Revision Record

Course Code	Product	Product Version	Course Issue

Developed By	Date	Approved By	Course Type (New/Update)	Update Description

Huawei AI Certification Training

HCIA-AI

Deep Learning and AI Development Framework

Lab Guide

ISSUE: 3.5

HUAWEI TECHNOLOGIES CO., LTD.

Huawei Certification System

Huawei Certification is an integral part of Huawei's "Platform + Ecosystem" strategy. It supports the ICT infrastructure featuring "Cloud-Pipe-Device" capabilities. We are constantly evolving our training programs to meet the latest ICT trends. With our leading talent development system and certification standards, Huawei is committed to developing ICT professionals in the digital era and building a healthy ICT talent ecosystem.

Huawei Certification consists of three categories: ICT Infrastructure Certification, Platform and Service Certification, and ICT Vertical Certification, making it one of the most extensive technical certification programs in the industry. These categories offer three levels of certification: Huawei Certified ICT Associate (HCIA), Huawei Certified ICT Professional (HCIP), and Huawei Certified ICT Expert (HCIE).

The HCIA-AI V3.5 certification is designed for engineers looking to learn and certify their knowledge in designing and developing AI products and solutions by using machine learning, deep learning, and other algorithms.

By completing the HCIA-AI V3.5 certification, you will:

(1) Understand the AI development history, Huawei Ascend AI system, full-stack all-scenario AI strategy, cutting-edge AI applications, and algorithms related to traditional machine learning and deep learning.

(2) Be able to build, train, and deploy neural networks by using the MindSpore development framework.

(3) Be ready to take on positions that require AI expertise in sales, marketing, product management, project management, technical support, and more.

About This Document

Overview

This document applies to candidates who are preparing for the HCIA-AI exam and others who want to learn basic AI knowledge and basic MindSpore programming.

Description

This guide introduces the following five exercises:

- Exercise 1: MindSpore basics, which describes the basic syntax and common modules of MindSpore.

- Exercise 2: Handwritten character recognition, in which the MindSpore framework is used to recognize handwritten characters.

- Exercise 3: MobileNetV2 image classification, which mainly introduces the classification of flower images using the lightweight network MobileNetV2.

- Exercise 4: ResNet-50 image classification exercise, which mainly introduces the classification of flower images using the ResNet-50 model.

- Exercise 5: TextCNN sentiment analysis, which mainly introduces sentiment analysis of statements using the TextCNN model.

Background Knowledge Required

This course is for Huawei's basic certification. To better understand this course, familiarize yourself with the following:

- Basic knowledge of Python, MindSpore concepts, and Python programming.

Contents

1 MindSpore Basics

1.1 Introduction

1.1.1 About This Lab

This exercise introduces the tensor data structure of MindSpore. By performing a series of operations on tensors, you can understand the basic syntax of MindSpore.

1.1.2 Objectives

- Master the method of creating tensors.
- Master the attributes and methods of tensors.

1.1.3 Lab Environment

MindSpore 1.7 or later is recommended. The exercise can be performed on a PC or by logging in to HUAWEI CLOUD and purchasing the ModelArts service.

1.2 Procedure

1.2.1 Introduction to Tensors

Tensor is a basic data structure in MindSpore network computing. For details about data types in tensors, see the dtype description on the MindSpore official website.

Tensors of different dimensions represent different data. For example, a 0-dimensional tensor represents a scalar, a 1-dimensional tensor represents a vector, a 2-dimensional tensor represents a matrix, and a 3-dimensional tensor may represent the three channels of RGB images.

MindSpore tensors support different data types, including int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32, float64 and bool_, which correspond to the data types of NumPy.

In the computation process of MindSpore, the int data type in Python is converted into the defined int64 type, and the float data type is converted into the defined float32 type.

1.2.1.1 Creating a Tensor

During tensor construction, the tensor, float, int, Boolean, tuple, list, and NumPy.array types can be input. The tuple and list can store only data of the float, int, and Boolean types.

The data type can be specified during tensor initialization. However, if the data type is not specified, the initial values **int**, **float**, and **bool** generate 0-dimensional tensors with mindspore.int32, mindspore.float32 and mindspore.bool_ data types, respectively. The data types of the 1-dimensional tensors generated by the initial values **tuple** and **list** correspond to the data types of tensors stored in the tuple and list. If multiple types of data are contained, the MindSpore data type corresponding to the data type with the highest priority is selected (Boolean < int < float). If the initial value is **Tensor**, the data type is tensor. If the initial value is **NumPy.array**, the generated tensor data type corresponds to NumPy.array.

Step 1   Create a tensor using an array.

Code:

```
# Import MindSpore.
import mindspore
# The cell outputs multiple lines at the same time.
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import numpy as np
from mindspore import Tensor
from mindspore import dtype
# Use an array to create a tensor.
x = Tensor(np.array([[1, 2], [3, 4]]), dtype.int32)
x
```

Output:

```
Tensor(shape=[2, 2], dtype=Int32, value=
[[1, 2],
 [3, 4]])
```

Step 2   Create tensors using numbers.

Code:

```
# Use a number to create tensors.
y = Tensor(1.0, dtype.int32)
z = Tensor(2, dtype.int32)
y
z
```

Output:

```
Tensor(shape=[], dtype=Int32, value= 1)
Tensor(shape=[], dtype=Int32, value= 2)
```

Step 3   Create a tensor using Boolean.

Code:

```
# Use Boolean to create a tensor.
```

```
m = Tensor(True, dtype.bool_)
m
```

Output:

```
Tensor(shape=[], dtype=Bool, value= True)
```

Step 4   Create a tensor using a tuple.

Code:

```
# Use a tuple to create a tensor.
n = Tensor((1, 2, 3), dtype.int16)
n
```

Output

```
Tensor(shape=[3], dtype=Int16, value= [1, 2, 3])
```

Step 5   Create a tensor using a list.

Code:

```
# Use a list to create a tensor.
p = Tensor([4.0, 5.0, 6.0], dtype.float64)
p
```

Output:

```
Tensor(shape=[3], dtype=Float64, value= [4.00000000e+000, 5.00000000e+000, 6.00000000e+000]
```

Step 6   Inherit attributes of another tensor to form a new tensor.

Code:

```
from mindspore import ops
oneslike = ops.OnesLike()
x = Tensor(np.array([[0, 1], [2, 1]]).astype(np.int32))
output = oneslike(x)
output
```

Output:

```
Tensor(shape=[2, 2], dtype=Int32, value=
[[1, 1],
 [1, 1]])
```

Step 7   Output constant tensor value.

Code:

```
from mindspore.ops import operations as ops
```

```
shape = (2, 2)
ones = ops.Ones()
output = ones(shape,dtype.float32)
print(output)

zeros = ops.Zeros()
output = zeros(shape, dtype.float32)
print(output)
```

Output:

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

1.2.1.2 Tensor Attributes

Tensor attributes include shape and data type (dtype).

- Shape: a tuple
- Data type: a data type of MindSpore

Code:

```
x = Tensor(np.array([[1, 2], [3, 4]]), dtype.int32)

x.shape # Shape
x.dtype # Data type
x.ndim   # Dimension
x.size # Size
```

Output:

```
(2, 2)
mindspore.int32
2
4
```

1.2.1.3 Tensor Methods

asnumpy(): converts a tensor to an array of NumPy.

Code:

```
y = Tensor(np.array([[True, True], [False, False]]), dtype.bool_)

# Convert the tensor data type to NumPy.
y_array = y.asnumpy()

y
y_array
```

Output:

```
Tensor(shape=[2, 2], dtype=Bool, value=
[[ True,   True],
 [False, False]])

array([[ True,   True],
       [False, False]])
```

1.2.1.4 Tensor Operations

There are many operations between tensors, including arithmetic, linear algebra, matrix processing (transposing, indexing, and slicing), and sampling. The following describes several operations. The usage of tensor computation is similar to that of NumPy.

Step 1   Perform indexing and slicing.

Code:

```
tensor = Tensor(np.array([[0, 1], [2, 3]]).astype(np.float32))
print("First row: {}".format(tensor[0]))
print("First column: {}".format(tensor[:, 0]))
print("Last column: {}".format(tensor[..., -1]))
```

Output:

```
First row: [0. 1.]
First column: [0. 2.]
Last column: [1. 3.]
```

Step 2   Concatenate tensors.

Code:

```
data1 = Tensor(np.array([[0, 1], [2, 3]]).astype(np.float32))
data2 = Tensor(np.array([[4, 5], [6, 7]]).astype(np.float32))
op = ops.Stack()
output = op([data1, data2])
print(output)
```

Output:

```
[[[0. 1.]
  [2. 3.]]

 [[4. 5.]
  [6. 7.]]]
```

Step 3   Convert to NumPy.

Code:

```
zeros = ops.Zeros()
output = zeros((2,2), dtype.float32)
print("output: {}".format(type(output)))
```

```
n_output = output.asnumpy()
print("n_output: {}".format(type(n_output)))
```

Output:

```
output: <class 'mindspore.common.tensor.Tensor'>
n_output: <class 'numpy.ndarray'>
```

1.2.2 Loading a Dataset

MindSpore.dataset provides APIs to load and process datasets such as MNIST, CIFAR-10, CIFAR-100, VOC, ImageNet, and CelebA.

Step 1　　Load the MNIST dataset.

You are advised to download the MNIST dataset from http://yann.lecun.com/exdb/mnist/ and save the training and test files to the MNIST folder.

Code:

```
import os
import mindspore.dataset as ds
import matplotlib.pyplot as plt

dataset_dir = "./MNIST/train"   # Path of the dataset
# Read three images from the MNIST dataset.
mnist_dataset = ds.MnistDataset(dataset_dir=dataset_dir, num_samples=3)
# View the images and set the image sizes.
plt.figure(figsize=(8,8))
i = 1

# Print three subgraphs.
for dic in mnist_dataset.create_dict_iterator(output_numpy=True):
    plt.subplot(3,3,i)
    plt.imshow(dic['image'][:,:,0])
    plt.axis('off')
    i +=1
plt.show()
```

Output:

Figure 1-1 MNIST dataset sample

Step 2　　Customize a dataset.

For datasets that cannot be directly loaded by MindSpore, you can build a custom dataset class and use the GeneratorDataset API to customize data loading.

Code:

```
import numpy as np
np.random.seed(58)

class DatasetGenerator:
# When a dataset object is instantiated, the __init__ function is called. You can perform operations
such as data initialization.
    def __init__(self):
        self.data = np.random.sample((5, 2))
        self.label = np.random.sample((5, 1))
# Define the __getitem__ function of the dataset class to support random access and obtain and
return data in the dataset based on the specified index value.
    def __getitem__(self, index):
        return self.data[index], self.label[index]
# Define the __len__ function of the dataset class and return the number of samples in the dataset.
    def __len__(self):
        return len(self.data)
# After the dataset class is defined, the GeneratorDataset API can be used to load and access dataset
samples in custom mode.
dataset_generator = DatasetGenerator()
dataset = ds.GeneratorDataset(dataset_generator, ["data", "label"], shuffle=False)
# Use the create_dict_iterator method to obtain data.
for data in dataset.create_dict_iterator():
    print('{}'.format(data["data"]), '{}'.format(data["label"]))
```

Output:

```
[0.36510558 0.45120592] [0.78888122]
[0.49606035 0.07562207] [0.38068183]
[0.57176158 0.28963401] [0.16271622]
[0.30880446 0.37487617] [0.54738768]
[0.81585667 0.96883469] [0.77994068]
```

Step 3   Perform data augmentation.

The dataset APIs provided by MindSpore support data processing methods such as shuffle and batch. You only need to call the corresponding function API to quickly process data.

In the following example, the datasets are shuffled, and two samples form a batch.

Code:

```
ds.config.set_seed(58)

# Shuffle the data sequence. buffer_size indicates the size of the shuffled buffer in the dataset.
dataset = dataset.shuffle(buffer_size=10)
# Divide the dataset into batches. batch_size indicates the number of data records contained in each
batch. Set this parameter to 2.
dataset = dataset.batch(batch_size=2)

for data in dataset.create_dict_iterator():
    print("data: {}".format(data["data"]))
    print("label: {}".format(data["label"]))
```

Output:

```
data: [[0.36510558 0.45120592]
 [0.57176158 0.28963401]]
label: [[0.78888122]
 [0.16271622]]
data: [[0.30880446 0.37487617]
 [0.49606035 0.07562207]]
label: [[0.54738768]
 [0.38068183]]
data: [[0.81585667 0.96883469]]
label: [[0.77994068]]
```

Code:

```python
import matplotlib.pyplot as plt

from mindspore.dataset.vision import Inter
import mindspore.dataset.vision.c_transforms as c_vision

DATA_DIR = './MNIST/train'
# Obtain six samples.
mnist_dataset = ds.MnistDataset(DATA_DIR, num_samples=6, shuffle=False)
# View the original image data.
mnist_it = mnist_dataset.create_dict_iterator()
data = next(mnist_it)
plt.imshow(data['image'].asnumpy().squeeze(), cmap=plt.cm.gray)
plt.title(data['label'].asnumpy(), fontsize=20)
plt.show()
```

Output:
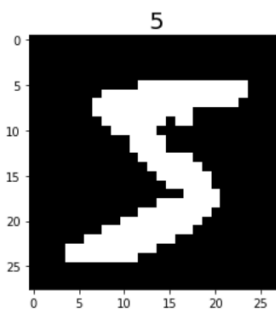
Figure 1-2 Data sample

Code:

```python
resize_op = c_vision.Resize(size=(40,40), interpolation=Inter.LINEAR)
crop_op = c_vision.RandomCrop(28)
transforms_list = [resize_op, crop_op]
mnist_dataset = mnist_dataset.map(operations=transforms_list, input_columns=["image"])
mnist_dataset = mnist_dataset.create_dict_iterator()
data = next(mnist_dataset)
plt.imshow(data['image'].asnumpy().squeeze(), cmap=plt.cm.gray)
plt.title(data['label'].asnumpy(), fontsize=20)
plt.show()
```

Output:
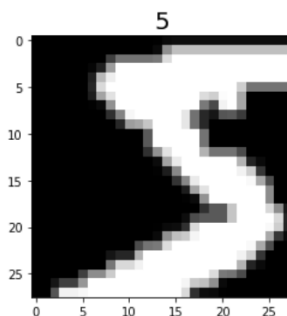
Figure 1-3 Effect after data argumentation

1.2.3 Building the Network

MindSpore encapsulates APIs for building network layers in the nn module. Different types of neural network layers are built by calling these APIs.

