tf.gfile**: implements operations on files.

  These operations include reading, writing, and renaming files, and operating folders.
- **tf.keras**: a high-level API 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.

2.2 Objectives

Upon completion of this exercise, you will be able to master the common deep learning modeling interfaces of **tf.keras**.

2.3 Tasks

2.3.1 Model Building

2.3.1.1 Stacking a Model (tf.keras.Sequential)

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

Code:

```
import tensorflow as tf
print(tf.__version__)
print(tf.keras.__version__)
import tensorflow.keras.layers as layers
model = tf.keras.Sequential()
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Output:

```
2.0.0-beta1
2.2.4-tf
```

2.3.1.2 Building a Functional Model

Functional models are mainly built 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.

Stacked model (.Sequential) vs. functional model (.Model):
The **tf.keras.Sequential** model is a simple stack of layers that cannot represent arbitrary models. You can use the Keras functional API to build complex model topologies, for example:

- 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 current layer as the input of the next layer.
x = tf.keras.Input(shape=(32,))
h1 = layers.Dense(32, activation='relu')(x)
h2 = layers.Dense(32, activation='relu')(h1)
y = layers.Dense(10, activation='softmax')(h2)
model_sample_2 = tf.keras.models.Model(x, y)

# Print model information.
model_sample_2.summary()
```

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
_________________________________________________________________
```

2.3.1.3 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, converts [[4],[20]] into [[0.25,0.1],[0.6,-0.2]]. The embedding layer can be used only as the first model layer.
- **tf.keras.layers.Dropout**: builds the dropout layer.

The following describes **tf.keras.layers.Dense**, **tf.keras.layers.Conv2D**, **tf.keras.layers.MaxPooling2D/AveragePooling2D**, and **tf.keras.layers.LSTM/tf.keras.layers.LSTMCell**.

The main network configuration parameters in **tf.keras.layers** include:

- **activation**: sets the activation function of a layer. By default, the system does not use any activation functions.
- **kernel_initializer** and **bias_initializer**: initialization schemes that create a layer's weights (kernel and bias). This defaults to the Glorot uniform initializer.
- **kernel_regularizer** and **bias_regularizer**: regularization schemes that apply to a layer's weights (kernel and bias), for example, L1 or L2 regularization. By default, the system does not use regularization functions.

2.3.1.3.1 tf.keras.layers.Dense

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

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

Code:

```
# Create a fully connected layer that contains 32 neurons and 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.
layers.Dense(32, kernel_initializer=tf.keras.initializers.he_normal)
# Set kernel_regularizer to the L2 regularization.
```

```
layers.Dense(32, kernel_regularizer=tf.keras.regularizers.l2(0.01))
```

Output:

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

2.3.1.3.2 tf.keras.layers.Conv2D

Main configuration 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 stride
- **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**: activation function
- **data_format**: data format. The value can be **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 as (128,128,3) if the value is **channels_last**. The default value of this parameter is the value defined in **~/.keras/keras.json**. If the value has never been set, the default 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:

```
layers.Conv2D(64,[1,1],2,padding='same',activation="relu")
```

Output:

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

2.3.1.3.3 tf.keras.layers.MaxPooling2D/AveragePooling2D

Main configuration 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 image becomes half of the original length in both dimensions. If this parameter is set to an integer, the integer is the value of all dimensions.
- **strides**: stride value.
- Other parameters include **padding** and **data_format**.

Code:

```
layers.MaxPooling2D(pool_size=(2,2),strides=(2,1))
```

Output:

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

2.3.1.3.4 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**, **input_dim**): **timestep** can be set to **None**, and **input_dim** indicates the input data dimensions.
- **activation**: activation function
- **recurrent_activation**: activation function to use for the recurrent step
- **return_sequences**: If the value is **True**, all sequences are returned. If the value is **False**, the output in the last cell of the output sequence is returned.
- **return_state**: Boolean value, indicating 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:

```
import numpy as np
inputs = tf.keras.Input(shape=(3, 1))
lstm = layers.LSTM(1, return_sequences=True)(inputs)
model_lstm_1 = tf.keras.models.Model(inputs=inputs, outputs=lstm)

inputs = tf.keras.Input(shape=(3, 1))
lstm = layers.LSTM(1, return_sequences=False)(inputs)
model_lstm_2 = tf.keras.models.Model(inputs=inputs, outputs=lstm)


# Sequences t1, t2, and t3
data = [[[0.1],
   [0.2],
   [0.3]]]
print(data)
print("output when return_sequences is set to True",model_lstm_1.predict(data))
print("output when return_sequences is set to False",model_lstm_2.predict(data))
```

Output:

```
[[[0.1], [0.2], [0.3]]]
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.

```
#LSTM
tf.keras.layers.LSTM(16, return_sequences=True)
```

```
#LSTMCell
x = tf.keras.Input((None, 3))
y = layers.RNN(layers.LSTMCell(16))(x)
model_lstm_3= tf.keras.Model(x, y)
```

2.3.2 Training and Evaluation

2.3.2.1 Model Compilation

After a model has been built, call **compile** to configure its learning process:

- 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, transfer 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:

```
model = tf.keras.Sequential()
model.add(layers.Dense(10, activation='softmax'))
# Determine the optimizer, loss function, and model evaluation method (metrics).
model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
                    loss=tf.keras.losses.categorical_crossentropy,
                    metrics=[tf.keras.metrics.categorical_accuracy])
```

2.3.2.2 Model 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 **32**.
- **epochs**: number of iteration rounds of the training model
- **verbose**: log display mode. The value can be **0**, **1**, or **2**.

