Huawei AI Certification Training

HCIA-AI

Machine Learning Lab Guide

ISSUE: 3.5



HUAWEI TECHNOLOGIES CO., LTD

**1**

Copyright © Huawei Technologies Co., Ltd. 2023. All rights reserved.

###### No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Technologies Co., Ltd.

Trademarks and Permissions

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

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

Notice

###### The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

###### The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

### Huawei Technologies Co., Ltd.

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

###### Website: https://[e](http://e.huawei.com/).huawei.com

**Huawei Certification System**

###### Huawei Certification is an integral part of the company's "Platform + Ecosystem" strategy, and it supports the ICT infrastructure featuring "Cloud-Pipe-Device". It evolves to reflect the latest trends of ICT development. Huawei Certification consists of three categories: ICT Infrastructure Certification, Basic Software & Hardware Certification and Cloud Platform & Services Certification, making it the most extensive technical certification program in the industry.

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

###### Huawei Certification covers all ICT fields and adapts to the industry trend of ICT convergence.

###### With its leading talent development system and certification standards, it is committed to fostering new ICT talent in the digital era, and building a sound ICT talent ecosystem.

###### HCIA-AI V3.5 certification is intended for cultivating and conducting qualification of engineers who are capable of creatively designing and developing AI products and solutions using machine learning and deep learning algorithms.

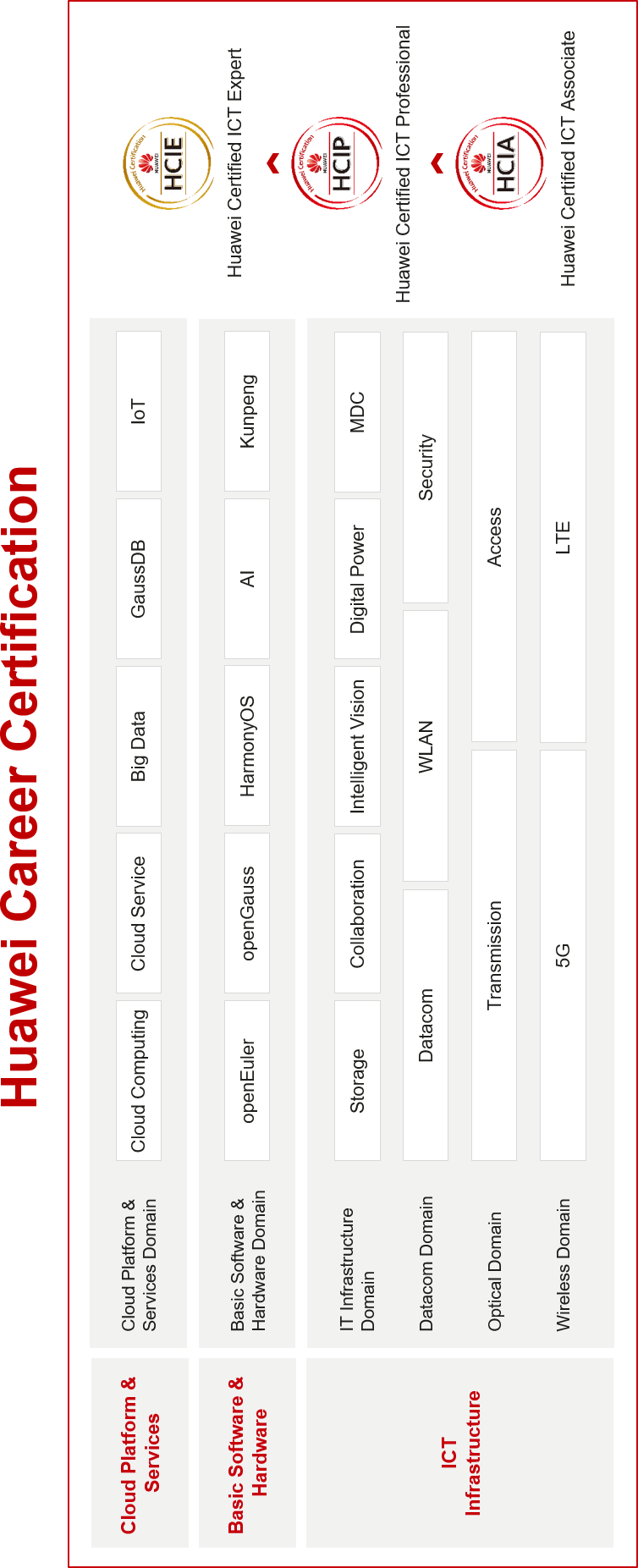
###### An HCIA-AI V3.5 certificate proves that you:

###### Have understood the development history of AI, Huawei Ascend AI system, and Huawei full-stack AI strategy in all scenarios;

###### Have mastered traditional machine learning and deep learning

###### Are able to use the MindSpore framework to build, train, and deploy neural networks;

###### Are competent in sales, marketing, product manager, project management, and technical support positions in the AI field.