Step 1   Build a fully-connected layer.

Fully-connected layer: mindspore.nn.Dense

- **in_channels**: input channel
- **out_channels**: output channel
- **weight_init**: weight initialization. Default value: **'normal'**.

Code:

```
import mindspore as ms
import mindspore.nn as nn
from mindspore import Tensor
import numpy as np

# Construct the input tensor.
input_a = Tensor(np.array([[1, 1, 1], [2, 2, 2]]), ms.float32)
print(input_a)
# Construct a fully-connected network. Set both in_channels and out_channels to 3.
net = nn.Dense(in_channels=3, out_channels=3, weight_init=1)
output = net(input_a)
print(output)
```

Output:

```
[[1. 1. 1.]
 [2. 2. 2.]]
[[3. 3. 3.]
 [6. 6. 6.]]
```

Step 2   Build a convolutional layer.

Code:

```
conv2d = nn.Conv2d(1, 6, 5, has_bias=False, weight_init='normal', pad_mode='valid')
input_x = Tensor(np.ones([1, 1, 32, 32]), ms.float32)
```

```
print(conv2d(input_x).shape)
```

Output:

```
(1, 6, 28, 28)
```

Step 3   Build a ReLU layer.

Code:

```
relu = nn.ReLU()
input_x = Tensor(np.array([-1, 2, -3, 2, -1]), ms.float16)
output = relu(input_x)

print(output)
```

Output:

```
[0. 2. 0. 2. 0.]
```

Step 4   Build a pooling layer.

Code:

```
max_pool2d = nn.MaxPool2d(kernel_size=2, stride=2)
input_x = Tensor(np.ones([1, 6, 28, 28]), ms.float32)

print(max_pool2d(input_x).shape)
```

Output:

```
(1, 6, 14, 14)
```

Step 5   Build a Flatten layer.

Code:

```
flatten = nn.Flatten()
input_x = Tensor(np.ones([1, 16, 5, 5]), ms.float32)
output = flatten(input_x)

print(output.shape)
```

Output:

```
(1, 400)
```

Step 6   Define a model class and view parameters.

The Cell class of MindSpore is the base class for building all networks and the basic unit of a network. When a neural network is required, you need to inherit the Cell class and overwrite the __init__ and construct methods.

Code:

```
class LeNet5(nn.Cell):
    """
    Lenet network structure
    """
    def __init__(self, num_class=10, num_channel=1):
        super(LeNet5, self).__init__()
        # Define the required operations.
        self.conv1 = nn.Conv2d(num_channel, 6, 5, pad_mode='valid')
        self.conv2 = nn.Conv2d(6, 16, 5, pad_mode='valid')
        self.fc1 = nn.Dense(16 * 4 * 4, 120)
        self.fc2 = nn.Dense(120, 84)
        self.fc3 = nn.Dense(84, num_class)
        self.relu = nn.ReLU()
        self.max_pool2d = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten = nn.Flatten()

    def construct(self, x):
        # Use the defined operations to build a feedforward network.
        x = self.conv1(x)
        x = self.relu(x)
        x = self.max_pool2d(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.max_pool2d(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x
# Instantiate the model and use the parameters_and_names method to view the model parameters.
modelle = LeNet5()
for m in modelle.parameters_and_names():
    print(m)
```

Output:

```
('conv1.weight', Parameter (name=conv1.weight, shape=(6, 1, 5, 5), dtype=Float32,
requires_grad=True))
('conv2.weight', Parameter (name=conv2.weight, shape=(16, 6, 5, 5), dtype=Float32,
requires_grad=True))
('fc1.weight', Parameter (name=fc1.weight, shape=(120, 400), dtype=Float32, requires_grad=True))
('fc1.bias', Parameter (name=fc1.bias, shape=(120,), dtype=Float32, requires_grad=True))
('fc2.weight', Parameter (name=fc2.weight, shape=(84, 120), dtype=Float32, requires_grad=True))
('fc2.bias', Parameter (name=fc2.bias, shape=(84,), dtype=Float32, requires_grad=True))
('fc3.weight', Parameter (name=fc3.weight, shape=(10, 84), dtype=Float32, requires_grad=True))
('fc3.bias', Parameter (name=fc3.bias, shape=(10,), dtype=Float32, requires_grad=True))
```

1.2.4 Training and Validating a Model

Step 1   Use loss functions.

A loss function is used to validate the difference between the predicted and actual values of a model. Here, the absolute error loss function L1Loss is used. mindspore.nn.loss also provides many other loss functions, such as SoftmaxCrossEntropyWithLogits, MSELoss, and SmoothL1Loss.

The output value and target value are provided to compute the loss value. The method is as follows:

Code:

```
import numpy as np
import mindspore.nn as nn
from mindspore import Tensor
import mindspore.dataset as ds
import mindspore as ms
loss = nn.L1Loss()
output_data = Tensor(np.array([[1, 2, 3], [2, 3, 4]]).astype(np.float32))
target_data = Tensor(np.array([[0, 2, 5], [3, 1, 1]]).astype(np.float32))
print(loss(output_data, target_data))
```

Output:

```
1.5
```

Step 2   Use an optimizer.

Common deep learning optimization algorithms include SGD, Adam, Ftrl, lazyadam, Momentum, RMSprop, Lars, Proximal_ada_grad, and lamb.

Momentum optimizer: mindspore.nn.Momentum

Code:

```
optim = nn.Momentum(params=modelle.trainable_params(), learning_rate=0.1, momentum=0.9,
weight_decay=0.0)
```

Step 3   Build a model.

mindspore.Model(network, loss_fn, optimizer, metrics)

Code:

```
from mindspore import Model

# Define a neural network.
net = LeNet5()
# Define the loss function.
loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
# Define the optimizer.
optim = nn.Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9)
# Build a model.
```

```
model = Model(network = net, loss_fn=loss, optimizer=optim, metrics={'accuracy'})
```

Step 4   Train the model.

Code:

```
import mindspore.dataset.transforms.c_transforms as C
import mindspore.dataset.vision.c_transforms as CV
from mindspore.train.callback import   LossMonitor

DATA_DIR = './MNIST/train'
mnist_dataset = ds.MnistDataset(DATA_DIR)

resize_op = CV.Resize((28,28))
rescale_op = CV.Rescale(1/255,0)
hwc2chw_op = CV.HWC2CHW()

mnist_dataset   = mnist_dataset .map(input_columns="image", operations=[rescale_op,resize_op,
hwc2chw_op])
mnist_dataset   = mnist_dataset .map(input_columns="label", operations=C.TypeCast(ms.int32))
mnist_dataset = mnist_dataset.batch(32)
loss_cb = LossMonitor(per_print_times=1000)
# dataset is an input parameter, which indicates the training set, and epoch indicates the number of
training epochs of the training set.
model.train(epoch=1, train_dataset=mnist_dataset,callbacks=[loss_cb])
```

Step 5   Validate the model.

Code:

```
# Test set
DATA_DIR = './forward_mnist/MNIST/test'
dataset = ds.MnistDataset(DATA_DIR)

resize_op = CV.Resize((28,28))
rescale_op = CV.Rescale(1/255,0)
hwc2chw_op = CV.HWC2CHW()

dataset   = dataset .map(input_columns="image", operations=[rescale_op,resize_op, hwc2chw_op])
dataset   = dataset .map(input_columns="label", operations=C.TypeCast(ms.int32))
dataset = dataset.batch(32)
model.eval(valid_dataset=dataset)
```

1.2.5 Saving and Loading a Model

After the preceding network model is trained, the model can be saved in two forms:

1. One is to simply save the network model before or after training. The advantage is that the API is easy to use, but only the network model status when the command is executed is retained.

Code:

```
import mindspore as ms
```

```
# net indicates a defined network model, which is used before or after training.
ms.save_checkpoint(net, "./MyNet.ckpt") # net indicates the training network, and ./MyNet.ckpt
indicates the path for saving the network model.
```

Output:

```
epoch: 1 step: 1000, loss is 2.283555746078491
```

2. The other one is to save the interface during network model training. MindSpore automatically saves the number of epochs and number of steps set during training. That is, the intermediate weight parameters generated during the model training process are also saved to facilitate network fine-tuning and stop training.

Code:

```
from mindspore.train.callback import ModelCheckpoint, CheckpointConfig

# Set the value of epoch_num.
epoch_num = 5

# Set model saving parameters.
config_ck = CheckpointConfig(save_checkpoint_steps=1875, keep_checkpoint_max=10)

# Use model saving parameters.
ckpoint = ModelCheckpoint(prefix="lenet", directory="./lenet", config=config_ck)
model.train(epoch_num, mnist_dataset, callbacks=[ckpoint])
```

1.2.6 Automatic Differentiation

Backward propagation is the commonly used algorithm for training neural networks. In this algorithm, parameters (model weights) are adjusted based on a gradient of a loss function for a given parameter.

The first-order derivative method of MindSpore is **mindspore.ops.GradOperation (get_all=False, get_by_list=False, sens_param=False)**. When **get_all** is set to **False**, the first input derivative is computed. When **get_all** is set to **True**, all input derivatives are computed. When **get_by_list** is set to **False**, weight derivatives are not computed. When **get_by_list** is set to **True**, weight derivatives are computed. **sens_param** scales the output value of the network to change the final gradient.

The following uses the MatMul operator derivative for in-depth analysis.

Step 1   Compute the first-order derivative of the input.

To compute the input derivative, you need to define a network requiring a derivative. The following uses a network $f(x,y)=z*x*y$ formed by the MatMul operator as an example.

Code:

```
import numpy as np
import mindspore.nn as nn
import mindspore.ops as ops
from mindspore import Tensor
from mindspore import ParameterTuple, Parameter
from mindspore import dtype as mstype
```

```
class Net(nn.Cell):
    def __init__(self):
        super(Net, self).__init__()
        self.matmul = ops.MatMul()
        self.z = Parameter(Tensor(np.array([1.0], np.float32)), name='z')

    def construct(self, x, y):
        x = x * self.z
        out = self.matmul(x, y)
        return out
class GradNetWrtX(nn.Cell):
    def __init__(self, net):
        super(GradNetWrtX, self).__init__()
        self.net = net
        self.grad_op = ops.GradOperation()

    def construct(self, x, y):
        gradient_function = self.grad_op(self.net)
        return gradient_function(x, y)
x = Tensor([[0.8, 0.6, 0.2], [1.8, 1.3, 1.1]], dtype=mstype.float32)
y = Tensor([[0.11, 3.3, 1.1], [1.1, 0.2, 1.4], [1.1, 2.2, 0.3]], dtype=mstype.float32)
output = GradNetWrtX(Net())(x, y)
print(output)
```

Output:

```
[[4.5099998 2.7        3.6000001]
 [4.5099998 2.7        3.6000001]]
```

Step 2   Compute the first-order derivative of the weight.

To compute weight derivatives, you need to set **get_by_list** in **ops.GradOperation** to **True**. If computation of certain weight derivatives is not required, set **requires_grad** to **False** when you definite the network.

Code:

```
class GradNetWrtX(nn.Cell):
    def __init__(self, net):
        super(GradNetWrtX, self).__init__()
        self.net = net
        self.params = ParameterTuple(net.trainable_params())
        self.grad_op = ops.GradOperation(get_by_list=True)

    def construct(self, x, y):
        gradient_function = self.grad_op(self.net, self.params)
        return gradient_function(x, y)
output = GradNetWrtX(Net())(x, y)
print(output)
```

Output:

```
(Tensor(shape=[1], dtype=Float32, value= [ 2.15359993e+01]),)
```

1.3 Question

When the following code is used to create two tensors, t1 and t2, can t1 be created properly? Check whether the two tensors have the same outputs. If not, what is the difference?

2 MNIST Handwritten Character Recognition

2.1 Introduction

2.1.1 About This Exercise

This exercise implements the MNIST handwritten character recognition, which is a typical case in the deep learning field. The whole process is as follows:

- Process the required dataset. (The MNIST dataset is used in this example.)
- Define a network. (A simple fully-connected network is built in this example.)
- Define a loss function and an optimizer.
- Load the dataset and perform training. After the training is complete, use the test set for validation.

2.2 Preparations

Before you start, check whether MindSpore has been correctly installed. You are advised to install MindSpore on your computer by referring to the MindSpore official website https://www.mindspore.cn/install/en.

In addition, you should have basic mathematical knowledge, including knowledge of Python coding basics, probability, and matrices.

Recommended environment:

Version: MindSpore 1.7

Programming language: Python 3.7

2.3 Detailed Design and Implementation

2.3.1 Data Preparation

The MNIST dataset used in this example consists of 10 classes of 28 x 28 pixels grayscale images. It has a training set of 60,000 examples, and a test set of 10,000 examples.

Download the MNIST dataset at http://yann.lecun.com/exdb/mnist/. Four dataset download links are provided. The first two links are for downloading test data files, and the last two links are for downloading training data files.

Download and decompress the files, and store them in the workspace directories **./MNIST /train** and **./MNIST /test**.