  0 = no display

  1 = progress bar

  2 = one line for each round
- **callbacks**: callback function used during training
- **validation_split**: ratio of the validation dataset to the training data
- **validation_data**: validation dataset. This parameter overwrites **validation_split**.
- **shuffle**: 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**: Total number of steps (batches of samples) to be validated before stopping. This parameter is valid only when **steps_per_epoch** is specified.

Code:

```
import numpy as np

train_x = np.random.random((1000, 36))
train_y = np.random.random((1000, 10))

val_x = np.random.random((200, 36))
val_y = np.random.random((200, 10))

model.fit(train_x, train_y, epochs=10, batch_size=100,
              validation_data=(val_x, val_y))
```

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>
```

For large datasets, you can use **tf.data** to build training input pipelines.

Code:

```
dataset = tf.data.Dataset.from_tensor_slices((train_x, train_y))
dataset = dataset.batch(32)
dataset = dataset.repeat()
val_dataset = tf.data.Dataset.from_tensor_slices((val_x, val_y))
val_dataset = val_dataset.batch(32)
val_dataset = val_dataset.repeat()

model.fit(dataset, epochs=10, steps_per_epoch=30,
              validation_data=val_dataset, validation_steps=3)
```

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>
```

2.3.2.3 Callback Functions

A callback function is an object passed to the model to customize and extend the model's behavior during training. You can customize callback functions or use embedded functions in **tf.keras.callbacks**. Common embedded callback functions include:

tf.keras.callbacks.ModelCheckpoint: periodically saves models.
tf.keras.callbacks.LearningRateScheduler: dynamically changes the learning rate.
tf.keras.callbacks.EarlyStopping: stops the training in advance.
tf.keras.callbacks.TensorBoard: uses the TensorBoard.

Code:

```
import os
# Set hyperparameters.
Epochs = 10
logdir=os.path.join("logs")
if not os.path.exists(logdir):
    os.mkdir(logdir)
# Define a function for dynamically setting the learning rate.
def lr_Scheduler(epoch):
    if epoch > 0.9 * Epochs:
        lr = 0.0001
    elif epoch > 0.5 * Epochs:
        lr = 0.001
    elif epoch > 0.25 * Epochs:
        lr = 0.01
    else:
        lr = 0.1

    print(lr)
    return lr


callbacks = [
   # Early stopping:
    tf.keras.callbacks.EarlyStopping(
        # Metric for determining whether the model performance has no further improvement
        monitor='val_loss',
        # Threshold for determining whether the model performance has no further improvement
        min_delta=1e-2,
        # Number of epochs in which the model performance has no further improvement
        patience=2),

    # Periodically save models.
     tf.keras.callbacks.ModelCheckpoint(
        # Model path
        filepath='testmodel_{epoch}.h5',
        # Determine whether to save the optimal model.
        save_best_only=True,
        monitor='val_loss'),

    # Dynamically change the learning rate.
    tf.keras.callbacks.LearningRateScheduler(lr_Scheduler),

    # Use the TensorBoard.
    tf.keras.callbacks.TensorBoard(log_dir=logdir)
]
model.fit(train_x, train_y, batch_size=16, epochs=Epochs,callbacks=callbacks, validation_data=(val_x, val_y))
```

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>
```

2.3.2.4 Evaluation and Prediction

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

Code:

```
# Model evaluation
test_x = np.random.random((1000, 36))
test_y = np.random.random((1000, 10))
model.evaluate(test_x, test_y, batch_size=32)
```

Output:

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

Code:

```
# Model prediction
pre_x = np.random.random((10, 36))
result = model.predict(test_x,)
print(result)
```

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]]
```

2.3.3 Model Saving and Restoration

2.3.3.1 Saving and Restoring an Entire Model

Code:

```
import numpy as np
# Save models.
logdir='./model'
if not os.path.exists(logdir):
    os.mkdir(logdir)

model.save(logdir+'/the_save_model.h5')
# Import models.
new_model = tf.keras.models.load_model(logdir+'/the_save_model.h5')
new_prediction = new_model.predict(test_x)
#np.testing.assert_allclose: determines whether the similarity of two objects exceeds the specified tolerance. If yes, the system displays an exception.
#atol: specified tolerance
np.testing.assert_allclose(result, new_prediction, atol=1e-6) # Prediction results are the same.
```

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

2.3.3.2 Saving and Loading Network Weights Only

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

Code:

```
model.save_weights('./model/model_weights')
model.save_weights('./model/model_weights.h5')
# Load the weights.
model.load_weights('./model/model_weights')
model.load_weights('./model/model_weights.h5')
```

3 Handwritten Digit Recognition with TensorFlow

3.1 Introduction

Handwritten digit recognition is a common image recognition task where computers recognize digits in handwritten images. Different from printed fonts, handwriting of different people have different styles and sizes, making it difficult for computers to recognize handwriting. This exercise uses deep learning and TensorFlow tools to train a model using MNIST datasets.