# About This Document

Overview

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

Description

This lab guide comprises the following five experiments:

* + Experiment 1 - Linear regression. The Python tool package scikit-learn is used to implement a simple linear regression algorithm.
  + Experiment 2 - Linear regression expansion. The Python basic tool package numpy is used to implement algorithms such as linear regression and gradient descent from scratch.
  + Experiment 3 - Logistic regression. The logistic regression algorithm in the tool package is used to implement simple classification.
  + Experiment 4 - Decision tree. Trainees will construct a decision tree to predict the weather, and visualize the decision tree.
  + Experiment 5 - K-means clustering algorithm.

Background Knowledge Required

This course is for Huawei's professional certification. Trainees are expected to meet the following requirements:

* + Have basic knowledge of Python programming.
  + Have basic math skills, including linear algebra and probability theory.

Lab Environment Overview

The lab environment is Python 3.7. For details about how to set up the environment, see the Environment Setup Guide.

Datasets used in this lab can be obtained at [https://certification-data.obs.cn-north-](https://certification-data.obs.cn-north-4.myhuaweicloud.com/ENG/HCIA-AI/V3.5/chapter2/ML.zip) [4.myhuaweicloud.com/ENG/HCIA-AI/V3.5/chapter2/ML.zip](https://certification-data.obs.cn-north-4.myhuaweicloud.com/ENG/HCIA-AI/V3.5/chapter2/ML.zip)

**Contents**

##### [About This Document 3](#_bookmark0)

[Overview 3](#_bookmark1)

[Description 3](#_bookmark2)

[Background Knowledge Required 3](#_bookmark3)

[Lab Environment Overview 3](#_bookmark4)

##### [Implementation of Common Machine Learning Algorithms 5](#_bookmark5)

* 1. [Introduction 5](#_bookmark6)
     1. [About This Lab 5](#_bookmark7)
     2. [Objectives 5](#_bookmark8)
  2. [Code Implementation 5](#_bookmark9)
     1. [Linear Regression 5](#_bookmark10)
     2. [Linear Regression Implementation (Expansion Experiment) 7](#_bookmark11)
     3. [Logistic Regression 10](#_bookmark12)
     4. [Decision Tree 12](#_bookmark13)
     5. [K-means Algorithm Implementation 14](#_bookmark14)
  3. [Question 18](#_bookmark15)
  4. [Summary 18](#_bookmark16)

1. **Implementation of Common Machine Learning**

**Algorithms**

* 1. Introduction
     1. About This Lab

This lab introduces common machine learning algorithms to help you better understand their functions and usages. Specifically, it will explain how to build a linear regression algorithm from scratch and implement the decision tree and K-means clustering based on scikit-learn.

* + 1. Objectives
       - Build a linear regression algorithm from scratch.
       - Master the use of classification and regression algorithms.
       - Master the use of the V clustering algorithm.
       - Master the implementation process of machine learning algorithms.
  1. Code Implementation
     1. Linear Regression

Step 1 Import dependent packages. Input:

from sklearn.linear\_model import LinearRegression # Import the linear regression model import matplotlib.pyplot as plt # The plotting library

import numpy as np

Step 2 Build and visualize a house price dataset. Input:

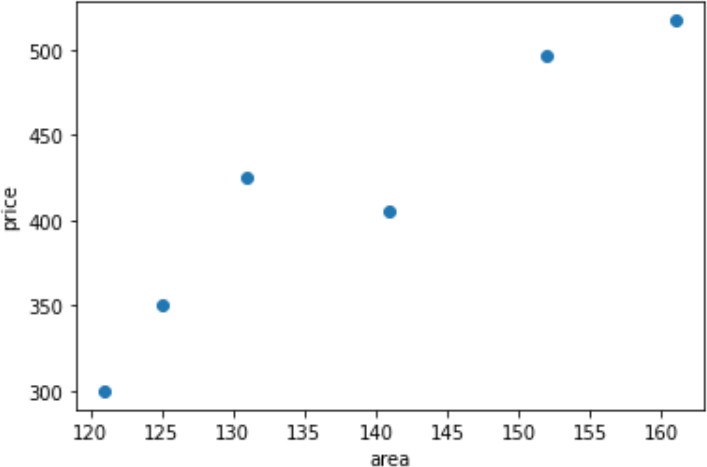
x = np.array([121, 125, 131, 141, 152, 161]).reshape(-1,1) # x denotes the house area as a feature. y = np.array([300, 350, 425, 405,496,517]) # y denotes the house price.

plt.scatter(x,y)

plt.xlabel("area") # X axis indicates the area. plt.ylabel("price") # Y axis indicates the price.

plt.show()

Output:



Step 3 Train the model. Input:

lr = LinearRegression() # Encapsulate the linear regression model into an object.

lr.fit(x,y) # Train the model on the dataset.

Step 4 Visualize the model. Input:

w = lr.coef\_# Slope of the model

b = lr.intercept\_# Intercept of the model print('Slope: ',w)

print('Intercept: ',b)

Output:

Slope: [4.98467124]

Intercept: -274.8769665187576

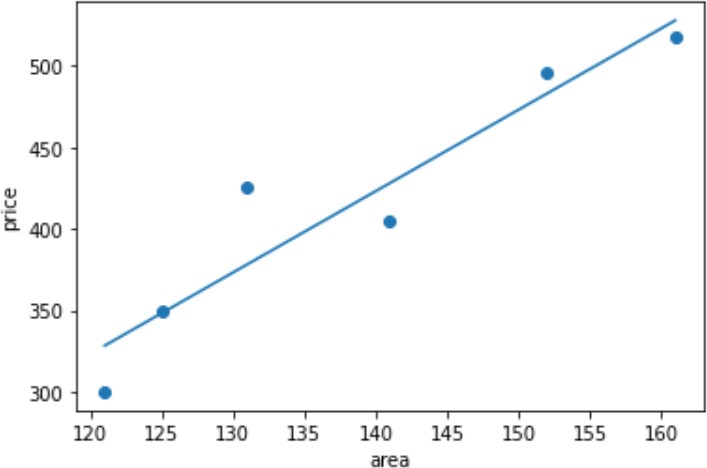
Input:

plt.scatter(x,y)

plt.xlabel("area") # X axis indicates the area. plt.ylabel("price") # Y axis indicates the price.

plt.plot([x[0],x[-1]],[x[0]\*w+b,x[-1]\*w+b])

Output:



Step 5 Start a prediction task using the model. Input:

testX = np.array([[130]])# A test sample with an area of 130

lr.predict(testX)

Output:

Array([373.13029447]) # The model predicts the house price of the sample.

* + 1. Linear Regression Implementation (Expansion Experiment)

This experiment uses the **lr2\_data.txt** dataset, which contains simulated house area and price data. To obtain the dataset, see the Lab Environment Setup.

Step 1 Import dependencies. Input:

import numpy as np

import matplotlib.pyplot as plt

Step 2 Define the function for calculating gradients. Input:

def generate\_gradient(X, theta, y): sample\_count = X.shape[0]

# Calculate the gradient based on the matrix 1/m ∑(((h(x^i)-y^i)) x\_j^i)

return (1./sample\_count)\*X.T.dot(X.dot(theta)-y)

Step 3 Define the function for reading datasets. Input:

def get\_training\_data(file\_path):

orig\_data = np.loadtxt(file\_path,skiprows=1) # Ignore the title in the first row of the dataset. cols = orig\_data.shape[1]

return (orig\_data, orig\_data[:, :cols - 1], orig\_data[:, cols-1:])

Step 4 Define the function for initializing parameters. Input:

# Initialize the θ array.

def init\_theta(feature\_count):

return np.ones(feature\_count).reshape(feature\_count, 1)

Step 5 Define the function for implementing gradient descent. Input:

def gradient\_descending(X, y, theta, alpha):

Jthetas= [] # Record the change trend of the cost function J(θ) to confirm the gradient descent is correct.