The directory structure is as follows:

```
└─MNIST
    ├─  test
    │       t10k-images.idx3-ubyte
    │       t10k-labels.idx1-ubyte
    │
    └─  train
            train-images.idx3-ubyte
            train-labels.idx1-ubyte
```

2.3.2 Procedure

Step 1   Import the Python library and module and configure running information.

Import the required Python library.

Currently, the **os** library is required. Other required libraries will not be described here. For details about the MindSpore modules, see the MindSpore API page. You can use context.set_context to configure the information required for running, such as the running mode, backend information, and hardware information.

Import the context module and configure the required information.

Code:

```
# Import related dependent libraries.
import   os
from matplotlib import pyplot as plt
import numpy as np

import mindspore as ms
import mindspore.context as context
import mindspore.dataset as ds
import mindspore.dataset.transforms.c_transforms as C
import mindspore.dataset.vision.c_transforms as CV
from mindspore.nn.metrics import Accuracy

from mindspore import nn
from mindspore.train import Model
from mindspore.train.callback import ModelCheckpoint, CheckpointConfig, LossMonitor, TimeMonitor

context.set_context(mode=context.GRAPH_MODE, device_target='CPU')
```

The graph mode is used in this exercise. You can configure hardware information as required. For example, if the code runs on the Ascend AI processor, set **device_target** to **Ascend**. This rule also applies to the code running on the CPU and GPU. For details about

parameters, see the context.set_context API description at
https://www.mindspore.cn/docs/en/r1.7/api_python/mindspore.context.html.

Step 2   Read data.

Use the data reading function of MindSpore to read the MNIST dataset and view the data volume and sample information of the training set and test set.

Code:

```
DATA_DIR_TRAIN = "MNIST/train" # Training set information
DATA_DIR_TEST = "MNIST/test" # Test set information
# Read data.
ds_train = ds.MnistDataset(DATA_DIR_TRAIN)
ds_test = ds.MnistDataset(DATA_DIR_TEST )
# Display the dataset features.
print('Data volume of the training dataset:',ds_train.get_dataset_size())
print('Data volume of the test dataset:',ds_test.get_dataset_size())
image=ds_train.create_dict_iterator().__next__()
print('Image length/width/channels:',image['image'].shape)
print('Image label style:',image['label'])     # Total 10 label classes which are represented by
numbers from 0 to 9.
```

Step 3   Process data.

Datasets are crucial for training. A good dataset can effectively improve training accuracy and efficiency. Generally, before loading a dataset, you need to perform some operations on the dataset.

Define a dataset and data operations.

We define the create_dataset function to create a dataset. In this function, we define the data augmentation and processing operations to be performed:

- Read the dataset.

- Define parameters required for data augmentation and processing.

- Generate corresponding data augmentation operations according to the parameters.

- Use the map function to apply data operations to the dataset.

- Process the generated dataset.

Code:

```
def create_dataset(training=True, batch_size=128, resize=(28, 28),
                   rescale=1/255, shift=0, buffer_size=64):
    ds = ms.dataset.MnistDataset(DATA_DIR_TRAIN if training else DATA_DIR_TEST)
    # Define the resizing, normalization, and channel conversion of the map operation.
    resize_op = CV.Resize(resize)
    rescale_op = CV.Rescale(rescale,shift)
    hwc2chw_op = CV.HWC2CHW()
    # Perform the map operation on the dataset.
    ds = ds.map(input_columns="image", operations=[rescale_op,resize_op, hwc2chw_op])
    ds = ds.map(input_columns="label", operations=C.TypeCast(ms.int32))
    # Set the shuffle parameter and batch size.
    ds = ds.shuffle(buffer_size=buffer_size)
    ds = ds.batch(batch_size, drop_remainder=True)
```

```
return ds
```

In the preceding information, **batch_size** indicates the number of data records in each batch. Assume that each batch contains 32 data records. Modify the image size, normalization, and image channel, and then modify the data type of the label. Perform the shuffle operation, set **batch_size**, and set **drop_remainder** to **True**. In this case, data that cannot form a batch in the dataset will be discarded.

MindSpore supports multiple data processing and augmentation operations, which are usually used together. For details, see **Data Processing and Augmentation** on the MindSpore official website.

Step 4   Sample visualization

Read the first 10 samples and visualize the samples to determine whether the samples are real datasets.

Code:

```
# Display the first 10 images and the labels, and check whether the images are correctly labeled.
ds = create_dataset(training=False)
data = ds.create_dict_iterator().__next__()
images = data['image'].asnumpy()
labels = data['label'].asnumpy()
plt.figure(figsize=(15,5))
for i in range(1,11):
    plt.subplot(2, 5, i)
    plt.imshow(np.squeeze(images[i]))
    plt.title('Number: %s' % labels[i])
    plt.xticks([])
plt.show()
```

Output:
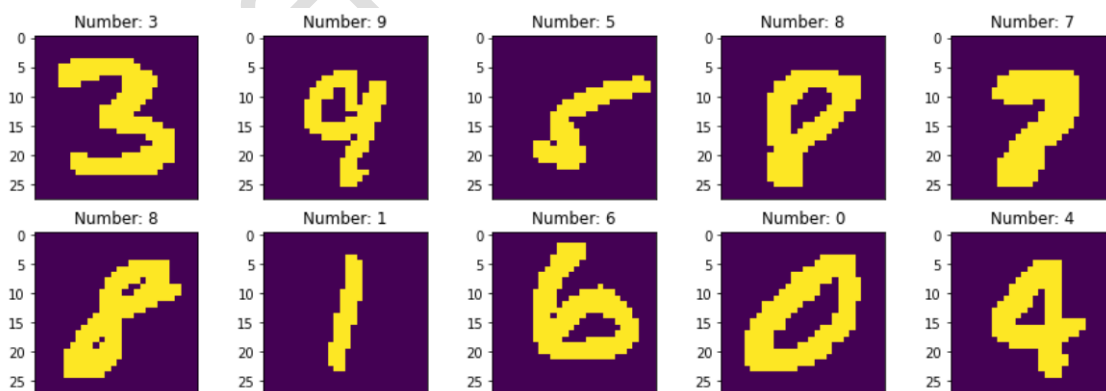
Figure 2-1 Sample visualization

Step 5   Define a network.

We define a simple fully-connected network to implement image recognition. The network has only three layers:

The first layer is a fully-connected layer of the shape 784 x 512.

The second layer is a fully-connected layer of the shape 512 x 128.

The last layer is an output layer of the shape 128 x 10.

To use MindSpore for neural network definition, inherit mindspore.nn.Cell. Cell is the base class of all neural networks (such as Conv2d).

Define each layer of a neural network in the __init__ method in advance, and then define the construct method to complete the feedforward construction of the neural network. The network layers are defined as follows:

Code:

```
# Create a model. The model consists of three fully connected layers. The final output layer uses
softmax for classification (10 classes consisting of numbers 0 to 9.)
class ForwardNN(nn.Cell):
    def __init__(self):
        super(ForwardNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Dense(784, 512, activation='relu')
        self.fc2 = nn.Dense(512, 128, activation='relu')
        self.fc3 = nn.Dense(128, 10, activation=None)


    def construct(self, input_x):
        output = self.flatten(input_x)
        output = self.fc1(output)
        output = self.fc2(output)
        output = self.fc3(output)
        return output
```

Step 6   Define a loss function and an optimizer.

A loss function is also called an objective function and is used to measure the difference between a predicted value and an actual value. Deep learning reduces the loss value by continuous iteration. Defining a good loss function can effectively improve model performance.

An optimizer is used to minimize the loss function, improving the model during training.

After the loss function is defined, the weight-related gradient of the loss function can be obtained. The gradient is used to indicate the weight optimization direction for the optimizer, improving model performance. Loss functions supported by MindSpore include SoftmaxCrossEntropyWithLogits, L1Loss, and MSELoss. SoftmaxCrossEntropyWithLogits is used in this example.

MindSpore provides the callback mechanism to execute custom logic during training. The following uses ModelCheckpoint provided by the framework as an example. ModelCheckpoint can save the network model and parameters for subsequent fine-tuning.

Code:

```
# Create a network, loss function, validation metric, and optimizer, and set related hyperparameters.
lr = 0.001
num_epoch = 10
momentum = 0.9
```

```
net = ForwardNN()
loss = nn.loss.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
metrics={"Accuracy": Accuracy()}
opt = nn.Adam(net.trainable_params(), lr)
```

Step 7   Start training.

The training process refers to a process in which a training dataset is transferred to a network for training and optimizing network parameters. In the MindSpore framework, the Model.train method is used to complete this process.

Code:

```
# Build a model.
model = Model(net, loss, opt, metrics)
config_ck = CheckpointConfig(save_checkpoint_steps=1875, keep_checkpoint_max=10)
ckpoint_cb = ModelCheckpoint(prefix="checkpoint_net",directory = "./ckpt" ,config=config_ck)
# Generate a dataset.
ds_eval = create_dataset(False, batch_size=32)
ds_train = create_dataset(batch_size=32)
# Train the model.
loss_cb = LossMonitor(per_print_times=1875)
time_cb = TimeMonitor(data_size=ds_train.get_dataset_size())
print("============= Starting Training =============")
model.train(num_epoch, ds_train,callbacks=[ckpoint_cb,loss_cb,time_cb ],dataset_sink_mode=False)
```

Although loss values may fluctuate, they gradually decrease and the accuracy gradually increases in general. Loss values displayed each time may be different because of their randomicity. The following is an example of loss values output during training:

```
============= Starting Training =============
epoch: 1 step: 1875, loss is 0.06333521
epoch time: 18669.680 ms, per step time: 9.957 ms
epoch: 2 step: 1875, loss is 0.07061358
epoch time: 21463.662 ms, per step time: 11.447 ms
epoch: 3 step: 1875, loss is 0.043515638
epoch time: 25836.919 ms, per step time: 13.780 ms
epoch: 4 step: 1875, loss is 0.03468642
epoch time: 25553.150 ms, per step time: 13.628 ms
epoch: 5 step: 1875, loss is 0.03934026
epoch time: 27364.246 ms, per step time: 14.594 ms
epoch: 6 step: 1875, loss is 0.0023852987
epoch time: 31432.281 ms, per step time: 16.764 ms
epoch: 7 step: 1875, loss is 0.010915326
epoch time: 33697.183 ms, per step time: 17.972 ms
epoch: 8 step: 1875, loss is 0.011417691
epoch time: 29594.438 ms, per step time: 15.784 ms
epoch: 9 step: 1875, loss is 0.00044568744
epoch time: 28676.948 ms, per step time: 15.294 ms
epoch: 10 step: 1875, loss is 0.071476705
epoch time: 34999.863 ms, per step time: 18.667 ms
```

Step 8   Validate the model.

In this step, the original test set is used to validate the model.

Code:

```
# Use the test set to validate the model and print the overall accuracy.
metrics=model.eval(ds_eval)
print(metrics)
```

Output:

```
{'Accuracy': 0.9740584935897436}
```

3 MobileNetV2 Image Classification

3.1 Introduction

In this exercise, the lightweight network MobileNetV2 is used to classify flower image datasets.

3.2 Preparations

Before you start, check whether MindSpore has been correctly installed. You are advised to install MindSpore on your computer by referring to the MindSpore official website https://www.mindspore.cn/install/en.

In addition, you should have basic mathematical knowledge, including knowledge of Python coding basics, probability, and matrix.

Recommended environment:

Version: MindSpore 1.7

Programming language: Python 3.7

3.3 Detailed Design and Implementation

3.3.1 Data Preparation

Image flower dataset used in the example is an open-source dataset and contains five flower types: daisies (633 images) dandelions (898 images), roses (641 images), sunflowers (699 images), and tulips (799 images). The 3670 photos, which are about 230 MB in total, are stored in five folders. To facilitate the model test after the model deployment, the dataset is divided into flower_photos_train and flower_photos_test.

The directory structure is as follows:

```
flower_photos_train
        ├── daisy
        ├── dandelion
        ├── roses
        ├── sunflowers
        ├── tulips
```

```
├── LICENSE.txt
flower_photos_test
    ├── daisy
    ├── dandelion
    ├── roses
    ├── sunflowers
    ├── tulips
    ├── LICENSE.txt
```

Obtain the datasets from the following links:

https://ascend-professional-construction-dataset.obs.myhuaweicloud.com/deep-learning/flower_photos_train.zip
https://ascend-professional-construction-dataset.obs.myhuaweicloud.com/deep-learning/flower_photos_test.zip

3.3.2 Procedure

Step 1   Load the dataset.

Define the create_dataset function, use the ImageFolderDataset API to load the flower image classification dataset, and perform image enhancement on the dataset. Code:

```python
import mindspore.dataset as ds
import mindspore.dataset.vision.c_transforms as CV
from mindspore import dtype as mstype

train_data_path = 'flower_photos_train'
val_data_path = 'flower_photos_test'


def create_dataset(data_path, batch_size=18, training=True):
    """Define the dataset."""

    data_set = ds.ImageFolderDataset(data_path, num_parallel_workers=8, shuffle=True,
                                    class_indexing={'daisy': 0, 'dandelion': 1, 'roses': 2, 'sunflowers': 3,
                                                    'tulips': 4})
    # Perform image enhancement on the dataset.
    image_size = 224
    mean = [0.485 * 255, 0.456 * 255, 0.406 * 255]
    std = [0.229 * 255, 0.224 * 255, 0.225 * 255]
    if training:
        trans = [
            CV.RandomCropDecodeResize(image_size, scale=(0.08, 1.0), ratio=(0.75, 1.333)),
            CV.RandomHorizontalFlip(prob=0.5),
            CV.Normalize(mean=mean, std=std),
            CV.HWC2CHW()
        ]
    else:
        trans = [
```

```
            CV.Decode(),
            CV.Resize(256),
            CV.CenterCrop(image_size),
            CV.HWC2CHW()
        ]

    # Perform the data map, batch, and repeat operations.
    data_set = data_set.map(operations=trans, input_columns="image", num_parallel_workers=8)
    # Set the value of the batch_size. Discard the samples if the number of samples last fetched is
less than the value of batch_size.
    data_set = data_set.batch(batch_size, drop_remainder=True)

    return data_set


dataset_train = create_dataset(train_data_path)
dataset_val = create_dataset(val_data_path)
```

Step 2   Visualize the dataset.