This section describes the basic process of TensorFlow computing and basic elements for building a network.

3.2 Objectives

- Understand the basic TensorFlow calculation process;
- Be familiar with the basic elements for building a network, including dataset acquisition, network model building, model training, and model validation.

3.3 Tasks

- Read the MNIST handwritten digit dataset.
- Get started with TensorFlow by using simple mathematical models.
- Implement softmax regression by using high-level APIs.
- Build a multi-layer CNN.
- Implement a CNN by using high-level APIs.
- Visualize prediction results.

3.3.1 Project Description and Dataset Acquisition

3.3.1.1 Description

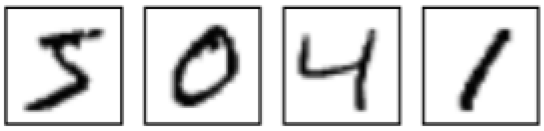
Handwritten digit recognition is a common image recognition task where computers recognize digits in handwritten images. Different from printed fonts, handwriting of different people have different styles and sizes, making it difficult for computers to recognize handwriting. This exercise uses deep learning and TensorFlow tools to train a model using MNIST datasets.

3.3.1.2 Data Acquisition and Processing

3.3.1.2.1 About the Dataset

1. The MNIST dataset is provided by the National Institute of Standards and Technology (NIST).

2. It consists of handwritten digits from 250 different individuals, of which 50% are high school students and 50% are staff from Bureau of the Census.
3. You can download the dataset from http://yann.lecun.com/exdb/mnist/, which consists of the following parts:
   - Training set images: train-images-idx3-ubyte.gz (9.9 MB, 47 MB after decompression, including 60,000 samples)
   - Training set labels: train-labels-idx1-ubyte.gz (29 KB, 60 KB after decompression, including 60,000 labels)
   - Test set images: t10k-images-idx3-ubyte.gz (1.6 MB, 7.8 MB after decompression, including 10,000 samples)
   - Test set labels: t10k-labels-idx1-ubyte.gz (5 KB, 10 KB after decompression, including 10,000 labels)
4. The MNIST is an entry-level computer vision dataset that contains images of various handwritten digits.



It also contains one label corresponding to each image to clarify the correct digit. For example, the labels for the preceding four images are 5, 0, 4, and 1.

3.3.1.2.2 MNIST Dataset Reading

Download the MNIST dataset from the official TensorFlow website and decompress it.

Code:

```
import os
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, optimizers, datasets
from matplotlib import pyplot as plt
import numpy as np

(x_train_raw, y_train_raw), (x_test_raw, y_test_raw) = datasets.mnist.load_data()

print(y_train_raw[0])
print(x_train_raw.shape, y_train_raw.shape)
print(x_test_raw.shape, y_test_raw.shape)

# Convert the labels into one-hot codes.
num_classes = 10
y_train = keras.utils.to_categorical(y_train_raw, num_classes)
y_test = keras.utils.to_categorical(y_test_raw, num_classes)
print(y_train[0])
```

Output:

```
5
(60000, 28, 28) (60000,)
(10000, 28, 28) (10000,)
```

```
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

In the MNIST dataset, the **images** is a tensor in the shape of [60000, 28, 28]. The first dimension is used to extract images, and the second and third dimensions are used to extract pixels in each image. Each element in this tensor indicates the strength of a pixel in an image. The value ranges from **0** to **255**.

The label data is converted from scalar to one-hot vectors. In a one-hot vector, one digit is 1 and digits in other dimensions are all 0s. For example, label 1 can be represented as [0,1,0,0,0,0,0,0,0,0,0]. Therefore, the labels are a digital matrix of [60000, 10].

3.3.2 Dataset Preprocessing and Visualization

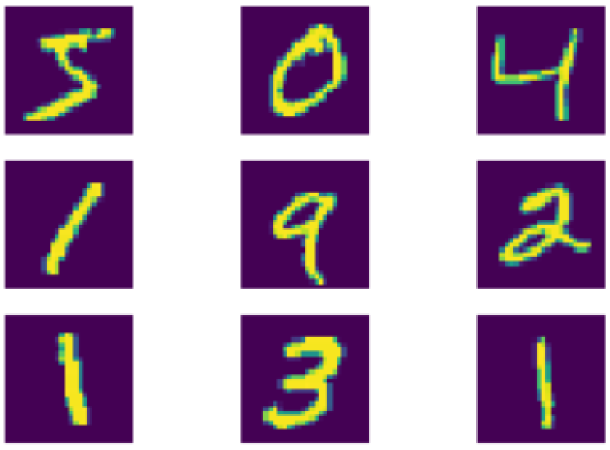
3.3.2.1 Data Visualization

Draw the first nine images.

Code:

```
plt.figure()
for i in range(9):
        plt.subplot(3,3,i+1)
        plt.imshow(x_train_raw[i])
        #plt.ylabel(y[i].numpy())
        plt.axis('off')
plt.show()
```

Output:



Data processing: The output of a fully connected network must be in the form of vector, instead of the matrix form of the current images. Therefore, you need to sort the images into vectors.

Code:

```
# Convert a 28 x 28 image to a 784 x 1 vector.
x_train = x_train_raw.reshape(60000, 784)
x_test = x_test_raw.reshape(10000, 784)
```

Currently, the dynamic range of pixels is 0 to 255. Image pixels are usually normalized to the range of 0 to 1 during processing of image pixel values.

Code:

```
Code:
 # Normalize image pixel values.
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255
```

3.3.3 DNN Construction

3.3.3.1 Building a DNN Model

Code:

```
# Create a deep neural network (DNN) model that consists of three fully connected layers and two ReLU activation functions.
model = keras.Sequential([
        layers.Dense(512, activation='relu', input_dim = 784),
        layers.Dense(256, activation='relu'),
        layers.Dense(124, activation='relu'),
layers.Dense(num_classes, activation='softmax')])

model.summary()
```

Output:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 512)               401920
_________________________________________________________________
dense_1 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_2 (Dense)              (None, 124)               31868
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1250
=================================================================
Total params: 566,366
Trainable params: 566,366
Non-trainable params: 0
_________________________________________________________________
```

layer.Dense() indicates a fully connected layer, and **activation** indicates a used activation function.

3.3.3.2 Compiling the DNN Model

Code:

```
Optimizer = optimizers.Adam(0.001)
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=Optimizer,
              metrics=['accuracy'])
```

In the preceding example, the loss function of the model is defined as cross entropy, and the optimization algorithm is the **Adam** gradient descent method.

3.3.3.3 Training the DNN Model

Code:

```
# Fit the training data to the model by using the fit method.
model.fit(x_train, y_train,
                    batch_size=128,
                    epochs=10,
                    verbose=1)
```

Output:

```
Epoch 1/10
60000/60000 [==============================] - 7s 114us/sample - loss: 0.2281 - acc: 0.9327s - loss: 0.2594 -
acc: 0. - ETA: 1s - loss: 0.2535 - acc: 0.9 - ETA: 1s - loss:
Epoch 2/10
60000/60000 [==============================] - 8s 129us/sample - loss: 0.0830 - acc: 0.9745s - loss: 0.0814 -
ac
Epoch 3/10
60000/60000 [==============================] - 8s 127us/sample - loss: 0.0553 - acc: 0.9822
Epoch 4/10
60000/60000 [==============================] - 7s 117us/sample - loss: 0.0397 - acc: 0.9874s - los
Epoch 5/10
60000/60000 [==============================] - 8s 129us/sample - loss: 0.0286 - acc: 0.9914
Epoch 6/10
60000/60000 [==============================] - 8s 136us/sample - loss: 0.0252 - acc: 0.9919
Epoch 7/10
60000/60000 [==============================] - 8s 129us/sample - loss: 0.0204 - acc: 0.9931s - lo
Epoch 8/10
60000/60000 [==============================] - 8s 135us/sample - loss: 0.0194 - acc: 0.9938
Epoch 9/10
60000/60000 [==============================] - 7s 109us/sample - loss: 0.0162 - acc: 0.9948
Epoch 10/10
60000/60000 [==============================] - ETA: 0s - loss: 0.0149 - acc: 0.994 - 7s 117us/sample - loss:
0.0148 - acc: 0.9948
```

epoch indicates a specific round of training. In the preceding example, full data is iterated for 10 times.

3.3.3.4 Evaluating the DNN Model

Code:

```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Output:

```
Test loss: 0.48341113169193267
Test accuracy: 0.8765
```

According to the evaluation, the model accuracy is 0.87 after 10 model training iterations.

3.3.3.5 Saving the DNN Model

Code:

```
logdir='./mnist_model'
if not os.path.exists(logdir):
    os.mkdir(logdir)
model.save(logdir+'/final_DNN_model.h5')
```

3.3.4 CNN Construction

The conventional CNN construction method helps you better understand the internal network structure but requires a large code volume. Therefore, attempts to construct a CNN by using high-level APIs are made to simplify the network construction process.

3.3.4.1 Building a CNN Model

Code:

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

model=keras.Sequential() # Create a network sequence.
## Add the first convolutional layer and pooling layer.
model.add(keras.layers.Conv2D(filters=32,kernel_size = 5,strides = (1,1),
                              padding = 'same',activation = tf.nn.relu,input_shape = (28,28,1)))
model.add(keras.layers.MaxPool2D(pool_size=(2,2), strides = (2,2), padding = 'valid'))
## Add the second convolutional layer and pooling layer.
model.add(keras.layers.Conv2D(filters=64,kernel_size = 3,strides = (1,1),padding = 'same',activation = tf.nn.relu))
model.add(keras.layers.MaxPool2D(pool_size=(2,2), strides = (2,2), padding = 'valid'))
## Add a dropout layer to reduce overfitting.
model.add(keras.layers.Dropout(0.25))
model.add(keras.layers.Flatten())
## Add two fully connected layers.
model.add(keras.layers.Dense(units=128,activation = tf.nn.relu))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(units=10,activation = tf.nn.softmax))
```

In the preceding network, two convolutional layers and pooling layers are first added by using **keras.layers**. Afterwards, a dropout layer is added to prevent overfitting. Finally, two full connection layers are added.

