Huawei AI Certification Training

# HCIA-AI

# Experiment Guide

ISSUE:3.0



Huawei Technologies Co., Ltd.

Huawei AI Certification Training

# HCIA-AI

# Deep Learning Overview

# Experiment Guide

ISSUE:3.0



Huawei Technologies Co., Ltd.

**Trademarks and Permissions**

and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.

All other trademarks or trade names mentioned in this document are the property of their respective holders.

**Notice**

The products, services, or features you purchase are subject to Huawei's commercial contracts and terms. All or some of the products, services, or features described in this document may not be in your purchase or use scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change due to version upgrade or other reasons. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, either express or implied.

## Huawei Technologies Co., Ltd.

Address:     Huawei Industrial Base Bantian, Longgang, Shenzhen 518129 People's Republic of China

Website:     http://e.huawei.com

# Huawei Certificate System

Huawei Certification follows the "platform + ecosystem" development strategy, which is a new collaborative architecture of ICT infrastructure based on "Cloud-Pipe-Terminal". Huawei has set up a complete certification system consisting of three categories: ICT infrastructure certification, Platform and Service certification and ICT vertical certification, and grants Huawei certification the only all-range technical certification in the industry.

Huawei offers three levels of certification: Huawei Certified ICT Associate (HCIA), Huawei Certified ICT Professional (HCIP), and Huawei Certified ICT Expert (HCIE).

HCIA-AI V3.0 aims to train and certify engineers who are capable of designing and developing AI products and solutions using algorithms such as machine learning and deep learning.

HCIA-AI V3.0 certification demonstrates that: You know the development history of AI, Huawei Ascend AI system and full-stack all-scenario AI strategies, and master traditional machine learning and deep learning algorithms; you can use the TensorFlow and MindSpore development frameworks to build, train, and deploy neural networks; you are competent for sales, marketing, product manager, project management, and technical support positions in the AI field.
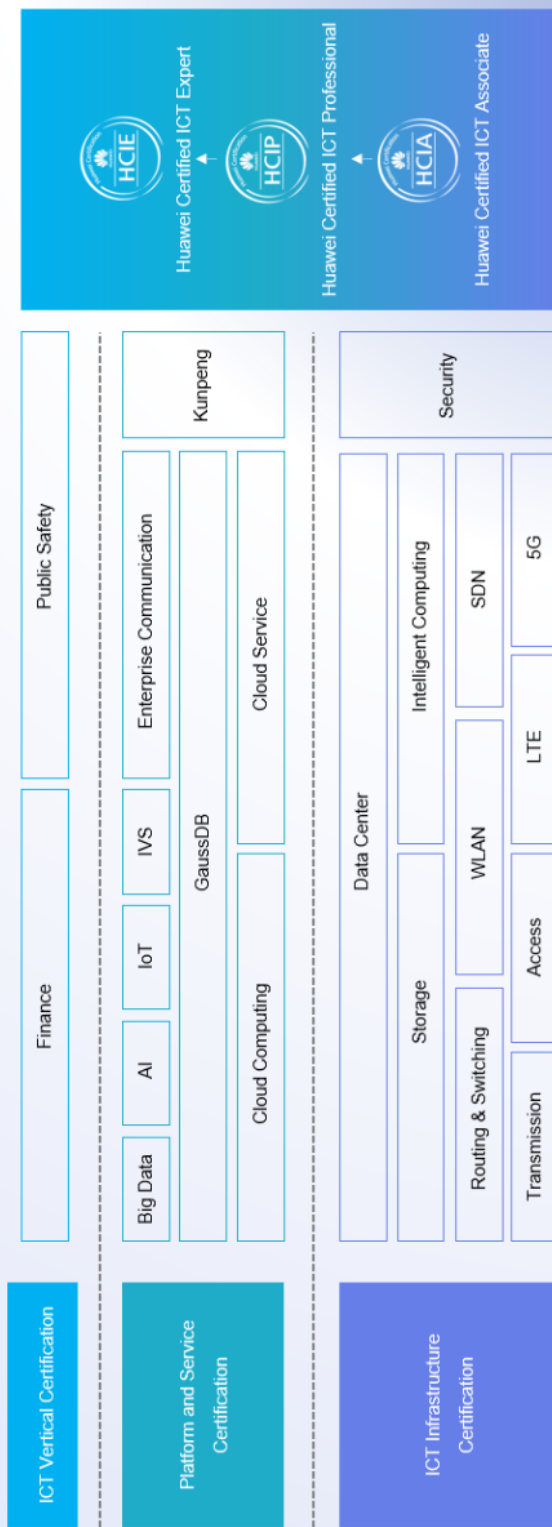
Huawei AI Certification Training