The return value of the training dataset loaded from the create_dataset API is a dictionary. You can use the create_dict_iterator API to create a data iterator and use **next** to iteratively access the dataset. Here, **batch_size** is set to **18**. Therefore, you can use **next** to obtain 18 images and label data at a time. Code:

```
import matplotlib.pyplot as plt
import numpy as np

data = next(dataset_train.create_dict_iterator())
images = data["image"]
labels = data["label"]

print("Tensor of image", images.shape)
print("Labels:", labels)

# class_name corresponds to label. Labels are marked in ascending order of the folder character
 string.
class_name = {0:'daisy',1:'dandelion',2:'roses',3:'sunflowers',4:'tulips'}
plt.figure(figsize=(15, 7))
for i in range(len(labels)):
    # Obtain an image and its label.
    data_image = images[i].asnumpy()
    data_label = labels[i]
    # Process images for display.
    data_image = np.transpose(data_image, (1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    data_image = std * data_image + mean
    data_image = np.clip(data_image, 0, 1)
    # Display the image.
    plt.subplot(3, 6, i + 1)
    plt.imshow(data_image)
    plt.title(class_name[int(labels[i].asnumpy())])
```

```
        plt.axis("off")
plt.show()
```

Output:
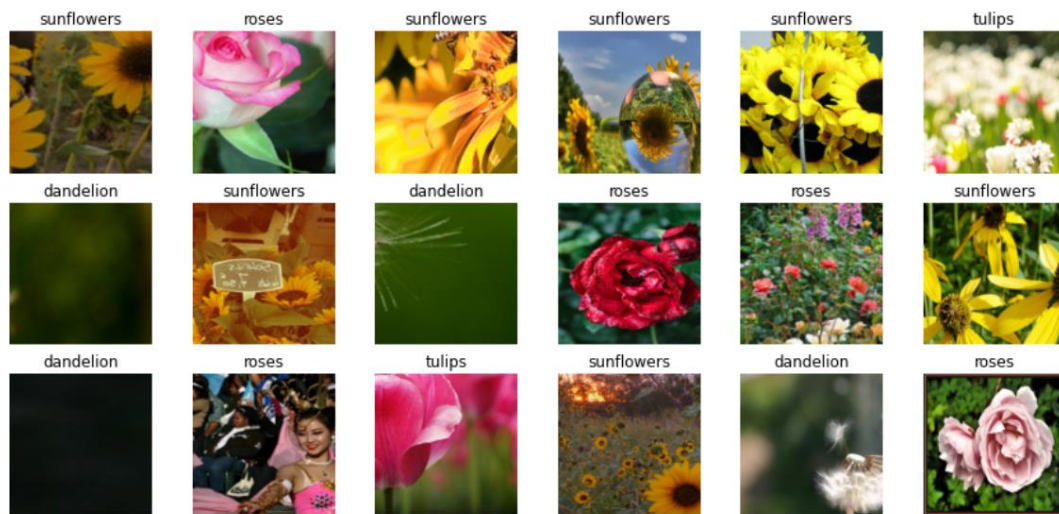
Step 3   Create a MobileNetV2 model.

Datasets are crucial for training. A good dataset can effectively improve training accuracy and efficiency. MobileNet is a lightweight CNN proposed by Google in 2017 to focus on mobile, embedded, and IoT devices. Compared with traditional convolutional neural networks, MobileNet uses depthwise separable convolution to greatly reduce the model parameters and computation amount with a slight decrease in accuracy. In addition, the width coefficient α and resolution coefficient β are introduced to meet the requirements of different application scenarios.

Because a large amount of data is lost when the ReLU activation function in the MobileNet processes low-dimensional feature information, the MobileNetV2 proposes to use an inverted residual block and Linear Bottlenecks to design the network, improving the accuracy of the model and making the optimized model smaller.
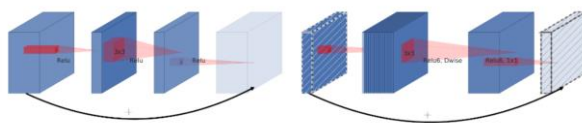
In the inverted residual block structure, the 1 x 1 convolution is used for dimension increase, the 3 x 3 depthwise convolution is used, and the 1 x 1 convolution is used for dimension reduction. This structure is opposite to the residual block structure. For the residual block, the 1 x 1 convolution is first used for dimension reduction, then the 3 x 3 convolution is used, and finally the 1 x 1 convolution is used for dimension increase.

For details, see the MobileNetV2 paper at https://arxiv.org/pdf/1801.04381.pdf.

Code:

```
import numpy as np
import mindspore as ms
```

```python
import mindspore.nn as nn
import mindspore.ops as ops

def _make_divisible(v, divisor, min_value=None):
    if min_value is None:
        min_value = divisor
    new_v = max(min_value, int(v + divisor / 2) // divisor * divisor)
    # Make sure that round down does not go down by more than 10%.
    if new_v < 0.9 * v:
        new_v += divisor
    return new_v


class GlobalAvgPooling(nn.Cell):

    def __init__(self):
        super(GlobalAvgPooling, self).__init__()
        self.mean = ops.ReduceMean(keep_dims=False)

    def construct(self, x):
        x = self.mean(x, (2, 3))
        return x


class ConvBNReLU(nn.Cell):

    def __init__(self, in_planes, out_planes, kernel_size=3, stride=1, groups=1):
        super(ConvBNReLU, self).__init__()
        padding = (kernel_size - 1) // 2
        in_channels = in_planes
        out_channels = out_planes
        if groups == 1:
            conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride, pad_mode='pad',
padding=padding)
        else:
            out_channels = in_planes
            conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride, pad_mode='pad',
                            padding=padding, group=in_channels)

        layers = [conv, nn.BatchNorm2d(out_planes), nn.ReLU6()]
        self.features = nn.SequentialCell(layers)

    def construct(self, x):
        output = self.features(x)
        return output


class InvertedResidual(nn.Cell):

    def __init__(self, inp, oup, stride, expand_ratio):
        super(InvertedResidual, self).__init__()
        assert stride in [1, 2]

        hidden_dim = int(round(inp * expand_ratio))
        self.use_res_connect = stride == 1 and inp == oup
```

```python
        layers = []
        if expand_ratio != 1:
            layers.append(ConvBNReLU(inp, hidden_dim, kernel_size=1))
        layers.extend([
            # dw
            ConvBNReLU(hidden_dim, hidden_dim,
                        stride=stride, groups=hidden_dim),
            # pw-linear
            nn.Conv2d(hidden_dim, oup, kernel_size=1,
                        stride=1, has_bias=False),
            nn.BatchNorm2d(oup),
        ])
        self.conv = nn.SequentialCell(layers)
        self.add = ops.Add()
        self.cast = ops.Cast()

    def construct(self, x):
        identity = x
        x = self.conv(x)
        if self.use_res_connect:
            return self.add(identity, x)
        return x


class MobileNetV2Backbone(nn.Cell):

    def __init__(self, width_mult=1., inverted_residual_setting=None, round_nearest=8,
                    input_channel=32, last_channel=1280):
        super(MobileNetV2Backbone, self).__init__()
        block = InvertedResidual
        # setting of inverted residual blocks
        self.cfgs = inverted_residual_setting
        if inverted_residual_setting is None:
            self.cfgs = [
                # t, c, n, s
                [1, 16, 1, 1],
                [6, 24, 2, 2],
                [6, 32, 3, 2],
                [6, 64, 4, 2],
                [6, 96, 3, 1],
                [6, 160, 3, 2],
                [6, 320, 1, 1],
            ]

        # building first layer
        input_channel = _make_divisible(input_channel * width_mult, round_nearest)
        self.out_channels = _make_divisible(last_channel * max(1.0, width_mult), round_nearest)
        features = [ConvBNReLU(3, input_channel, stride=2)]
        # building inverted residual blocks
        for t, c, n, s in self.cfgs:
            output_channel = _make_divisible(c * width_mult, round_nearest)
            for i in range(n):
                stride = s if i == 0 else 1
                features.append(block(input_channel, output_channel, stride, expand_ratio=t))
```

```python
                input_channel = output_channel
        # building last several layers
        features.append(ConvBNReLU(input_channel, self.out_channels, kernel_size=1))
        # make it nn.CellList
        self.features = nn.SequentialCell(features)
        self._initialize_weights()

    def construct(self, x):
        x = self.features(x)
        return x

    def _initialize_weights(self):

        self.init_parameters_data()
        for _, m in self.cells_and_names():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.set_data(ms.Tensor(np.random.normal(0, np.sqrt(2. / n),

m.weight.data.shape).astype("float32")))

                if m.bias is not None:
                    m.bias.set_data(
                        ms.numpy.zeros(m.bias.data.shape, dtype="float32"))
            elif isinstance(m, nn.BatchNorm2d):
                m.gamma.set_data(
                    ms.Tensor(np.ones(m.gamma.data.shape, dtype="float32")))
                m.beta.set_data(
                    ms.numpy.zeros(m.beta.data.shape, dtype="float32"))

    @property
    def get_features(self):
        return self.features


class MobileNetV2Head(nn.Cell):

    def __init__(self, input_channel=1280, num_classes=1000, has_dropout=False, activation="None"):
        super(MobileNetV2Head, self).__init__()
        # mobilenet head
        head = ([GlobalAvgPooling()] if not has_dropout else
                [GlobalAvgPooling(), nn.Dropout(0.2)])
        self.head = nn.SequentialCell(head)
        self.dense = nn.Dense(input_channel, num_classes, has_bias=True)
        self.need_activation = True
        if activation == "Sigmoid":
            self.activation = ops.Sigmoid()
        elif activation == "Softmax":
            self.activation = ops.Softmax()
        else:
            self.need_activation = False
        self._initialize_weights()

    def construct(self, x):
        x = self.head(x)
```

```
            x = self.dense(x)
            if self.need_activation:
                x = self.activation(x)
            return x

    def _initialize_weights(self):

        self.init_parameters_data()
        for _, m in self.cells_and_names():
            if isinstance(m, nn.Dense):
                m.weight.set_data(ms.Tensor(np.random.normal(
                    0, 0.01, m.weight.data.shape).astype("float32")))
                if m.bias is not None:
                    m.bias.set_data(
                        ms.numpy.zeros(m.bias.data.shape, dtype="float32"))

class MobileNetV2Combine(nn.Cell):

    def __init__(self, backbone, head):
        super(MobileNetV2Combine, self).__init__(auto_prefix=False)
        self.backbone = backbone
        self.head = head

    def construct(self, x):
        x = self.backbone(x)
        x = self.head(x)
        return x

def mobilenet_v2(num_classes):
    backbone_net = MobileNetV2Backbone()
    head_net = MobileNetV2Head(backbone_net.out_channels,num_classes)
    return MobileNetV2Combine(backbone_net, head_net)
```

Step 4   Train and validate the model.

After a model, a loss function and an optimizer are created, the Model API is used to initialize the model, the model.train API is used to train the model, and the model.eval API is used to validate the model accuracy.

This section involves the following knowledge of transfer learning:

1.   Download a pre-trained model weight.

Download the model file pre-trained on the ImageNet dataset, and save it to the directory at the same level as the running code. Download link: https://download.mindspore.cn/models/r1.7/mobilenetv2_ascend_v170_imagenet2012 _official_cv_top1acc71.88.ckpt.

2.   Read the pre-trained model.

Read the pre-trained model file through the load_checkpoint() API. The output result is in dictionary data format.

3.   Modify pre-trained model parameters.

Modify the parameters related to the pre-trained model weight. (The model is pre-trained on the ImageNet dataset to classify 1001 types. However, the current exercise is

to classify five types of flowers. The network model modifies the last fully-connected layer.)

Code:

```
import mindspore
import mindspore.nn as nn
from mindspore.train import Model
from mindspore import Tensor, save_checkpoint
from mindspore.train.callback import ModelCheckpoint, CheckpointConfig, LossMonitor
from mindspore.train.serialization import load_checkpoint, load_param_into_net
# Create a model. The number of target classes is 5.
network = mobilenet_v2(5)

# Load the pre-trained weight.
param_dict =
load_checkpoint("./mobilenetv2_ascend_v170_imagenet2012_official_cv_top1acc71.88.ckpt")

# Modify the weight data based on the modified model structure.
param_dict["dense.weight"] =
mindspore.Parameter(Tensor(param_dict["dense.weight"][:5, :],mindspore.float32),
name="dense.weight", requires_grad=True)
param_dict["dense.bias"] =
mindspore.Parameter(Tensor(param_dict["dense.bias"][:5, ],mindspore.float32), name="dense.bias",
requires_grad=True)

# Load the modified weight parameters to the model.
load_param_into_net(network, param_dict)

train_step_size = dataset_train.get_dataset_size()
epoch_size = 20
lr = nn.cosine_decay_lr(min_lr=0.0, max_lr=0.1,total_step=epoch_size *
train_step_size,step_per_epoch=train_step_size,decay_epoch=200)
# Define the optimizer.
network_opt = nn.Momentum(params=network.trainable_params(), learning_rate=0.01,
momentum=0.9)

# Define the loss function.
network_loss = loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")

# Define evaluation metrics.
metrics = {"Accuracy": nn.Accuracy()}

# Initialize the model.
model = Model(network, loss_fn=network_loss, optimizer=network_opt, metrics=metrics)

# Monitor the loss value.
loss_cb = LossMonitor(per_print_times=train_step_size)

# Set the number of steps for saving a model and the maximum number of models that can be
saved.
ckpt_config = CheckpointConfig(save_checkpoint_steps=100, keep_checkpoint_max=10)

# Save the model. Set the name, path, and parameters for saving the model.
ckpoint_cb = ModelCheckpoint(prefix="mobilenet_v2", directory='./ckpt', config=ckpt_config)
```

```
print("============== Starting Training ==============")
# Train a model, set the number of training times to 5, and set the training set and callback function.
model.train(5, dataset_train, callbacks=[loss_cb,ckpoint_cb], dataset_sink_mode=True)
# Use the test set to validate the model and output the accuracy of the test set.
metric = model.eval(dataset_val)
print(metric)
```

Output:

```
============== Starting Training ==============
epoch: 1 step: 201, loss is 0.8389087915420532
epoch: 2 step: 201, loss is 0.5519619584083557
epoch: 3 step: 201, loss is 0.26490363478660583
epoch: 4 step: 201, loss is 0.4540162682533264
epoch: 5 step: 201, loss is 0.5963617563247681
{'Accuracy': 0.9166666666666666}
```

Step 5   Visualize the model prediction result.