3.3.4.2 Compiling and Training the CNN Model

Code:

```
# Expand data dimensions to adapt to the CNN model.
X_train=x_train.reshape(60000,28,28,1)
X_test=x_test.reshape(10000,28,28,1)
model.compile(optimizer="adam",loss="categorical_crossentropy",metrics=['accuracy'])
model.fit(x=X_train,y=y_train,epochs=5,batch_size=128)
```

Output:

```
Epoch 1/5
55000/55000 [==============================] - 49s 899us/sample - loss: 0.2107 - acc: 0.9348
Epoch 2/5
55000/55000 [==============================] - 48s 877us/sample - loss: 0.0793 - acc: 0.9763
Epoch 3/5
55000/55000 [==============================] - 52s 938us/sample - loss: 0.0617 - acc: 0.9815
```

```
Epoch 4/5
55000/55000 [==============================] - 48s 867us/sample - loss: 0.0501 - acc: 0.9846
Epoch 5/5
55000/55000 [==============================] - 50s 901us/sample - loss: 0.0452 - acc: 0.9862

<tensorflow.python.keras.callbacks.History at 0x214bbf34ac8>
```

During training, the network training data is iterated for only five times. You can increase the number of network iterations to check the effect.

3.3.4.3 Evaluating the CNN Model

Code:

```
test_loss,test_acc=model.evaluate(x=X_test,y=y_test)
print("Test Accuracy %.2f"%test_acc)
```

Output:

```
10000/10000 [==============================] - 2s 185us/sample - loss: 0.0239 - acc: 0.9921
Test Accuracy 0.99
```

The evaluation shows that the accuracy of the CNN model reaches up to 99%.

3.3.4.4 Saving the CNN Model

Code:

```
logdir='./mnist_model'
if not os.path.exists(logdir):
        os.mkdir(logdir)
model.save(logdir+'/final_CNN_model.h5')
```

Output:

```
10000/10000 [==============================] - 5s 489us/sample - loss: 0.0263 - acc: 0.9920s - loss: 0.0273 -
ac
Test Accuracy 0.99
```

3.3.5 Prediction Result Visualization

3.3.5.1 Loading the CNN Model

Code:

```
from tensorflow.keras.models import load_model
new_model = load_model('./mnist_model/final_CNN_model.h5')
new_model.summary()
```

Output:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
```

```
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
dropout (Dropout)            (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense_4 (Dense)              (None, 128)               401536
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1290
=================================================================
Total params: 422,154
Trainable params: 422,154
Non-trainable params: 0
_________________________________________________________________
```

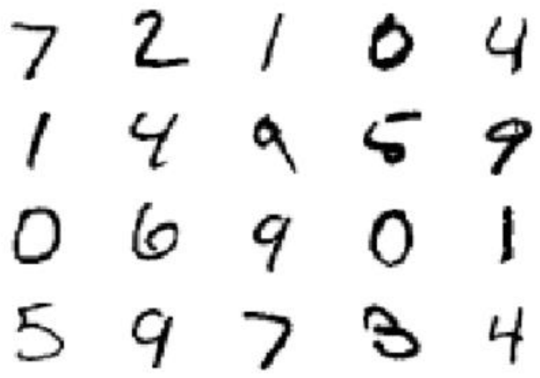
Visualize the prediction results.

Code:

```
# Visualize the test dataset output.
import matplotlib.pyplot as plt
%matplotlib inline
def res_Visual(n):
    final_opt_a=new_model.predict_classes(X_test[0:n])# Perform predictions on the test dataset by using the model.
    fig, ax = plt.subplots(nrows=int(n/5),ncols=5 )
    ax = ax.flatten()
    print('prediction results of the first {} images:'.format(n))
    for i in range(n):
        print(final_opt_a[i],end=',')
        if int((i+1)%5) ==0:
            print('\t')
       # Display images.
        img = X_test[i].reshape( (28,28))# Read each row of data in Ndarry format.
        plt.axis("off")
       ax[i].imshow(img, cmap='Greys', interpolation='nearest')# Visualization
        ax[i].axis("off")
    print('first {} images in the test dataset are in the following format:'.format(n))
res_Visual(20)
```

Output:

```
Prediction results of the first 20 images:
7,2,1,0,4,
1,4,9,5,9,
0,6,9,0,1,
5,9,7,3,4,
First 20 images in the test dataset:
```



4 Image Classification

4.1 Introduction

4.1.1 About This Exercise

This exercise is about a basic task in computer vision, that is, image recognition. The NumPy and TensorFlow modules are required. The NumPy module is used to create image objects, and the TensorFlow framework is used to create deep learning algorithms and build CNNs. This exercise recognizes image classes based on the CIFAR10 dataset.

4.1.2 Objectives

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

- Strengthen the understanding of the Keras-based neural network model building process.
- Master the method to load a pre-trained model.
- Learn how to use the checkpoint function.
- Learn how to use a trained model to make predictions.

4.2 Tasks

4.2.1 Importing Dependencies

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, optimizers, datasets, Sequential
from tensorflow.keras.layers import Conv2D,Activation,MaxPooling2D,Dropout,Flatten,Dense
import    os
import numpy as np
import matplotlib.pyplot as plt
```

4.2.2 Preprocessing Data

Code:

```
# Download the dataset.
(x_train,y_train), (x_test, y_test) = datasets.cifar10.load_data()
# Print the dataset size.
print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)
print(y_train[0])

# Convert the labels.
```

```
num_classes = 10
y_train_onehot = keras.utils.to_categorical(y_train, num_classes)
y_test_onehot = keras.utils.to_categorical(y_test, num_classes)
y_train[0]
```

Output:

```
(50000, 32, 32, 3) (50000, 1) (10000, 32, 32, 3) (10000, 1)
[6]

array([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.], dtype=float32)
```

Display nine sample images.

Code:

```
# Generate an image label list.
category_dict = {0:'airplane',1:'automobile',2:'bird',3:'cat',4:'deer',5:'dog',
                          6:'frog',7:'horse',8:'ship',9:'truck'}
# Display the first nine images and their labels.
plt.figure()
for i in range(9):
      plt.subplot(3,3,i+1)
      plt.imshow(x_train[i])
      plt.ylabel(category_dict[y_train[i][0]])
plt.show()
```

Output:

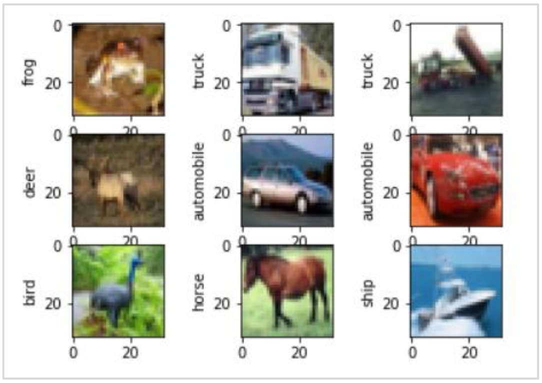


Figure 4-1 First nine images and their labels

Code:

```
# Pixel normalization
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255
```

4.2.3 Building a Model

Code:

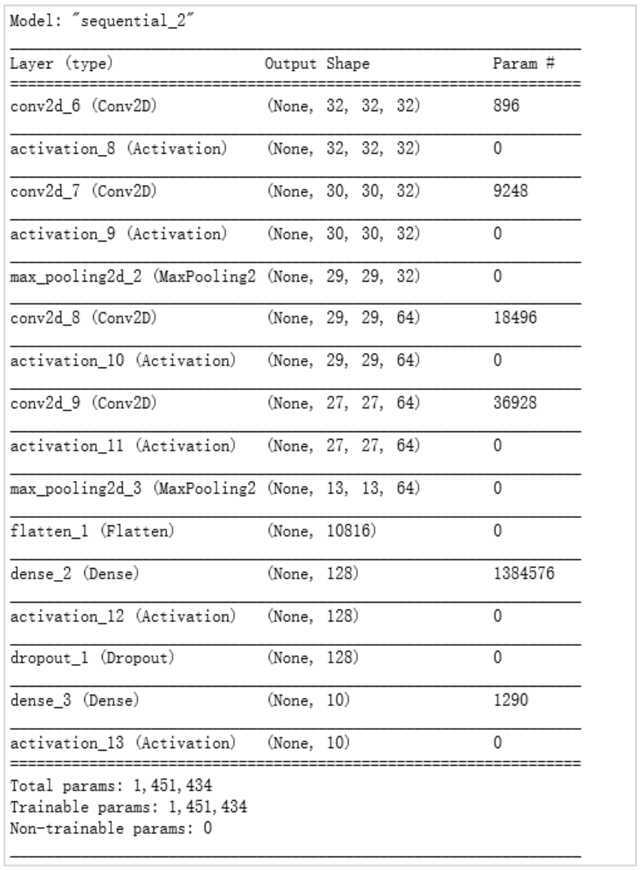
```
def CNN_classification_model(input_size = x_train.shape[1:]):
      model = Sequential()
```

```
    # The first two modules have two convolutional layers and one pooling layer.
    '''Conv1 with 32 3*3 kernels
        padding="same": it applies zero padding to the input image so that the input image gets fully covered by
the filter and specified stride.
        It is called SAME because, for stride 1, the output will be the same as the input.
        output: 32*32*32'''
    model.add(Conv2D(32, (3, 3), padding='same',
              input_shape=input_size))
    model.add(Activation('relu'))
    #Conv2
    model.add(Conv2D(32, (3, 3)))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2),strides =1))
    # The second module
    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(Activation('relu'))
    model.add(Conv2D(64, (3, 3)))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

   # Perform flattening before connecting to the fully connected network.
    model.add(Flatten())
    model.add(Dense(128))
    model.add(Activation('relu'))
   # The dropout layer parameter value ranges from 0 to 1.
    model.add(Dropout(0.25))
    model.add(Dense(num_classes))
    model.add(Activation('softmax'))
    opt = keras.optimizers.Adam(lr=0.0001)

    model.compile(loss='sparse_categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
return model
model=CNN_classification_model()
model.summary()
```