# Calculate the loss function, which is equal to the square of the difference between the actual value and the predicted value: (y^i-h(x^i))^2

Jtheta = (X.dot(theta)-y).T.dot(X.dot(theta)-y) index = 0

gradient = generate\_gradient(X, theta, y) # Calculate the gradient.

while not np.all(np.absolute(gradient) <= 1e-5): # End the calculation when the gradient is less than 0.00001. theta = theta - alpha \* gradient

gradient = generate\_gradient(X, theta, y) # Calculate the new gradient.

# Calculate the loss function, which is equal to the square of the difference between the actual value and the predicted value: (y^i-h(x^i))^2

Jtheta = (X.dot(theta)-y).T.dot(X.dot(theta)-y) if (index+1) % 10 == 0:

Jthetas.append((index, Jtheta[0])) # Record the result every 10 calculations. index += 1

return theta,Jthetas

Step 6 Define the function for visualizing the change curve of the loss function. Input:

# Plot the loss function change curve. def showJTheta(diff\_value):

p\_x = []

p\_y = []

for (index, sum) in diff\_value: p\_x.append(index) p\_y.append(sum)

plt.plot(p\_x, p\_y, color='b') plt.xlabel('steps') plt.ylabel('loss funtion')

plt.title('step - loss function curve')

plt.show()

Step 7 Define the function for visualizing data points and the fitted curve. Input:

# Plot the actual data points and the fitted curve. def showlinercurve(theta, sample\_training\_set):

x, y = sample\_training\_set[:, 1], sample\_training\_set[:, 2] z = theta[0] + theta[1] \* x

plt.scatter(x, y, color='b', marker='x',label="sample data") plt.plot(x, z, 'r', color="r",label="regression curve") plt.xlabel('x')

plt.ylabel('y')

plt.title('liner regression curve') plt.legend()

plt.show()

Step 8 Plot the final results. Input:

# Read the dataset.

training\_data\_include\_y, training\_x, y = get\_training\_data("./ML/02/lr2\_data.txt") # Obtain the numbers of samples and features, respectively.

sample\_count, feature\_count = training\_x.shape

# Define the learning step α.

alpha = 0.01

# Initialize θ.

theta = init\_theta(feature\_count)

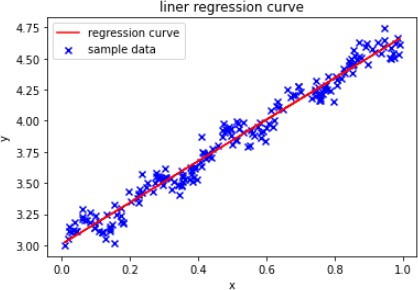
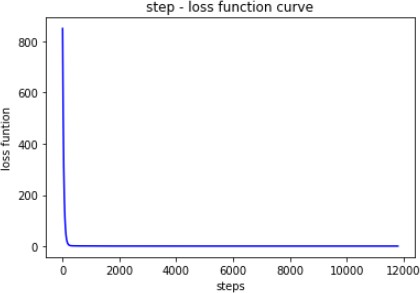
# Obtain the final parameter θ and cost.

result\_theta,Jthetas = gradient\_descending(training\_x, y, theta, alpha) # Display the parameter. print("w:{}".format(result\_theta[0][0]),"b:{}".format(result\_theta[1][0])) showJTheta(Jthetas)

showlinercurve(result\_theta, training\_data\_include\_y)

Output (ignore warnings):

w:3.0076279423997594 b:1.668677412281192



* + 1. Logistic Regression

This experiment uses a custom dataset of house rent and area. The dataset is defined during the initial phase of the experiment.

Step 1 Import dependencies. Input:

# Import StandardScaler from sklearn.preprocessing. from sklearn.preprocessing import StandardScaler

# Import LogisticRegression from sklearn.linear\_model.

from sklearn.linear\_model import LogisticRegression

Step 2 Define the dataset. Input:

# Each item in X denotes the rent and area.

# y indicates whether to rent the room (0: no; 1: yes). X=[[2200,15],[2750,20],[5000,40],[4000,20],[3300,20],[2000,10],[2500,12],[12000,80],

[2880,10],[2300,15],[1500,10],[3000,8],[2000,14],[2000,10],[2150,8],[3400,20],

[5000,20],[4000,10],[3300,15],[2000,12],[2500,14],[10000,100],[3150,10],

[2950,15],[1500,5],[3000,18],[8000,12],[2220,14],[6000,100],[3050,10]

]

y=[1,1,0,0,1,1,1,1,0,1,1,0,1,1,0,1,0,0,0,1,1,1,0,1,0,1,0,1,1,0]

Step 3 Preprocess data.