Define the visualize_model function, use the model with the highest validation accuracy described above to predict the input image and visualize the prediction result.

Code:

```
import matplotlib.pyplot as plt
import mindspore as ms

def visualize_model(best_ckpt_path, val_ds):
    num_class = 5   # Perform binary classification on wolf and dog images.
    net = mobilenet_v2(num_class)
    # Load model parameters.
    param_dict = ms.load_checkpoint(best_ckpt_path)
    ms.load_param_into_net(net, param_dict)
    model = ms.Model(net)
    # Load the validation dataset.
    data = next(val_ds.create_dict_iterator())
    images = data["image"].asnumpy()
    labels = data["label"].asnumpy()
    class_name = {0:'daisy',1:'dandelion',2:'roses',3:'sunflowers',4:'tulips'}
    # Predict the image type.
    output = model.predict(ms.Tensor(data['image']))
    pred = np.argmax(output.asnumpy(), axis=1)

    # Display the image and the predicted value of the image.
    plt.figure(figsize=(15, 7))
    for i in range(len(labels)):
        plt.subplot(3, 6, i + 1)
        # If the prediction is correct, it is displayed in blue. If the prediction is incorrect, it is di
splayed in red.
        color = 'blue' if pred[i] == labels[i] else 'red'
        plt.title('predict:{}'.format(class_name[pred[i]]), color=color)
        picture_show = np.transpose(images[i], (1, 2, 0))
        mean = np.array([0.485, 0.456, 0.406])
        std = np.array([0.229, 0.224, 0.225])
        picture_show = std * picture_show + mean
```

```
        picture_show  =  np.clip(picture_show,  0,  1)
        plt.imshow(picture_show)
        plt.axis('off')

    plt.show()

visualize_model('ckpt/mobilenet_v2-5_201.ckpt',  dataset_val)
```

Output:
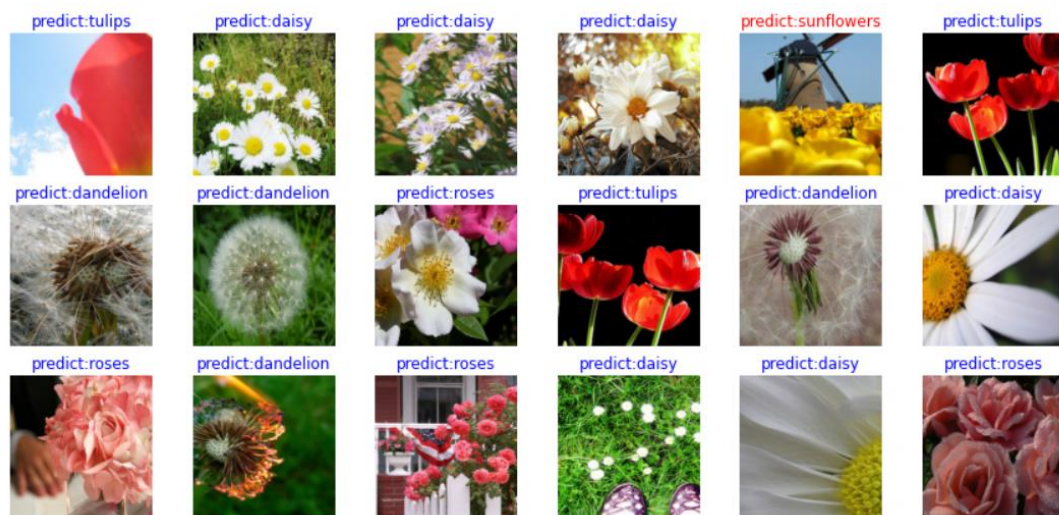
3.4 Question

What API is used to read pre-trained models?

4 ResNet-50 Image Classification

4.1 Introduction

This exercise implements the ResNet-50 image classification, a classic case in the deep learning field. The entire process is as follows:

- Process the required dataset. (The flower image dataset is used in this example.)
- Define a network. You need to set up a ResNet-50 model structure.
- Define a loss function and an optimizer.
- Load the dataset and perform training. After the training is complete, use the test set for validation.

4.2 Preparations

Before you start, check whether MindSpore has been correctly installed. You are advised to install MindSpore on your computer by referring to the MindSpore official website https://www.mindspore.cn/install/en.

In addition, you should have basic mathematical knowledge, including knowledge of Python coding basics, probability, and matrices.

Recommended environment:

Version: MindSpore 1.7

Programming language: Python 3.7

4.3 Detailed Design and Implementation

4.3.1 Data Preparation

The image flower dataset used in the example is an open-source dataset and contains five flower types: daisies (633 images) dandelions (898 images), roses (641 images), sunflowers (699 images), and tulips (799 images). The 3670 photos, which are about 230 MB in total, are stored in five folders. To facilitate the model test after the model deployment, the dataset is divided into flower_photos_train and flower_photos_test.

The directory structure is as follows:

flower_photos_train

```
    ├── daisy
    ├── dandelion
    ├── roses
    ├── sunflowers
    ├── tulips
    ├── LICENSE.txt
flower_photos_test
    ├── daisy
    ├── dandelion
    ├── roses
    ├── sunflowers
    ├── tulips
    ├── LICENSE.txt
```

Obtain the datasets from the following links:

https://ascend-professional-construction-dataset.obs.myhuaweicloud.com/deep-learning/flower_photos_train.zip
https://ascend-professional-construction-dataset.obs.myhuaweicloud.com/deep-learning/flower_photos_test.zip

4.3.2 Procedure

Step 1   Import the Python library and module and configure running information.

Import the required Python library.

For details about the MindSpore modules, see the MindSpore API page.

You can use context.set_context to configure the information required for running, such as the running mode, backend information, and hardware information.

Import the context module and configure the required information.

Code:

```
from easydict import EasyDict as edict
# Dictionary access, used to store hyperparameters
import os
# os module, used to process files and directories
import numpy as np
# Scientific computing library
import matplotlib.pyplot as plt
# Graphing library

import mindspore
# MindSpore library
import mindspore.dataset as ds
# Dataset processing module
```

```
from mindspore.dataset.vision import c_transforms as vision
# Image enhancement module
from mindspore import context
#Environment setting module
import mindspore.nn as nn
# Neural network module
from mindspore.train import Model
# Model build
from mindspore.nn.optim.momentum import Momentum
# Momentum optimizer
from mindspore.train.callback import ModelCheckpoint, CheckpointConfig, LossMonitor
# Model saving settings
from mindspore import Tensor
# Tensor
from mindspore.train.serialization import export
# Model export
from mindspore.train.loss_scale_manager import FixedLossScaleManager
# Loss value smoothing
from mindspore.train.serialization import load_checkpoint, load_param_into_net
# Model loading
import mindspore.ops as ops
# Common operators

# MindSpore execution mode and device setting
context.set_context(mode=context.GRAPH_MODE, device_target="CPU")
```

Step 2   Define parameter variables.

The edict stores the parameter configurations required for model training and testing.

Code:

```
cfg = edict({
    'data_path': 'flowers/flower_photos_train',      # Path of the training dataset
    'test_path':'flowers/flower_photos_train',        # Path of the test dataset
    'data_size': 3616,
    'HEIGHT': 224,    # Image height
    'WIDTH': 224,    # Image width
    '_R_MEAN': 123.68, # Average value of CIFAR-10
    '_G_MEAN': 116.78,
    '_B_MEAN': 103.94,
    '_R_STD': 1, # Customized standard deviation
    '_G_STD': 1,
    '_B_STD':1,
    '_RESIZE_SIDE_MIN': 256, # Minimum resize value for image enhancement
    '_RESIZE_SIDE_MAX': 512,

    'batch_size': 32, # Batch size
    'num_class': 5,        # Number of classes
    'epoch_size': 5,    # Number of training times
    'loss_scale_num':1024,

    'prefix': 'resnet-ai',    # Name of the model
    'directory': './model_resnet',    # Path for storing the model
    'save_checkpoint_steps': 10, # The checkpoint is saved every 10 steps.
})
```

Step 3   Read and process data.

Datasets are crucial for training. A good dataset can effectively improve training accuracy and efficiency. Generally, before loading a dataset, you need to perform some operations on the dataset.

Define a dataset and data operations.

We define the create_dataset function to create a dataset. In this function, we define the data augmentation and processing operations to be performed:

- Read the dataset.
- Define parameters required for data augmentation and processing.
- Generate corresponding data augmentation operations according to the parameters.
- Use the map function to apply data operations to the dataset.
- Process the generated dataset.
- Display the processed data as an example.

Code:

```
# Data processing
def read_data(path,config,usage="train"):
    # Read the source dataset of an image from a directory.
    dataset = ds.ImageFolderDataset(path,

class_indexing={'daisy':0,'dandelion':1,'roses':2,'sunflowers':3,'tulips':4})
    # define map operations
    # Operator for image decoding
    decode_op = vision.Decode()
    # Operator for image normalization
    normalize_op = vision.Normalize(mean=[cfg._R_MEAN, cfg._G_MEAN, cfg._B_MEAN],
std=[cfg._R_STD, cfg._G_STD, cfg._B_STD])
    # Operator for image resizing
    resize_op = vision.Resize(cfg._RESIZE_SIDE_MIN)
    # Operator for image cropping
    center_crop_op = vision.CenterCrop((cfg.HEIGHT, cfg.WIDTH))
    # Operator for image random horizontal flipping
    horizontal_flip_op = vision.RandomHorizontalFlip()
    # Operator for image channel quantity conversion
    channelswap_op = vision.HWC2CHW()
    # Operator for random image cropping, decoding, encoding, and resizing
    random_crop_decode_resize_op = vision.RandomCropDecodeResize((cfg.HEIGHT, cfg.WIDTH),
(0.5, 1.0), (1.0, 1.0), max_attempts=100)

    # Preprocess the training set.
    if usage == 'train':
        dataset = dataset.map(input_columns="image", operations=random_crop_decode_resize_op)
        dataset = dataset.map(input_columns="image", operations=horizontal_flip_op)
    # Preprocess the test set.
    else:
        dataset = dataset.map(input_columns="image", operations=decode_op)
        dataset = dataset.map(input_columns="image", operations=resize_op)
        dataset = dataset.map(input_columns="image", operations=center_crop_op)
```

```
    # Preprocess all datasets.
    dataset = dataset.map(input_columns="image", operations=normalize_op)
    dataset = dataset.map(input_columns="image", operations=channelswap_op)

    # Batch the training set.
    if usage == 'train':
        dataset = dataset.shuffle(buffer_size=10000)    # 10000 as in imageNet train script
        dataset = dataset.batch(cfg.batch_size, drop_remainder=True)
    # Batch the test set.
    else:
        dataset = dataset.batch(1, drop_remainder=True)

    # Data augmentation
    dataset = dataset.repeat(1)

    dataset.map_model = 4

    return dataset

# Display the numbers of training sets and test sets.
de_train = read_data(cfg.data_path,cfg,usage="train")
de_test = read_data(cfg.test_path,cfg,usage="test")
print('Number of training datasets: ',de_train.get_dataset_size()*cfg.batch_size)# get_dataset_size()
obtains the batch processing size.
print('Number of test datasets: ',de_test.get_dataset_size())

# Display the sample graph of the training set.
data_next = de_train.create_dict_iterator(output_numpy=True).__next__()
print('Number of channels/Image length/width: ', data_next['image'][0,...].shape)
print('Label style of an image: ', data_next['label'][0])    # Total 5 label classes which are represented
by numbers from 0 to 4.

plt.figure()
plt.imshow(data_next['image'][0,0,...])
plt.colorbar()
plt.grid(False)
plt.show()
```

Output:

```
Number of training datasets: 3616
Number of test datasets: 32
Number of channels/Image length/width: (3, 224, 224)
Label style of an image: 3
```

Data processing display:
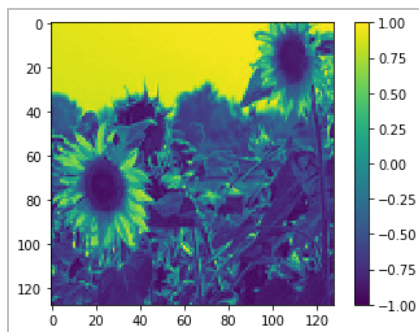
Step 4   Build and train the model.

- Define the model.