Output:



```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
activation_8 (Activation)    (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
activation_9 (Activation)    (None, 30, 30, 32)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 29, 29, 32)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 29, 29, 64)        18496
_________________________________________________________________
activation_10 (Activation)   (None, 29, 29, 64)        0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 27, 27, 64)        36928
_________________________________________________________________
activation_11 (Activation)   (None, 27, 27, 64)        0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 13, 13, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 10816)             0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               1384576
_________________________________________________________________
activation_12 (Activation)   (None, 128)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
_________________________________________________________________
activation_13 (Activation)   (None, 10)                0
=================================================================
Total params: 1,451,434
Trainable params: 1,451,434
Non-trainable params: 0
_________________________________________________________________
```

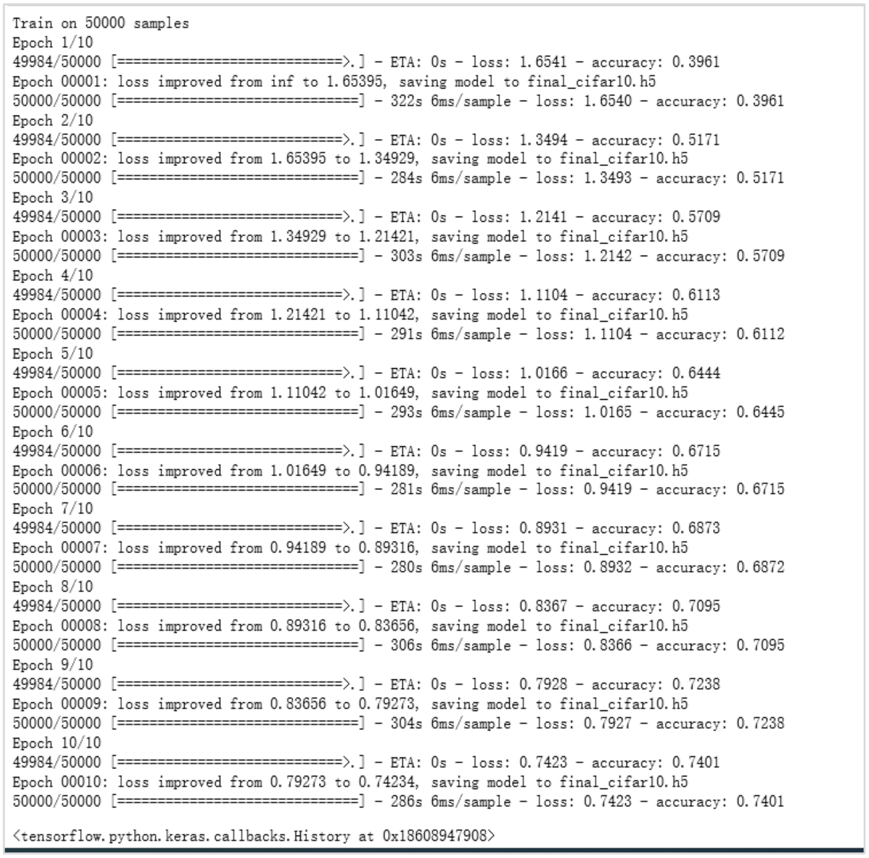
4.2.4 Training the Model

Code:

```
from tensorflow.keras.callbacks import ModelCheckpoint
model_name = "final_cifar10.h5"
model_checkpoint = ModelCheckpoint(model_name, monitor='loss',verbose=1, save_best_only=True)

# Load the pre-trained model.
trained_weights_path = 'cifar10_weights.h5'
if os.path.exists(trained_weights_path):
    model.load_weights(trained_weights_path, by_name =True)
# Training
model.fit(x_train,y_train, batch_size=32, epochs=10,callbacks = [model_checkpoint],verbose=1)
```