# HCIA-AI

# Mainstream Development Frameworks in the Industry

# Experiment Guide

ISSUE:3.0

Huawei Technologies Co., Ltd.

**Trademarks and Permissions**

and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.

All other trademarks or trade names mentioned in this document are the property of their respective holders.

**Notice**

The products, services, or features you purchase are subject to Huawei's commercial contracts and terms. All or some of the products, services, or features described in this document may not be in your purchase or use scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change due to version upgrade or other reasons. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, either express or implied.

# Huawei Technologies Co., Ltd.

Address: Huawei Industrial Base Bantian, Longgang, Shenzhen 518129 People's Republic of China

Website: http://e.huawei.com

# Huawei Certificate System

Huawei Certification follows the "platform + ecosystem" development strategy, which is a new collaborative architecture of ICT infrastructure based on "Cloud-Pipe-Terminal". Huawei has set up a complete certification system consisting of three categories: ICT infrastructure certification, Platform and Service certification and ICT vertical certification, and grants Huawei certification the only all-range technical certification in the industry.

Huawei offers three levels of certification: Huawei Certified ICT Associate (HCIA), Huawei Certified ICT Professional (HCIP), and Huawei Certified ICT Expert (HCIE).

HCIA-AI V3.0 aims to train and certify engineers who are capable of designing and developing AI products and solutions using algorithms such as machine learning and deep learning.

HCIA-AI V3.0 certification demonstrates that: You know the development history of AI, Huawei Ascend AI system and full-stack all-scenario AI strategies, and master traditional machine learning and deep learning algorithms; you can use the TensorFlow and MindSpore development frameworks to build, train, and deploy neural networks; you are competent for sales, marketing, product manager, project management, and technical support positions in the AI field.

# About This Document

## Overview

This document is applicable to the candidates who are preparing for the HCIA-AI exam or readers who want to understand AI basics, TensorFlow basics and AI programming basics. After learning this guide, you will be able to perform basic AI image recognition programming.

## Description

This experiment guide introduces the following three experiments:

- Experiment 1: TensorFlow basics

  This experiment mainly describes the basic syntax of TensorFlow 2.

- Experiment 2: common modules of TensorFlow 2

  This experiment mainly introduces Keras interfaces.

- Experiment 3: handwritten text recognition

  This experiment uses basic code to help learners understand how to implement handwritten text recognition through TensorFlow 2.0.

- Experiment 4: Image Classification

  This experiment is based on how to use TensorFlow 2 and python packages to predict image categories from CIFAR10 image classification dataset. It is hoped that trainees or readers can get started with deep learning and have the basic programming capability of implementing image recognition models.

## Background Knowledge Required

The readers must be able to:

- Know basic Python knowledge.
- Understand basic TensorFlow concepts.
- Command basic Python programming.
- knowledge of data structures and deep learning algorithms.

# Experiment Environment Overview

## Python Development Tool

This experiment environment is developed and compiled based on Python 3.6 and TensorFlow 2.1.0.