Residual blocks

Use the ouput of the first several layers as the input of the last several layers, skipping the intermediate layers. That is, the feature layers have some linear contributions of the first several layers. This design is intended to resolve the problems of impaired learning efficiency and accuracy when the number of network layers increases.

Figure 2. Residual learning: a building block.

(https://arxiv.org/pdf/1512.03385.pdf)
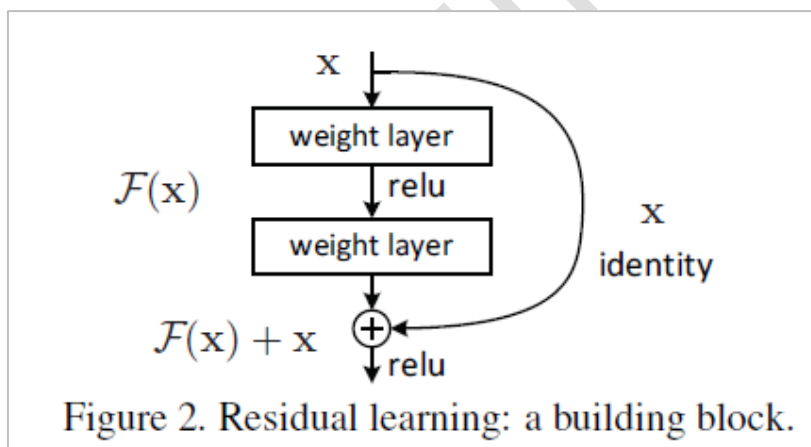
If the dimensions are the same:

$$y = F(x, W_i) + x$$
$$F = W_2 \sigma(W, x)$$

If the dimensions are different:

$$y = F(x, W_i) + W_s x$$

Bottleneck module

The bottleneck module uses the 1x1 convolutional layer to reduce or expand the feature map dimension so that the number of filters at the 3x3 convolutional layer is not affected by the input of the upper layer, and the output of the 3x3 convolutional layer does not affect the lower-layer module.
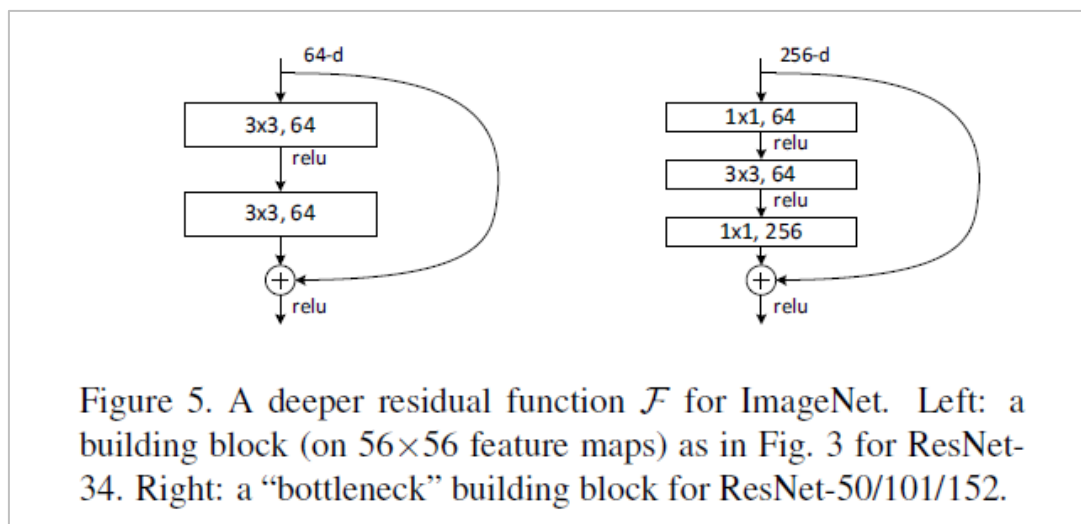
Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

(https://arxiv.org/pdf/1512.03385.pdf)

ResNet-50 model

ResNet-50 has two basic blocks: convolutional block and identity block. The input and output dimensions of the convolutional block are different and cannot be connected in series. The convolutional block is used to change the network dimensions. The identity block has the same input and output dimensions, which can be connected in series to deepen the network.

| layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3{\times}3,\,64 \\ 3{\times}3,\,64 \end{bmatrix}{\times}2$	$\begin{bmatrix} 3{\times}3,\,64 \\ 3{\times}3,\,64 \end{bmatrix}{\times}3$	$\begin{bmatrix} 1{\times}1,\,64 \\ 3{\times}3,\,64 \\ 1{\times}1,\,256 \end{bmatrix}{\times}3$	$\begin{bmatrix} 1{\times}1,\,64 \\ 3{\times}3,\,64 \\ 1{\times}1,\,256 \end{bmatrix}{\times}3$	$\begin{bmatrix} 1{\times}1,\,64 \\ 3{\times}3,\,64 \\ 1{\times}1,\,256 \end{bmatrix}{\times}3$
conv3_x	28×28	$\begin{bmatrix} 3{\times}3,\,128 \\ 3{\times}3,\,128 \end{bmatrix}{\times}2$	$\begin{bmatrix} 3{\times}3,\,128 \\ 3{\times}3,\,128 \end{bmatrix}{\times}4$	$\begin{bmatrix} 1{\times}1,\,128 \\ 3{\times}3,\,128 \\ 1{\times}1,\,512 \end{bmatrix}{\times}4$	$\begin{bmatrix} 1{\times}1,\,128 \\ 3{\times}3,\,128 \\ 1{\times}1,\,512 \end{bmatrix}{\times}4$	$\begin{bmatrix} 1{\times}1,\,128 \\ 3{\times}3,\,128 \\ 1{\times}1,\,512 \end{bmatrix}{\times}8$
conv4_x	14×14	$\begin{bmatrix} 3{\times}3,\,256 \\ 3{\times}3,\,256 \end{bmatrix}{\times}2$	$\begin{bmatrix} 3{\times}3,\,256 \\ 3{\times}3,\,256 \end{bmatrix}{\times}6$	$\begin{bmatrix} 1{\times}1,\,256 \\ 3{\times}3,\,256 \\ 1{\times}1,\,1024 \end{bmatrix}{\times}6$	$\begin{bmatrix} 1{\times}1,\,256 \\ 3{\times}3,\,256 \\ 1{\times}1,\,1024 \end{bmatrix}{\times}23$	$\begin{bmatrix} 1{\times}1,\,256 \\ 3{\times}3,\,256 \\ 1{\times}1,\,1024 \end{bmatrix}{\times}36$
conv5_x	7×7	$\begin{bmatrix} 3{\times}3,\,512 \\ 3{\times}3,\,512 \end{bmatrix}{\times}2$	$\begin{bmatrix} 3{\times}3,\,512 \\ 3{\times}3,\,512 \end{bmatrix}{\times}3$	$\begin{bmatrix} 1{\times}1,\,512 \\ 3{\times}3,\,512 \\ 1{\times}1,\,2048 \end{bmatrix}{\times}3$	$\begin{bmatrix} 1{\times}1,\,512 \\ 3{\times}3,\,512 \\ 1{\times}1,\,2048 \end{bmatrix}{\times}3$	$\begin{bmatrix} 1{\times}1,\,512 \\ 3{\times}3,\,512 \\ 1{\times}1,\,2048 \end{bmatrix}{\times}3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		$1.8{\times}10^9$	$3.6{\times}10^9$	$3.8{\times}10^9$	$7.6{\times}10^9$	$11.3{\times}10^9$

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

(https://arxiv.org/pdf/1512.03385.pdf)

Code:

```
"""ResNet."""

# Define the weight initialization function.
def _weight_variable(shape, factor=0.01):
    init_value = np.random.randn(*shape).astype(np.float32) * factor
    return Tensor(init_value)

# Define the 3x3 convolution layer functions.
def _conv3x3(in_channel, out_channel, stride=1):
    weight_shape = (out_channel, in_channel, 3, 3)
    weight = _weight_variable(weight_shape)
```

```
        return nn.Conv2d(in_channel, out_channel,
                         kernel_size=3, stride=stride, padding=0, pad_mode='same', weight_init=weight)

# Define the 1x1 convolution layer functions.
def _conv1x1(in_channel, out_channel, stride=1):
    weight_shape = (out_channel, in_channel, 1, 1)
    weight = _weight_variable(weight_shape)
    return nn.Conv2d(in_channel, out_channel,
                     kernel_size=1, stride=stride, padding=0, pad_mode='same', weight_init=weight)

# Define the 7x7 convolution layer functions.
def _conv7x7(in_channel, out_channel, stride=1):
    weight_shape = (out_channel, in_channel, 7, 7)
    weight = _weight_variable(weight_shape)
    return nn.Conv2d(in_channel, out_channel,
                     kernel_size=7, stride=stride, padding=0, pad_mode='same', weight_init=weight)

# Define the Batch Norm layer functions.
def _bn(channel):
    return nn.BatchNorm2d(channel, eps=1e-4, momentum=0.9,
                          gamma_init=1, beta_init=0, moving_mean_init=0, moving_var_init=1)

# Define the Batch Norm functions at the last layer.
def _bn_last(channel):
    return nn.BatchNorm2d(channel, eps=1e-4, momentum=0.9,
                          gamma_init=0, beta_init=0, moving_mean_init=0, moving_var_init=1)

# Define the functions of the fully-connected layers.
def _fc(in_channel, out_channel):
    weight_shape = (out_channel, in_channel)
    weight = _weight_variable(weight_shape)
    return nn.Dense(in_channel, out_channel, has_bias=True, weight_init=weight, bias_init=0)

# Construct a residual module.
class ResidualBlock(nn.Cell):
    """
    ResNet V1 residual block definition.

    Args:
        in_channel (int): Input channel.
        out_channel (int): Output channel.
        stride (int): Stride size for the first convolutional layer. Default: 1.

    Returns:
        Tensor, output tensor.

    Examples:
        >>> ResidualBlock(3, 256, stride=2)
    """
    expansion = 4 # In conv2_x--conv5_x, the number of convolution kernels at the first two layers is
one fourth of the number of convolution kernels at the third layer (an output channel).

    def __init__(self, in_channel, out_channel, stride=1):
        super(ResidualBlock, self).__init__()
```

```
        # The number of convolution kernels at the first two layers is equal to a quarter of the
number of convolution kernels at the output channels.
        channel = out_channel // self.expansion

        # Layer 1 convolution
        self.conv1 = _conv1x1(in_channel, channel, stride=1)
        self.bn1 = _bn(channel)
        # Layer 2 convolution
        self.conv2 = _conv3x3(channel, channel, stride=stride)
        self.bn2 = _bn(channel)

        # Layer 3 convolution. The number of convolution kernels is equal to that of output
channels.
        self.conv3 = _conv1x1(channel, out_channel, stride=1)
        self.bn3 = _bn_last(out_channel)

        # ReLU activation layer
        self.relu = nn.ReLU()

        self.down_sample = False

        # When the step is not 1 or the number of output channels is not equal to that of input
channels, downsampling is performed to adjust the number of channels.
        if stride != 1 or in_channel != out_channel:
            self.down_sample = True
        self.down_sample_layer = None
        # Adjust the number of channels using the 1x1 convolution.
        if self.down_sample:
            self.down_sample_layer = nn.SequentialCell([_conv1x1(in_channel, out_channel, stride),
# 1x1 convolution
                                                        _bn(out_channel)])   # Batch Norm
        # Addition operator
        self.add = ops.Add()

    # Construct a residual block.
    def construct(self, x):
        # Input
        identity = x

        # Layer 1 convolution 1x1
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

       # Layer 2 convolution 3x3
        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        # Layer 3 convolution 1x1
        out = self.conv3(out)
        out = self.bn3(out)

        # Change the network dimension.
        if self.down_sample:
```

```
            identity = self.down_sample_layer(identity)

        # Add the residual.
        out = self.add(out, identity)
        # ReLU activation
        out = self.relu(out)

        return out

# Construct a residual network.
class ResNet(nn.Cell):
    """
    ResNet architecture.

    Args:
        block (Cell): Block for network.
        layer_nums (list): Numbers of block in different layers.
        in_channels (list): Input channel in each layer.
        out_channels (list): Output channel in each layer.
        strides (list):   Stride size in each layer.
        num_classes (int): The number of classes that the training images belong to.
    Returns:
        Tensor, output tensor.

    Examples:
        >>> ResNet(ResidualBlock,
        >>>        [3, 4, 6, 3],
        >>>        [64, 256, 512, 1024],
        >>>        [256, 512, 1024, 2048],
        >>>        [1, 2, 2, 2],
        >>>        10)
    """

    # Input parameters: residual block, number of repeated residual blocks, input channel, output
channel, stride, and number of image classes
    def __init__(self, block, layer_nums, in_channels, out_channels, strides, num_classes):
        super(ResNet, self).__init__()
        if not len(layer_nums) == len(in_channels) == len(out_channels) == 4:
            raise ValueError("the length of layer_num, in_channels, out_channels list must be 4!")

        # Layer 1 convolution; convolution kernels: 7x7, input channels: 3; output channels: 64; step:
2
        self.conv1 = _conv7x7(3, 64, stride=2)
        self.bn1 = _bn(64)
        self.relu = ops.ReLU()

        # 3x3 pooling layer; step: 2
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, pad_mode="same")

        # conv2_x residual block
        self.layer1 = self._make_layer(block,
                                       layer_nums[0],
                                       in_channel=in_channels[0],
                                       out_channel=out_channels[0],
                                       stride=strides[0])
```

```python
        # conv3_x residual block
        self.layer2 = self._make_layer(block,
                                       layer_nums[1],
                                       in_channel=in_channels[1],
                                       out_channel=out_channels[1],
                                       stride=strides[1])
        # conv4_x residual block
        self.layer3 = self._make_layer(block,
                                       layer_nums[2],
                                       in_channel=in_channels[2],
                                       out_channel=out_channels[2],
                                       stride=strides[2])
        # conv5_x residual block
        self.layer4 = self._make_layer(block,
                                       layer_nums[3],
                                       in_channel=in_channels[3],
                                       out_channel=out_channels[3],
                                       stride=strides[3])
        # Mean operator
        self.mean = ops.ReduceMean(keep_dims=True)
        # Flatten layer
        self.flatten = nn.Flatten()
        # Output layer
        self.end_point = _fc(out_channels[3], num_classes)

    # Input parameters: residual block, number of repeated residual blocks, input channel, output
channel, and stride
    def _make_layer(self, block, layer_num, in_channel, out_channel, stride):
        """
        Make stage network of ResNet.

        Args:
            block (Cell): Resnet block.
            layer_num (int): Layer number.
            in_channel (int): Input channel.
            out_channel (int): Output channel.
            stride (int): Stride size for the first convolutional layer.

        Returns:
            SequentialCell, the output layer.

        Examples:
            >>> _make_layer(ResidualBlock, 3, 128, 256, 2)
        """
        # Build the residual block of convn_x.

        layers = []

        resnet_block = block(in_channel, out_channel, stride=stride)
        layers.append(resnet_block)

        for _ in range(1, layer_num):
            resnet_block = block(out_channel, out_channel, stride=1)
            layers.append(resnet_block)
```

```
            return nn.SequentialCell(layers)

    # Build a ResNet network.
    def construct(self, x):
        x = self.conv1(x)    # Layer 1 convolution: 7x7; step: 2
        x = self.bn1(x)      # Batch Norm of layer 1
        x = self.relu(x)     # ReLU activation layer
        c1 = self.maxpool(x)   # Max pooling: 3x3; step: 2

        c2 = self.layer1(c1) # conv2_x residual block
        c3 = self.layer2(c2) # conv3_x residual block
        c4 = self.layer3(c3) # conv4_x residual block
        c5 = self.layer4(c4) # conv5_x residual block

        out = self.mean(c5, (2, 3)) # Mean pooling layer
        out = self.flatten(out)   # Flatten layer
        out = self.end_point(out)   # Output layer

        return out

# Build a ResNet-50 network.
def resnet50(class_num=5):
    """
    Get ResNet50 neural network.

    Args:
        class_num (int): Class number.

    Returns:
        Cell, cell instance of ResNet50 neural network.

    Examples:
        >>> net = resnet50(10)
    """
    return ResNet(ResidualBlock,      # Residual block
                  [3, 4, 6, 3],   # Number of residual blocks
                  [64, 256, 512, 1024],   # Input channel
                  [256, 512, 1024, 2048], # Output channel
                  [1, 2, 2, 2],   # Step
                  class_num) # Number of output classes
```