Standardize data to ensure that the variance of feature data in each dimension is 1 and the mean is

0. In this way, the prediction result is not dominated by large feature values of some dimensions. Input:

ss = StandardScaler()

X\_train = ss.fit\_transform(X)

Display the standardized data. Input:

print(X\_train)

Output:

[[-0.60583897 -0.29313058]

[-0.37682768 -0.09050576]

[ 0.56003671 0.71999355]

[ 0.14365254 -0.09050576]

[-0.14781638 -0.09050576]

[-0.68911581 -0.49575541]

[-0.48092372 -0.41470548]

[ 3.47472592 2.34099218]

[-0.32269773 -0.49575541]

[-0.56420055 -0.29313058]

[-0.89730789 -0.49575541]

[-0.27273163 -0.57680534]

[-0.68911581 -0.33365555]

[-0.68911581 -0.49575541]

[-0.62665818 -0.57680534]

[-0.10617796 -0.09050576]

[ 0.56003671 -0.09050576]

[ 0.14365254 -0.49575541]

[-0.14781638 -0.29313058]

[-0.68911581 -0.41470548]

[-0.48092372 -0.33365555]

[ 2.64195758 3.15149149]

[-0.21027401 -0.49575541]

[-0.29355084 -0.29313058]

[-0.89730789 -0.69838024]

[-0.27273163 -0.17155569]

[ 1.80918923 -0.41470548]

[-0.59751129 -0.33365555]

[ 0.97642089 3.15149149]

[-0.25191242 -0.49575541]]

Step 4 Fit the data.

Input:

# Use the fit method of LogisticRegression to train model parameters. lr = LogisticRegression()

lr.fit(X\_train, y)

Output:

LogisticRegression()

Step 5 Predict the data.

Input:

testX = [[2000,8]]

X\_test = ss.transform(testX) print("Value to be predicted: ",X\_test) label = lr.predict(X\_test) print("predicted label = ", label)

# Output the predicted probability.

prob = lr.predict\_proba(X\_test) print("probability = ",prob)

Output:

Value to be predicted: [[-0.68911581 -0.57680534]] predicted label = [1]

probability = [[0.41886952 0.58113048]]

* + 1. Decision Tree

This experiment uses the **tennis.txt** dataset, which contains 14 samples. Each sample contains weather-related features and whether it is suitable for tennis.

Step 1 Import dependencies. Input:

import pandas as pd import numpy as np from sklearn import tree

import pydotplus

Step 2 Define the function for generating a decision tree. Input:

# Generate a decision tree. def createTree(trainingData):

data = trainingData.iloc[:, :-1] # Feature matrix labels = trainingData.iloc[:, -1] # Labels

trainedTree = tree.DecisionTreeClassifier(criterion="entropy") # Decision tree classifier trainedTree.fit(data, labels) # Train the model.

return trainedTree

Step 3 Define the function for saving the generated tree diagram. Input:

def showtree2pdf(trainedTree,finename):

dot\_data = tree.export\_graphviz(trainedTree, out\_file=None) # Export the tree in Graphviz format. graph = pydotplus.graph\_from\_dot\_data(dot\_data)

graph.write\_pdf(finename) # Save the tree diagram to the local machine in PDF format.

Step 4 Define the function for generating vectorized data.

In the function, **pd.Categorical(list).codes** obtains the sequence number list corresponding to the original data, so as to convert the categorical information into numeric information.

Input:

def data2vectoc(data):

names = data.columns[:-1] for i in names:

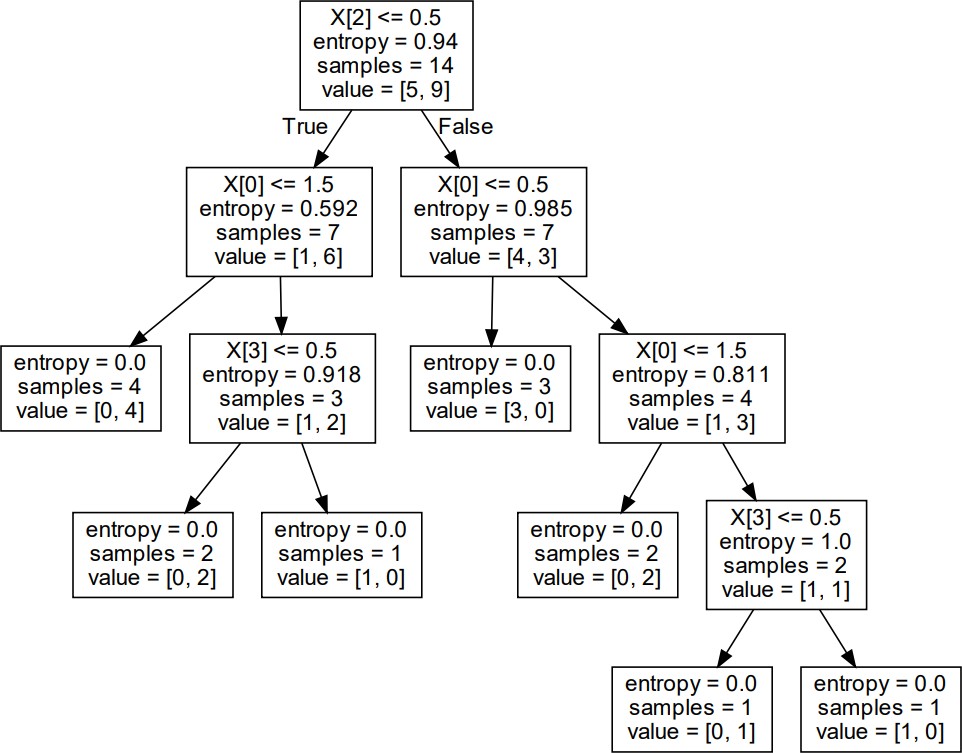
col = pd.Categorical(data[i]) data[i] = col.codes

return data

Step 5 Invoke the function for prediction. Input:

data = pd.read\_table("./ML/tennis.txt",header=None,sep='\t') # Read training data. trainingvec=data2vectoc(data) # Vectorize data. decisionTree=createTree(trainingvec) # Create a decision tree.

showtree2pdf(decisionTree,"tennis.pdf") # Plot the decision tree.

A decision tree diagram named **tennis.pdf** is generated on the local machine.

The file content is a visualized display of the decision tree. In the diagram, **X[2]** is the third feature variable (humidity); **X[0]** is the first feature variable (weather); **X[3]** is the fourth feature variable (wind); **entropy** is the entropy value of the node; and **samples** is the number of samples in the node, for example, 14 in the first node (root node) indicates the number of samples in the training set; and **value** indicates the numbers of samples of different types, for example, in the root node, 5 indicates the number of "no" samples, and 9 indicates the number of "yes" samples.

Predict a new sample. Input:

testVec = [0,0,1,1] # Weather is sunny, temperature is low, humidity is high, and wind is strong.

print(decisionTree.predict(np.array(testVec).reshape(1,-1))) # Predict.

Output:

['N']

* + 1. K-means Algorithm Implementation

Step 1 Import dependencies.

**make\_blobs** is used to generate the dataset required for this experiment, **matplotlib** is used for data visualization, and **KMeans** is used for fitting data based on the K-means algorithm.

Input:

from sklearn.datasets import make\_blobs import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

Step 2 Generate the dataset.

**make\_blobs()** is used to generate a dataset for a clustering task by returning the generated dataset and labels. Note: the K-means algorithm is often used to process unlabeled datasets in production. The generated dataset contains labels only to facilitate understanding.

Input:

X, y = make\_blobs(n\_samples=500,n\_features=2,centers=4,random\_state=1)

**n\_samples** is the number of samples, **n\_features** is the number of features of each sample (the dimension of data), **centers** is the number of categories, and **random\_state** is the seed for the random number generator. The seed ensures that data generated each time is the same.

Display the dimension information of the generated dataset. Input:

Print("Dimension of X is {}".format(X.shape))

Print("Dimension of y is {}".format(y.shape))

Output:

Dimension of X is (500, 2)

Dimension of y is (500,).

The dataset contains 500 sample points, and each sample point contains two features.

Step 3 Draw scatter graphs.

First, directly draw a scatter graph of the dataset. The labels generated during dataset generation are not considered for now. The graph contains four different clusters.