# Contents

# 1 TensorFlow 2.x Basics

## 1.1 Introduction

### 1.1.1 About This Experiment

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

### 1.1.2 Objectives

Upon completion of this task, 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 Experiment Steps

### 1.2.1 Introduction to 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 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()**, and **tf.ones_like():** create 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.
- Creating a list object by using NumPy, and then converting 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 shape
- **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 value of **shape**. If they are inconsistent, the system reports an error.

Code:

```
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:

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

Code:

```
#View common attributes.
print("value of the constant const_a:", const_a.numpy())
print("data type of the constant const_a:", const_a.dtype.)
print("shape of the constant const_a:", const_a.shape)
print("name of the device that is to generate the constant const_a:", const_a.device)
```

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**: constant name

Code:

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

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:

```
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, same as **shape** above.
- **value**: tensor value
- **name**: tensor name

Code:

```
fill_d = tf.fill([3,3], 8) # Create a 2x3 matrix with all 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. 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:

```
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 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:

```
#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 operated 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:

```
#Create a variable. Only the initial value needs to be provided.
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 the variable var_1:",var_1.read_value())
Assign a variable value.
var_value_1=[[1,2,3],[4,5,6]]
var_1.assign(var_value_1)
print("Value of the variable var_1 after the assignment:",var_1.read_value())
```

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:

```
#Variable addition
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

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:

```
#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 at an interval of 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 position [20,40] 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 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:

```
#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,batch_dims=1)
```

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**: indexes of the data to be extracted, which is generally a multidimensional list.

Code:

```
#Extract the pixel in [1,1] from the first dimension of the first image and the pixel in [2,2] from 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, and **tf.shape(x)** returns Tensor objects.

### 1.2.1.3.2 Dimension Reshaping

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

- **tensor**: input tensor
- **shape**: dimension 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. 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:

```
#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("size of the original data:",expand_sample_1.shape)
```

```
print("add a dimension before the first dimension (axis = 0): ",tf.expand_dims(expand_sample_1,
axis=0).shape)
print("add a dimension before the second dimension (axis = 1): ",tf.expand_dims(expand_sample_1,
axis=1).shape)
print("add a dimension after the last dimension (axis = –1): ",tf.expand_dims(expand_sample_1,
axis=-1).shape)
```

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)
```

### 1.2.1.3.4 Dimension 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:

```
#Generate a 100 x 100 x 3 tensor to represent a 100 x 100 three-channel color image.
squeeze_sample_1 = tf.random.normal([1,100,100,3])
print("size of the original data:",squeeze_sample_1.shape)
squeezed_sample_1 = tf.squeeze(expand_sample_1)
print("data size after dimension squeezing:",squeezed_sample_1.shape)
Output:
```

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

### 1.2.1.3.5 Transpose

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:

```
#Input the tensor to be transposed, and call tf.transpose.
trans_sample_1 = tf.constant([1,2,3,4,5,6],shape=[2,3])
print("size of the original data:",trans_sample_1.shape)
transposed_sample_1 = tf.transpose(trans_sample_1)
print("size of transposed data:",transposed_sample_1.shape)
```

Output:

```
Size of the original data: (2, 3)
Size of the transposed data: (3, 2)
```

Code:

**perm** is required for high-dimensional data transpose, and indicates the dimension sequence of the input tensor.

The original dimension sequence of a three-dimensional tensor is [0, 1, 2] (**perm**), indicating the length, width, and height of high-dimensional data, respectively.

Data dimensions can be transposed by changing the sequence of values in **perm**.

```
#Generate an $ 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("size of the original data:",trans_sample_2.shape)
#Exchange the length and width for the four images: The original perm value is [0,1,2,3], and the new
perm value is [0,2,1,3].
transposed_sample_2 = tf.transpose(trans_sample_2,[0,2,1,3])
print("size of transposed data:",transposed_sample_2.shape)
```

Output:

```
Size of the original data: (4, 100, 200, 3)
Size of the transposed data: (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("broadcasted data:",broadcasted_sample_1.numpy())
```

Output:

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

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:

```
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 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:

```
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, 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 all dimensions of a tensor.

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

- **input_tensor**: input tensor
- **axis**: axis to be calculated. If this parameter is not specified, the mean value of all elements is calculated.
- **keepdims**: indicates 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:

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

Output:

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

Calculate the sum of elements in each column by 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 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'):

- **values**: input tensor
- **axis**: dimension to concatenate
- **name**: operation name

Code:

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

Output:

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

A dimension can be added to an original matrix in the same way. **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:

```
stack_sample_1 = tf.random.normal([100,100,3])
stack_sample_2 = tf.random.normal([100,100,3])
print("sizes of the original data: ",stack_sample_1.shape, stack_sample_2.shape.)
#Dimensions increase after the concatenation. 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("size of the concatenated data:",stacked_sample_1.shape)
```

Output:

```
Sizes of the original data: (100, 100, 3) (100, 100, 3)
Size of the concatenated data: (2, 100, 100, 3)
```

## 1.2.1.6.2 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**: specifies the dimension based on which the tensor is split.

- **name**: operation name

Code:

```
#Split data based on the first dimension and output the split 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 sub tensors

- **axis**: specifies the dimension based on which the tensor is split.

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

- 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).

- 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:

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

Output:

Size of the original data: (10, 100, 100, 3)
Size of the split data when **m_or_size_splits** is set to **10**: (5, 2, 100, 100, 3)
Sizes of the split data when **num_or_size_splits** is set to **[3,5,2]**: (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 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:

```
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("indexes 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]
Indexes 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**: 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:

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

Output:

```
Input tensor: [1 8 7 9 6 5 4 2 3 0]
First five values in ascending order: [9 8 7 6 5]
Indexes of the first five values in ascending order: [3 1 2 4 5]
```

# 1.2.2 Eager Execution of TensorFlow 2.x

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:

```
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 2.0 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)
#Enable the drawing function, 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 2.0 and enable the eager execution mode. Another advantage of the eager execution mode lies in availability of native Python functions, for example, 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 graph mode.

## 1.2.3 AutoGraph of TensorFlow 2.x

When 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 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:

```
@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 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 CNN 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 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.x

## 2.1 Introduction

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

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

These operations include reading datasets directly from the memory, reading CSV files, reading TFRecord files, and augmenting data.

- **tf.image**: implements processing operations on images.

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 task, you will be able to master the common deep learning modeling interfaces in **tf.keras**.

## 2.3 EXPERIMENT STEPS

### 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.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'))
```

## 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 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.
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**.

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). This defaults to the Glorot uniform initializer.
- **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.

### 2.3.1.3.1 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 scheme that creates the layer's weight (kernel)
- **bias_initializer**: initialization scheme that creates the layer's weight (bias)
- **kernel_regularizer**: regularization scheme that applies to the layer's weight (kernel)
- **bias_regularizer**: regularization scheme that applies 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.
layers.Dense(32, kernel_initializer=tf.keras.initializers.he_normal)
#Set kernel_regularizer to 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 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 specified in **~/.keras/keras.json**. If this parameter 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 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:

```
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** 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 in the last cell of the output sequence.
- **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 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:

```
model = tf.keras.Sequential()
model.add(layers.Dense(10, activation='softmax'))
#Determine the optimizer (optimizer), loss function (loss), 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 is **32**.

- **epochs**: number of iteration rounds of the training model

- **verbose**: log display mode, set to **0**, **1**, or **2**.

  ◆ **0**: no display

  ◆ **1**: progress bar

  ◆ **2**: one line 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**: Total number of steps (batches of samples) to validate 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>
```

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

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**: exports and visualizes the training progress and results with TensorBoard.

Code:

```
#Set hyperparameters.
Epochs = 10

#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',
        #Whether to save the optimal model.
        save_best_only=True,
        #Monitored metric
        monitor='val_loss'),

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

    #Use TensorBoard.
    tf.keras.callbacks.TensorBoard(log_dir='./logs')
```

```
]

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.
model.save('./model/the_save_model.h5')
#Import models.
new_model = tf.keras.models.load_model('./model/the_save_model.h5')
new_prediction = new_model.predict(test_x)
#np.testing.assert_allclose: determines whether the similarity between 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, or 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 text in handwriting images. Different from printed fonts, handwriting of different people has different sizes and styles, making it difficult for computers to recognize handwriting.