- Start training.

After preprocessing data and defining the network, loss function, and optimizer, start model training. Model training involves two iterations: multi-epoch iteration of datasets and single-step iteration based on the batch size. The single-step iteration refers to extracting data from a dataset by batch, inputting the data to a network to calculate a loss function, and then calculating and updating a gradient of training parameters by using an optimizer.

1. Download a pre-trained model.

    Create the **model_resnet** directory, download the model file pre-trained on the ImageNet dataset, and save the file to the **model_resnet** directory. Download link: https://download.mindspore.cn/models/r1.7/resnet50_ascend_v170_imagenet2012_official_cv_top1acc76.97_top5acc93.44.ckpt.

2.   Load the pre-trained model.

Read the pre-trained model file through the load_checkpoint() API to obtain the parameter file in dictionary format.

3.   Modify pre-trained model parameters.

Modify the connection parameters of the last layer of the pre-trained model parameters. (The model is pre-trained on the ImageNet dataset to classify 1001 types. However, the current exercise is to classify the five types of flowers.) Therefore, you need to modify the parameters of the last fully-connected layer.

Code:

```
# Construct a ResNet-50 network. The number of output classes is 5, corresponding to five flower classes.
net=resnet50(class_num=cfg.num_class)

# Read the parameters of the pre-trained model.
param_dict =
load_checkpoint("model_resnet/resnet50_ascend_v170_imagenet2012_official_cv_top1acc76.97_top5acc93.44.ckpt")

# Display the read model parameters.
print(param_dict)

# Modify the shape corresponding to end_point.weight and end_point.bias by using mindspore.Parameter().
param_dict["end_point.weight"] =
mindspore.Parameter(Tensor(param_dict["end_point.weight"][:5, :], mindspore.float32),
name="variable")
param_dict["end_point.bias"]= mindspore.Parameter(Tensor(param_dict["end_point.bias"][:5,],
mindspore.float32), name="variable")

# Set the Softmax cross-entropy loss function.
loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")

# Set the learning rate.
train_step_size = de_train.get_dataset_size()
lr = nn.cosine_decay_lr(min_lr=0.0001, max_lr=0.001, total_step=train_step_size *
cfg.epoch_size,step_per_epoch=train_step_size, decay_epoch=cfg.epoch_size)

# Set the momentum optimizer.
opt = Momentum(net.trainable_params(), lr, momentum=0.9, weight_decay=1e-4,
loss_scale=cfg.loss_scale_num)

# Smooth the loss value to solve the problem of the gradient being too small during training.
loss_scale = FixedLossScaleManager(cfg.loss_scale_num, False)

# Build the model. Input the network structure, loss function, optimizer, loss value smoothing, and model evaluation metrics.
model = Model(net, loss_fn=loss, optimizer=opt, loss_scale_manager=loss_scale, metrics={'acc'})

# Loss value monitoring
loss_cb = LossMonitor(per_print_times=train_step_size)
```

```
# Model saving parameters. Set the number of steps for saving a model and the maximum number
of models that can be saved.
ckpt_config = CheckpointConfig(save_checkpoint_steps=cfg.save_checkpoint_steps,
keep_checkpoint_max=1)

# Save the model. Set the name, path, and parameters for saving the model.
ckpoint_cb = ModelCheckpoint(prefix=cfg.prefix, directory=cfg.directory, config=ckpt_config)

print("============== Starting Training ==============")
# Train the model. Set the training times, training set, callback function, and whether to use the data
offloading mode (can be applied on Ascend and GPUs to accelerate training speed).
model.train(cfg.epoch_size, de_train, callbacks=[loss_cb,ckpoint_cb], dataset_sink_mode=True)
# The training takes 15 to 20 minutes.

# Use the test set to validate the model and output the accuracy of the test set.
metric = model.eval(de_test)
print(metric)
```

Output:

```
            ============== Starting Training ==============
epoch: 1 step: 113, loss is 0.23505911231040955
epoch: 2 step: 113, loss is 0.11479882150888443
epoch: 3 step: 113, loss is 0.13273288309574127
epoch: 4 step: 113, loss is 0.42304447293281555
epoch: 5 step: 113, loss is 0.14625898003578186
{'acc': 0.9587912087912088}
```

Step 5   Use the model for prediction.

Use the trained weight file to call model.predict() to test the test data and output the prediction result and actual result.

Code:

```
# Model prediction. Select 10 samples from the test set for testing and output the prediction result
and actual result.
class_names = {0:'daisy',1:'dandelion',2:'roses',3:'sunflowers',4:'tulips'}
for i in range(10):
    test_ = de_test.create_dict_iterator().__next__()
    test = Tensor(test_['image'], mindspore.float32)
    # Use the model for prediction.
    predictions = model.predict(test)
    predictions = predictions.asnumpy()
    true_label = test_['label'].asnumpy()
    # Show the prediction result.
    p_np = predictions[0, :]
    pre_label = np.argmax(p_np)
    print('Prediction result of the ' + str(i) + '-th sample: ', class_names[pre_label], '    Actual result: ',
class_names[true_label[0]])
```

Output:

Prediction result of sample 0: sunflowers   Actual result: sunflowers
Prediction result of sample 1: daisy   Actual result: daisy
Prediction result of sample 2: roses   Actual result: roses
Prediction result of sample 3: roses   Actual result: roses
Prediction result of sample 4: roses   Actual result: roses
Prediction result of sample 5: dandelion   Actual result: dandelion
Prediction result of sample 6: dandelion   Actual result: dandelion
Prediction result of sample 7: roses   Actual result: roses
Prediction result of sample 8: tulips   Actual result: tulips
Prediction result of sample 9: dandelion   Actual result: dandelion

4.4 Question

In this exercise, how many convolutional and fully-connected layers does the ResNet-50 have? How many epochs are defined for model training? How many classes are output for the network?

5 TextCNN Sentiment Analysis

5.1 Introduction

This exercise implements the TextCNN sentiment analysis, a classic case in the deep learning field. The entire process is as follows:

- Download the required dataset. (The rt-polarity dataset is used to define the data preprocessing function.)

- Generate data for training and validation.

- Define the TextCNN model structure build, training, validation, offline model loading, and online inference functions.

- Define various parameters required for training, such as optimizer, loss function, checkpoint, and time monitor.

- Load the dataset and perform training. After the training is complete, use the test set for validation.

5.2 Preparations

Before you start, check whether MindSpore has been correctly installed. You are advised to install MindSpore on your computer by referring to the MindSpore official website https://www.mindspore.cn/install/en.

In addition, you should have basic mathematical knowledge, including basic knowledge of Python coding basics, probability, and matrices.

Recommended environment:

Version: MindSpore 1.7

Programming language: Python 3.7

5.3 Detailed Design and Implementation

5.3.1 Data Preparation

The rt-polarity used in this example contains English movie reviews. The **rt-polarity.pos** file stores 5000 positive reviews, and the **rt-polarity.neg** file stores 5000 negative reviews.

Download link: https://certification-data.obs.cn-north-4.myhuaweicloud.com/CHS/HCIA-AI/V3.5/chapter4/TextCNN.zip

The data directory structure is as follows:

```
└─data
        ├── rt-polarity.pos
        └── rt-polarity.neg
```

5.3.2 Procedure

Step 1   Import the Python library and module and configure running information.

Import the required Python library.

Currently, the **os** library is required. Other required libraries will not be described here. For details about the MindSpore modules, see the MindSpore API page. You can use context.set_context to configure the information required for running, such as the running mode, backend information, and hardware information.

Import the context module and configure the required information.

Code:

```python
import math
import numpy as np
import pandas as pd
import os
import math
import random
import codecs
from pathlib import Path

import mindspore
import mindspore.dataset as ds
import mindspore.nn as nn
from mindspore import Tensor
from mindspore import context
from mindspore.train.model import Model
from mindspore.nn.metrics import Accuracy
from mindspore.train.serialization import load_checkpoint, load_param_into_net
from mindspore.train.callback import ModelCheckpoint, CheckpointConfig, LossMonitor, TimeMonitor
from mindspore.ops import operations as ops

from easydict import EasyDict as edict


cfg = edict({
    'name': 'movie review',
    'pre_trained': False,
    'num_classes': 2,
    'batch_size': 64,
    'epoch_size': 4,
    'weight_decay': 3e-5,
    'data_path': './data/',
    'device_target': 'CPU',
    'device_id': 0,
    'keep_checkpoint_max': 1,
```

```
        'checkpoint_path': './ckpt/train_textcnn-4_149.ckpt',
        'word_len': 51,
        'vec_length': 40
})


context.set_context(mode=context.GRAPH_MODE, device_target=cfg.device_target,
device_id=cfg.device_id)
```

The graph mode is used in this exercise. You can configure hardware information as required. For example, if the code runs on the Ascend AI processor, set **device_target** to **Ascend**. This rule also applies to the code running on the CPU and GPU. For details about the parameters, see the context.set_context API description at https://www.mindspore.cn/docs/en/r1.7/api_python/mindspore.context.html.

Step 2   Read and process data.