Output:



```
Train on 50000 samples
Epoch 1/10
49984/50000 [============================>.] - ETA: 0s - loss: 1.6541 - accuracy: 0.3961
Epoch 00001: loss improved from inf to 1.65395, saving model to final_cifar10.h5
50000/50000 [==============================] - 322s 6ms/sample - loss: 1.6540 - accuracy: 0.3961
Epoch 2/10
49984/50000 [============================>.] - ETA: 0s - loss: 1.3494 - accuracy: 0.5171
Epoch 00002: loss improved from 1.65395 to 1.34929, saving model to final_cifar10.h5
50000/50000 [==============================] - 284s 6ms/sample - loss: 1.3493 - accuracy: 0.5171
Epoch 3/10
49984/50000 [============================>.] - ETA: 0s - loss: 1.2141 - accuracy: 0.5709
Epoch 00003: loss improved from 1.34929 to 1.21421, saving model to final_cifar10.h5
50000/50000 [==============================] - 303s 6ms/sample - loss: 1.2142 - accuracy: 0.5709
Epoch 4/10
49984/50000 [============================>.] - ETA: 0s - loss: 1.1104 - accuracy: 0.6113
Epoch 00004: loss improved from 1.21421 to 1.11042, saving model to final_cifar10.h5
50000/50000 [==============================] - 291s 6ms/sample - loss: 1.1104 - accuracy: 0.6112
Epoch 5/10
49984/50000 [============================>.] - ETA: 0s - loss: 1.0166 - accuracy: 0.6444
Epoch 00005: loss improved from 1.11042 to 1.01649, saving model to final_cifar10.h5
50000/50000 [==============================] - 293s 6ms/sample - loss: 1.0165 - accuracy: 0.6445
Epoch 6/10
49984/50000 [============================>.] - ETA: 0s - loss: 0.9419 - accuracy: 0.6715
Epoch 00006: loss improved from 1.01649 to 0.94189, saving model to final_cifar10.h5
50000/50000 [==============================] - 281s 6ms/sample - loss: 0.9419 - accuracy: 0.6715
Epoch 7/10
49984/50000 [============================>.] - ETA: 0s - loss: 0.8931 - accuracy: 0.6873
Epoch 00007: loss improved from 0.94189 to 0.89316, saving model to final_cifar10.h5
50000/50000 [==============================] - 280s 6ms/sample - loss: 0.8932 - accuracy: 0.6872
Epoch 8/10
49984/50000 [============================>.] - ETA: 0s - loss: 0.8367 - accuracy: 0.7095
Epoch 00008: loss improved from 0.89316 to 0.83656, saving model to final_cifar10.h5
50000/50000 [==============================] - 306s 6ms/sample - loss: 0.8366 - accuracy: 0.7095
Epoch 9/10
49984/50000 [============================>.] - ETA: 0s - loss: 0.7928 - accuracy: 0.7238
Epoch 00009: loss improved from 0.83656 to 0.79273, saving model to final_cifar10.h5
50000/50000 [==============================] - 304s 6ms/sample - loss: 0.7927 - accuracy: 0.7238
Epoch 10/10
49984/50000 [============================>.] - ETA: 0s - loss: 0.7423 - accuracy: 0.7401
Epoch 00010: loss improved from 0.79273 to 0.74234, saving model to final_cifar10.h5
50000/50000 [==============================] - 286s 6ms/sample - loss: 0.7423 - accuracy: 0.7401

<tensorflow.python.keras.callbacks.History at 0x18608947908>
```

This exercise is performed on a laptop. The network in this exercise is simple, consisting of four convolutional layers. To improve the performance of this model, you can increase the number of epochs and the complexity of the model.

4.2.5 Evaluating the Model

Code:

```
new_model = CNN_classification_model()
new_model.load_weights('final_cifar10.h5')

model.evaluate(x_test, y_test, verbose=1)
```

Output:

```
10000/10000 [==============================] - 13s 1ms/sample - loss: 0.8581 - accuracy: 0.7042s - loss: 0.854
[0.8581173644065857, 0.7042]
```

Predict an image.

Code:

```
# Output the possibility of each class.
new_model.predict(x_test[0:1])
```

Output:

```
array([[2.3494475e-03, 6.9919275e-05, 8.1065837e-03, 7.8556609e-01,
        2.3783690e-03, 1.8864134e-01, 6.8611270e-03, 1.2157968e-03,
        4.3428279e-03, 4.6843957e-04]], dtype=float32)
```

Code:

```
# Output the prediction result.
new_model.predict_classes(x_test[0:1])
```

Output:

```
array([3])
```

Output the first four images and their prediction results.

Code:

```
pred_list = []

plt.figure()
for i in range(0,4):
    plt.subplot(2,2,i+1)
    plt.imshow(x_test[i])
    pred = new_model.predict_classes(x_test[0:10])
    pred_list.append(pred)
    plt.title("pred:"+category_dict[pred[i]]+"     actual:"+ category_dict[y_test[i][0]])
    plt.axis('off')
plt.show()
```

Output:

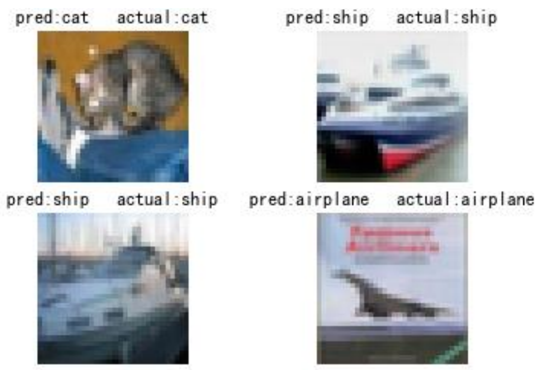


Figure 4-2 Images and prediction results

4.3 Summary

This section describes how to build an image classification model based on TensorFlow 2 and Python. It provides trainees with basic concepts in building deep learning models.