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

## 3.2 Objectives

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

- Master the basic process of TensorFlow computing.
- Be familiar with the basic elements of network building, including dataset, network model building, model training, and model validation.

## 3.3 EXPERIMENT STEPS

This experiment involves the following steps:

- Reading the MNIST handwritten digit dataset.
- Getting started with TensorFlow by using simple mathematical models.
- Implementing softmax regression by using high-level APIs.
- Building a multi-layer CNN.
- Implementing a CNN by using high-level APIs.
- Visualizing prediction results.

### 3.3.1 PROJECT DESCRIPTION AND DATASET ACQUISITION

#### 3.3.1.1 Description

This project applies deep learning and TensorFlow tools to train and build models based on the MNIST handwriting dataset.

#### 3.3.1.2 Data Acquisition and Processing

##### 3.3.1.2.1 About the Dataset

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

The dataset consists of handwritten digits from 250 individuals, of which 50% are high school students and 50% are staff from Bureau of the Census.

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)

The MNIST dataset is an entry-level computer vision dataset that contains images of various handwritten digits.
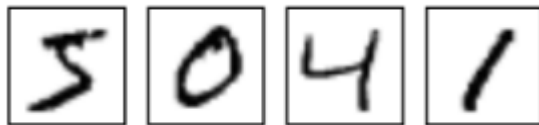


It also contains one label for each image, to clarify the correct digit. For example, the labels for the preceding four images are 5, 0, 4, and 1.

### 3.3.1.2.2 MNIST Dataset Reading

Download the MNIST dataset directly 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 are 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**.

Label data is one-hot vectors. In a one-hot vector, one digit is 1, and digits in other dimensions are all 0s. For example, label 1 may 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
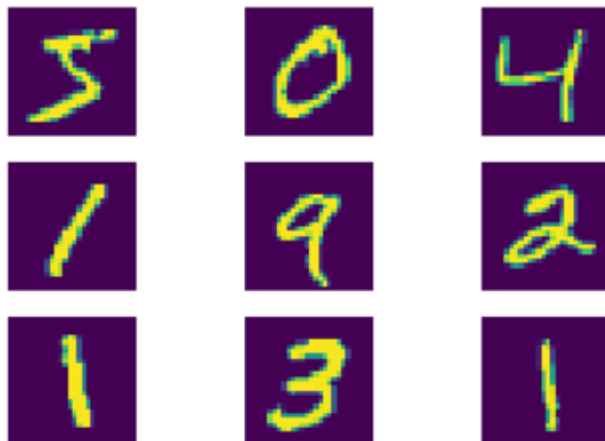
## 3.3.2.1 Data Visualization

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:



## 3.3.2.2 Data Preprocessing

An 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 into 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:

```
#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 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
```

The evaluation shows that the model accuracy reaches 0.87, and 10 training iterations have been performed.

### 3.3.3.5 Saving the DNN Model

Code:

```
model.save('./mnist_model/final_DNN_model.h5')
```

## 3.3.4 CNN Construction

The conventional CNN construction method helps you better understand the internal network structure but has 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 two pooling layers are first added by using **keras.layers**. Afterwards, a dropout layer is added to prevent overfitting. Finally, two fully connected 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=tf.train.AdamOptimizer(),loss="categorical_crossentropy",metrics=['accurac
y'])
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=mnist.test.labels)
print("Test Accuracy %.2f"%test_acc)
```

Output:

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

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

### 3.3.4.4 Saving 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 [==============================] - 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 prediction results.

Code:

```
#Visualize test set output results.
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 set 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')
        #Visualize image display.
        The img = X_test[i].reshape((28,28))#Read each row of data in the format of Ndarry.
        plt.axis("off")
        ax[i].imshow(img, cmap='Greys', interpolation='nearest')#Visualization
        ax[i].axis("off")
    print('first {} images in the test set:'.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 of the test set:
```

# Contents

# 4 Image Classification

## 4.1 Introduction

### 4.1.1 About This Experiment

This experiment is about a basic task in computer vision, that is, image recognition. The numay and TensorFlow frameworks are required. The NumPy arrays are used as the image objects. The TensorFlow framework is mainly used to create deep learning algorithms and build a convolutional neural network (CNN). This experiment recognizes image categories based on the CIFAR10 dataset.

### 4.1.2 Objectives

- Upon completion of this task, you will be able to:
    - Strengthen the understanding of Keras-based neural network model construction process
    - Master the method to load the pre-trained model.
    - Learn to use checkpoint function.
    - Master how to use a trained model to make predictions.

## 4.2 Experiment Code

### 4.2.1 Introducing 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 Data Preprocessing

```
#download Cifar-10 dataset
(x_train,y_train), (x_test, y_test) = datasets.cifar10.load_data()
#print the size of the dataset
print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)
print(y_train[0])
```

```
#Convert the category label into onehot encoding
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)
```

### Show the first 9 images

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

Output:
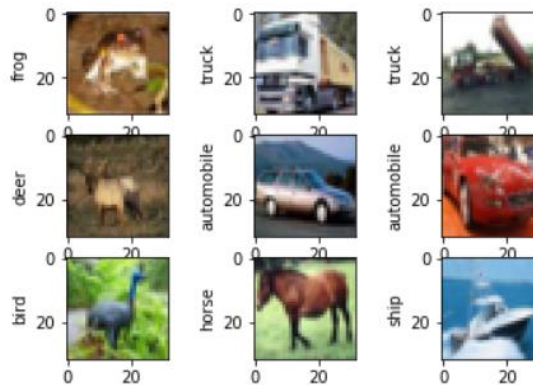


Input:

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

# 4.2.3 Model Creation

```
def CNN_classification_model(input_size = x_train.shape[1:]):
    model = Sequential()
    #the first block with 2 convolutional layers and 1 maxpooling 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))
    #relu activation function
    model.add(Activation('relu'))
    #Conv2
    model.add(Conv2D(32, (3, 3)))
    model.add(Activation('relu'))
    #maxpooling
    model.add(MaxPooling2D(pool_size=(2, 2),strides =1))

    #the second block
    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(Activation('relu'))
    model.add(Conv2D(64, (3, 3)))
    model.add(Activation('relu'))
    #maxpooling.the default strides =1
    model.add(MaxPooling2D(pool_size=(2, 2)))


    #Before sending a feature map into a fully connected network, it should be flattened into a column vector.
    model.add(Flatten())
    #fully connected layer
    model.add(Dense(128))
    model.add(Activation('relu'))
    #dropout layer.every neuronis set to 0 with a probability of 0.25
    model.add(Dropout(0.25))
    model.add(Dense(num_classes))
    #map the score of each class into probability
    model.add(Activation('softmax'))

    opt = keras.optimizers.Adam(lr=0.0001)

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

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 Model Training

```
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 pretrained models
trained_weights_path = 'cifar10_weights.h5'
if os.path.exists(trained_weights_path):
     model.load_weights(trained_weights_path, by_name =True)
#train
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 experiment is performed on a laptop. The network in this experiment 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 Model Evaluation

```
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 on a single image.

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

Output:

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

Input:

```
#output the predicted label
new_model.predict_classes(x_test[1:2])
```

Output:

```
array([0], dtype=int64)
```

Plot the first 4 images in the test set and their corresponding predicted labels.

```
#label list
pred_list = []

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

Output:



# 4.3 Summary

This chapter describes how to build an image classification model based on TensorFlow 2 and python. It provides trainees with a basic concept of deep learning model building.