Input:

fig, ax1 = plt.subplots(1) ax1.scatter(X[:, 0], X[:, 1]

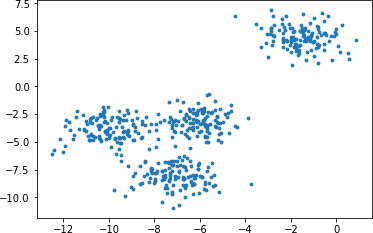
,marker='o' # Set the shape of the point to circle.

,s=8 # Set the size of the point.

)

plt.show()

Output:



Next, draw a scatter graph where points in different colors correspond to the generated labels. Input:

color = ["red","pink","orange","green"] fig, ax1 = plt.subplots(1)

for i in range(4):

ax1.scatter(X[y==i, 0], X[y==i, 1] # Draw the color based on the label.

,marker='o' # Set the shape of the point to circle.

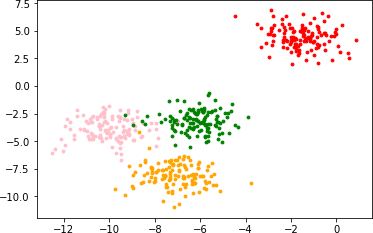
,s=8 # Set the size of the point.

,c=color[i]

)

plt.show()

Output:



Step 4 Perform k-means clustering.

Use **sklearn.cluster.KMeans** provided by scikit-learn to implement K-means clustering. First, aggregate the data samples into three types.

Input:

n\_clusters = 3

cluster1 = KMeans(n\_clusters=n\_clusters,random\_state=3).fit(X)

View the label of each sample point after clustering. Input:

y\_pred1 = cluster1.labels\_

print(y\_pred1)

Output:

[0 0 2 1 2 1 2 2 2 2 0 0 2 1 2 0 2 0 1 2 2 2 2 1 2 2 1 1 2 2 0 1 2 0 2 0 2

2 0 2 2 2 1 2 2 0 2 2 1 1 1 2 2 2 0 2 2 2 2 2 1 1 2 2 1 2 0 2 2 2 0 2 2 0

2 2 0 2 2 2 1 1 2 1 1 2 2 1 2 2 1 0 2 2 1 0 0 2 0 1 1 0 1 2 1 2 2 1 1 2 2

0 1 2 1 2 1 2 1 2 2 0 0 2 2 2 1 0 0 2 1 2 2 2 2 0 1 2 1 1 2 0 2 1 1 1 2 2

0 0 2 2 1 0 1 2 2 2 2 2 2 2 2 2 1 0 0 0 2 1 0 2 2 0 1 2 2 2 2 0 2 2 1 0 0

2 2 0 0 2 1 1 0 0 2 1 2 0 0 1 0 2 1 2 2 0 2 2 0 2 2 2 2 0 2 2 2 1 2 1 2 0

2 2 2 2 2 1 2 1 0 2 0 2 1 1 2 0 1 0 2 2 0 0 0 0 2 2 0 2 2 1 1 2 2 1 2 2 2

1 2 1 2 2 1 2 0 0 2 2 2 2 1 1 2 1 2 0 1 0 1 0 0 1 0 1 1 2 2 2 2 2 2 2 0 1

0 0 0 2 2 2 0 2 0 0 2 0 0 2 1 0 2 2 1 1 2 0 1 1 2 0 1 1 2 2 1 2 2 0 0 1 2

0 2 1 1 2 2 2 0 2 1 1 2 1 1 1 1 0 0 2 1 2 2 0 1 2 1 2 1 2 2 2 1 2 2 0 1 0

0 0 0 0 0 2 0 1 0 1 1 2 1 2 2 2 0 1 2 1 2 0 2 2 0 2 2 1 1 0 2 2 1 2 2 0 0

2 0 2 2 0 2 0 2 1 0 1 2 2 1 2 2 1 0 2 1 1 2 2 2 2 0 1 0 2 1 0 0 0 2 1 2 0

2 2 2 2 0 2 2 2 2 2 2 0 2 2 0 2 1 2 1 2 2 2 1 1 1 2 2 2 0 2 1 2 0 1 0 1 0

2 1 1 0 2 2 0 2 2 2 0 2 1 2 2 0 0 0 2]

Print the centroid of each cluster. Input:

centroid1 = cluster1.cluster\_centers\_

print(centroid1)

Output:

[[-7.09306648 -8.10994454]

[-1.54234022 4.43517599]

[-8.0862351 -3.5179868 ]]

Visualize the clustering result. Input:

color = ["red","pink","orange","gray"] fig, ax1 = plt.subplots(1)

for i in range(n\_clusters): ax1.scatter(X[y\_pred1==i, 0], X[y\_pred1==i, 1]

,marker='o' # Set the shape of the point to circle.