Data preview code:

```
with open("./data/rt-polarity.neg", 'r', encoding='utf-8') as f:
        print("Negative reivews:")
        for i in range(5):
            print("[{0}]:{1}".format(i,f.readline()))
with open("./data/rt-polarity.pos", 'r', encoding='utf-8') as f:
        print("Positive reivews:")
        for i in range(5):
            print("[{0}]:{1}".format(i,f.readline()))
```

Data preview output:

```
Negative reivews:
[0]:simplistic , silly and tedious .

[1]:it's so laddish and juvenile , only teenage boys could possibly find it funny .

[2]:exploitative and largely devoid of the depth or sophistication that would make watching such a graphic treatment of the crimes bearable .

[3]:[garbus] discards the potential for pathological study , exhuming instead , the skewed melodrama of the circumstantial situation .

[4]:a visually flashy but narratively opaque and emotionally vapid exercise in style and mystification .

Positive reivews:
[0]:the rock is destined to be the 21st century's new " conan " and that he's going to make a splash even greater than arnold schwarzenegger , jean-claud van damme or steven segal .

[1]:the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge that a column of words cannot adequately describe co-writer/director peter jackson's expanded vision of j . r . r . tolkien's middle-earth .

[2]:effective but too-tepid biopic

[3]:if you sometimes like to go to the movies to have fun , wasabi is a good place to start .

[4]:emerges as something rare , an issue movie that's so honest and keenly observed that it doesn't feel like one .
```

Code for defining the data processing function:

```
# Define the data generation class.
class Generator():
    def __init__(self, input_list):
        self.input_list=input_list
    def __getitem__(self,item):
        return (np.array(self.input_list[item][0],dtype=np.int32),
                np.array(self.input_list[item][1],dtype=np.int32))
    def __len__(self):
        return len(self.input_list)



class MovieReview:
```

```python
'''
Movie review dataset
'''
def __init__(self, root_dir, maxlen, split):
    '''
    input:
        root_dir: movie review data directory
        maxlen: maximum length of a sentence
        split: the training/validation ratio in the dataset
    '''
    self.path = root_dir
    self.feelMap = {
        'neg':0,
        'pos':1
    }
    self.files = []

    self.doConvert = False

    mypath = Path(self.path)
    if not mypath.exists() or not mypath.is_dir():
        print("please check the root_dir!")
        raise ValueError

    # Find the files in the data directory.
    for root,_,filename in os.walk(self.path):
        for each in filename:
            self.files.append(os.path.join(root,each))
        break

    # Check whether there are two files: .neg and .pos.
    if len(self.files) != 2:
        print("There are {} files in the root_dir".format(len(self.files)))
        raise ValueError


    # Read data.
    self.word_num = 0
    self.maxlen = 0
    self.minlen = float("inf")
    self.maxlen = float("-inf")
    self.Pos = []
    self.Neg = []
    for filename in self.files:
        self.read_data(filename)

    self.text2vec(maxlen=maxlen)
    self.split_dataset(split=split)

def read_data(self, filePath):
    with open(filePath,'r') as f:
        for sentence in f.readlines():
            sentence = sentence.replace('\n','')\
                               .replace('"','')\
```

```
                                                    .replace('\",")\
                                                    .replace('.',")\
                                                    .replace(',',")\
                                                    .replace('[',")\
                                                    .replace(']',")\
                                                    .replace('(',")\
                                                    .replace(')',")\
                                                    .replace(':',")\
                                                    .replace('--',")\
                                                    .replace('-',' ')\
                                                    .replace('\\',")\
                                                    .replace('0',")\
                                                    .replace('1',")\
                                                    .replace('2',")\
                                                    .replace('3',")\
                                                    .replace('4',")\
                                                    .replace('5',")\
                                                    .replace('6',")\
                                                    .replace('7',")\
                                                    .replace('8',")\
                                                    .replace('9',")\
                                                    .replace('`',")\
                                                    .replace('=',")\
                                                    .replace('$',")\
                                                    .replace('/',")\
                                                    .replace('*',")\
                                                    .replace(';',")\
                                                    .replace('<b>',")\
                                                    .replace('%',")


                        sentence = sentence.split(' ')
                        sentence = list(filter(lambda x: x, sentence))
                        if sentence:
                            self.word_num += len(sentence)
                            self.maxlen = self.maxlen if self.maxlen >= len(sentence) else len(sentence)
                            self.minlen = self.minlen if self.minlen <= len(sentence) else len(sentence)
                            if 'pos' in filePath:
                                self.Pos.append([sentence,self.feelMap['pos']])
                            else:
                                self.Neg.append([sentence,self.feelMap['neg']])

    def text2vec(self, maxlen):
        '''
# Convert sentences into vectors.
        '''
        # Vocab = {word : index}
        self.Vocab = dict()

        # self.Vocab['None']
        for SentenceLabel in self.Pos+self.Neg:
            vector = [0]*maxlen
            for index, word in enumerate(SentenceLabel[0]):
                if index >= maxlen:
                    break
```

```
                    if word not in self.Vocab.keys():
                        self.Vocab[word] = len(self.Vocab)
                        vector[index] = len(self.Vocab) - 1
                    else:
                        vector[index] = self.Vocab[word]
                SentenceLabel[0] = vector
        self.doConvert = True

def split_dataset(self, split):
'''
Divide the dataset into a training set and a test set.
'''
        trunk_pos_size = math.ceil((1-split)*len(self.Pos))
        trunk_neg_size = math.ceil((1-split)*len(self.Neg))
        trunk_num = int(1/(1-split))
        pos_temp=list()
        neg_temp=list()
        for index in range(trunk_num):
            pos_temp.append(self.Pos[index*trunk_pos_size:(index+1)*trunk_pos_size])
            neg_temp.append(self.Neg[index*trunk_neg_size:(index+1)*trunk_neg_size])
        self.test = pos_temp.pop(2)+neg_temp.pop(2)
        self.train = [i for item in pos_temp+neg_temp for i in item]

random.shuffle(self.train)

    def get_dict_len(self):
        '''
        Obtain the length of a dictionary consisting of characters in a dataset.
        '''
        if self.doConvert:
            return len(self.Vocab)
        else:
            print("Haven't finished Text2Vec")
            return -1

    def create_train_dataset(self, epoch_size, batch_size):
        dataset = ds.GeneratorDataset(
                                        source=Generator(input_list=self.train),
                                        column_names=["data","label"],
                                        shuffle=False
                                        )
        dataset=dataset.batch(batch_size=batch_size,drop_remainder=True)
        dataset=dataset.repeat(epoch_size)
        return dataset

    def create_test_dataset(self, batch_size):
        dataset = ds.GeneratorDataset(
                                        source=Generator(input_list=self.test),
                                        column_names=["data","label"],
                                        shuffle=False
                                        )
        dataset=dataset.batch(batch_size=batch_size,drop_remainder=True)
        return dataset
```

Use the customized read_data function to load the original dataset and use the text2vec function to vectorize the text of the data. Use split_dataset to split data into training data and test data, and then create_test_dataset and create_train_dataset call mindspore.dataset. GeneratorDataset generates data for training and validation.

Configuration code:

```
instance = MovieReview(root_dir=cfg.data_path, maxlen=cfg.word_len, split=0.9)
dataset = instance.create_train_dataset(batch_size=cfg.batch_size,epoch_size=cfg.epoch_size)
batch_num = dataset.get_dataset_size()
```

Code for displaying the data processing result:

```
vocab_size=instance.get_dict_len()
print("vocab_size:{0}".format(vocab_size))
item =dataset.create_dict_iterator()
for i,data in enumerate(item):
    if i<1:
        print(data)
        print(data['data'][1])
    else:
        break
```

Output the data processing result:

```
vocab_size:18848
{'data': Tensor(shape=[64, 51], dtype=Int32, value=
[[  15, 3514,   10 ...    0,    0,    0],
 [   0,  889, 1305 ...    0,    0,    0],
 [ 128,   15, 1836 ...    0,    0,    0],
 ...
 [3429, 2678,  203 ...    0,    0,    0],
 [   0,   77,    2 ...    0,    0,    0],
 [5277, 5278,  997 ...    0,    0,    0]]), 'label': Tensor(shape=[64], dtype=Int32, value= [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])}
[   0  889 1305 5099   10   88 2409 1755 1588  186  672    0 5100   32
    0   86 5101   14   82 1059  147    0 1024    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0]
```

Step 3   Configure training parameters.

Code for setting the learning rate:

```
learning_rate = []
warm_up = [1e-3 / math.floor(cfg.epoch_size / 5) * (i + 1) for _ in range(batch_num)
              for i in range(math.floor(cfg.epoch_size / 5))]
shrink = [1e-3 / (16 * (i + 1)) for _ in range(batch_num)
            for i in range(math.floor(cfg.epoch_size * 3 / 5))]
normal_run = [1e-3 for _ in range(batch_num) for i in
                 range(cfg.epoch_size - math.floor(cfg.epoch_size / 5)
                      - math.floor(cfg.epoch_size * 2 / 5))]
learning_rate = learning_rate + warm_up + normal_run + shrink
```

The dynamic learning rate in training is implemented through epoch.

Step 4   Define a TextCNN model.

Model classes define model structure build, training, validation, offline model loading, and online inference functions.

Code:

```
def _weight_variable(shape, factor=0.01):
    init_value = np.random.randn(*shape).astype(np.float32) * factor
    return Tensor(init_value)

def make_conv_layer(kernel_size):
    weight_shape = (96, 1, *kernel_size)
    weight = _weight_variable(weight_shape)
    return nn.Conv2d(in_channels=1, out_channels=96, kernel_size=kernel_size, padding=1,
                     pad_mode="pad", weight_init=weight, has_bias=True)

class TextCNN(nn.Cell):
    def __init__(self, vocab_len, word_len, num_classes, vec_length):
        super(TextCNN, self).__init__()
        self.vec_length = vec_length
        self.word_len = word_len
        self.num_classes = num_classes

        self.unsqueeze = ops.ExpandDims()
        self.embedding = nn.Embedding(vocab_len, self.vec_length, embedding_table='normal')

        self.slice = ops.Slice()
        self.layer1 = self.make_layer(kernel_height=3)
        self.layer2 = self.make_layer(kernel_height=4)
        self.layer3 = self.make_layer(kernel_height=5)

        self.concat = ops.Concat(1)

        self.fc = nn.Dense(96*3, self.num_classes)
        self.drop = nn.Dropout(keep_prob=0.5)
        self.print = ops.Print()
        self.reducemean = ops.ReduceMax(keep_dims=False)

    def make_layer(self, kernel_height):
        return nn.SequentialCell(
            [
                make_conv_layer((kernel_height,self.vec_length)),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=(self.word_len-kernel_height+1,1)),
            ]
        )

    def construct(self,x):
        x = self.unsqueeze(x, 1)
        x = self.embedding(x)
        x1 = self.layer1(x)
        x2 = self.layer2(x)
        x3 = self.layer3(x)

        x1 = self.reducemean(x1, (2, 3))
        x2 = self.reducemean(x2, (2, 3))
        x3 = self.reducemean(x3, (2, 3))

        x = self.concat((x1, x2, x3))
```

```
            x = self.drop(x)
            x = self.fc(x)
            return x
net = TextCNN(vocab_len=instance.get_dict_len(), word_len=cfg.word_len,
               num_classes=cfg.num_classes, vec_length=cfg.vec_length)
print(net)
```

View the neural network structure.

Step 5   Define training parameters.

A loss function is also called an objective function and is used to measure the difference between a predicted value and an actual value. Deep learning reduces the loss value by continuous iteration. Defining a good loss function can effectively improve model performance.

An optimizer is used to minimize the loss function, improving the model during training.

After the loss function is defined, the weight-related gradient of the loss function can be obtained. The gradient is used to indicate the weight optimization direction for the optimizer, improving model performance. Loss functions supported by MindSpore include SoftmaxCrossEntropyWithLogits, L1Loss, and MSELoss. SoftmaxCrossEntropyWithLogits is used in this example.

MindSpore provides the callback mechanism to execute custom logic during training. The following uses ModelCheckpoint provided by the framework as an example. ModelCheckpoint can save the network model and parameters for subsequent fine-tuning.

Code:

```
# Optimizer, loss function, checkpoint, and time monitor settings
opt = nn.Adam(filter(lambda x: x.requires_grad, net.get_parameters()),
               learning_rate=learning_rate, weight_decay=cfg.weight_decay)
loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
model = Model(net, loss_fn=loss, optimizer=opt, metrics={'acc': Accuracy()})
config_ck = CheckpointConfig(save_checkpoint_steps=int(cfg.epoch_size*batch_num/2),
keep_checkpoint_max=cfg.keep_checkpoint_max)
time_cb = TimeMonitor(data_size=batch_num)
ckpt_save_dir = "./ckpt"
ckpoint_cb = ModelCheckpoint(prefix="train_textcnn", directory=ckpt_save_dir, config=config_ck)
```

```
loss_cb = LossMonitor()
```

Step 6   Start training.

The training process refers to a process in which training dataset is transferred to a network for training and optimizing network parameters. In the MindSpore framework, the Model.train method is used to complete this process.

Code:

```
model.train(cfg.epoch_size, dataset, callbacks=[time_cb, ckpoint_cb, loss_cb])
print("train success")
```

Output:

```
epoch: 1 step: 596, loss is 0.023292765
epoch time: 36818.071 ms, per step time: 61.775 ms
epoch: 2 step: 596, loss is 0.0016317479
epoch time: 4320.621 ms, per step time: 7.249 ms
epoch: 3 step: 596, loss is 0.00025401893
epoch time: 4271.667 ms, per step time: 7.167 ms
epoch: 4 step: 596, loss is 0.0001774891
epoch time: 4332.598 ms, per step time: 7.269 ms
train success
```

Step 7   Test and validate data.

A single piece of review text data is obtained for testing, and a sentiment category and a probability of the sentiment category are output.

Code:

```
def preprocess(sentence):
    sentence = sentence.lower().strip()
    sentence = sentence.replace('\n','')\
                        .replace('"','')\
                        .replace('\'','')\
                        .replace('.','')\
                        .replace(',','')\
                        .replace('[','')\
                        .replace(']','')\
                        .replace('(','')\
                        .replace(')','')\
                        .replace(':','')\
                        .replace('--','')\
                        .replace('-',' ')\
                        .replace('\\','')\
                        .replace('0','')\
                        .replace('1','')\
                        .replace('2','')\
                        .replace('3','')\
                        .replace('4','')\
                        .replace('5','')\
                        .replace('6','')\
                        .replace('7','')\
```

```
                                      .replace('8','')\
                                      .replace('9','')\
                                      .replace('`','')\
                                      .replace('=','')\
                                      .replace('$','')\
                                      .replace('/','')\
                                      .replace('*','')\
        .replace(';','')\
                                      .replace('<b>','')\
                                      .replace('%','')\
                                      .replace("   "," ")
    sentence = sentence.split(' ')
    maxlen = cfg.word_len
    vector = [0]*maxlen
    for index, word in enumerate(sentence):
        if index >= maxlen:
            break
        if word not in instance.Vocab.keys():
            print(word,"The word does not appear in the dictionary.")
        else:
            vector[index] = instance.Vocab[word]
    sentence = vector

    return sentence

def inference(review_en):
    review_en = preprocess(review_en)
    input_en = Tensor(np.array([review_en]).astype(np.int32))
    output = net(input_en)
    if np.argmax(np.array(output[0])) == 1:
        print("Positive comments")
    else:
        print("Negative comments")
```

Code:

```
review_en = "the movie is so boring"
inference(review_en)
```

Output:

```
Negative comments
```

5.4 Question

If you want to perform the preceding exercise on a GPU, which configuration item do you need to change to GPU?