,s=8 # Set the size of the point.

,c=color[i]

)

ax1.scatter(centroid1[:,0],centroid1[:,1]

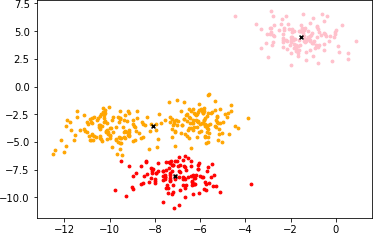
,marker="x"

,s=15

,c="black")

plt.show()

Output:



The graph shows that the data samples are aggregated into three clusters, and the centroid of each cluster is represented by **x**.

Step 5 Perform k-means clustering again.

Next, perform the preceding steps to use k-means clustering to aggregate data samples into four types.

Input:

n\_clusters = 4

cluster2 = KMeans(n\_clusters=n\_clusters,random\_state=0).fit(X) y\_pred2 = cluster2.labels\_

centroid2 = cluster2.cluster\_centers\_

print("Centroid: {}".format(centroid2))

Output:

|  |  |
| --- | --- |
| [ -1.54234022 | 4.43517599] |
| [ -7.09306648 | -8.10994454] |
| [-10.00969056 | -3.84944007]] |

Visualize the clustering result. Input:

Centroid: [[ -6.08459039 -3.17305983]

color = ["red","pink","orange","green"] fig, ax1 = plt.subplots(1)

for i in range(n\_clusters): ax1.scatter(X[y\_pred2==i, 0], X[y\_pred2==i, 1]

,marker='o' # Set the shape of the point to circle.

,s=8 # Set the size of the point.

,c=color[i]

)

ax1.scatter(centroid2[:,0],centroid2[:,1]

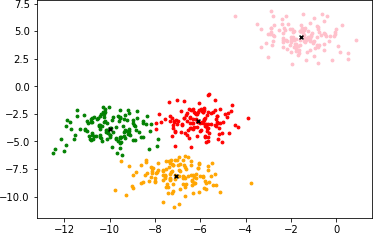
,marker="x"

,s=15

,c="black")

plt.show()

Output:



The graph shows that the data samples are aggregated into four clusters, and the centroid of each cluster is represented by **x**. Compare the scatter graphs generated based on the original data and data aggregated into four types. It can be seen that many sample points are aggregated into wrong clusters.

* 1. Question

How to implement K-means from scratch using Python?

* 1. Summary

This lab introduces the machine learning implementation process, including data import, segmentation, standardization, as well as models and hyperparameters. Common machine learning algorithms are implemented based on scikit-learn to help you better understand how to build and use machine learning models.

Huawei AI Certification Training

HCIA-AI

Deep Learning and AI Development Framework Lab Guide

ISSUE: 3.5



HUAWEI TECHNOLOGIES CO., LTD.

**1**

Copyright © Huawei Technologies Co., Ltd. 2023. All rights reserved.

###### No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Technologies Co., Ltd.

Trademarks and Permissions

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

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

Notice

###### The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

###### The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

### Huawei Technologies Co., Ltd.

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

###### Website: https://e.huawei.com

Huawei Certification System

###### Huawei Certification is an integral part of the company's "Platform + Ecosystem" strategy, and it supports the ICT infrastructure featuring "Cloud-Pipe-Device". It evolves to reflect the latest trends of ICT development. Huawei Certification consists of three categories: ICT Infrastructure Certification, Basic Software & Hardware Certification and Cloud Platform & Services Certification, making it the most extensive technical certification program in the industry.

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

###### Huawei Certification covers all ICT fields and adapts to the industry trend of ICT convergence.

###### With its leading talent development system and certification standards, it is committed to fostering new ICT talent in the digital era, and building a sound ICT talent ecosystem.

###### HCIA-AI V3.5 certification is intended for cultivating and conducting qualification of engineers who are capable of creatively designing and developing AI products and solutions using machine learning and deep learning algorithms.

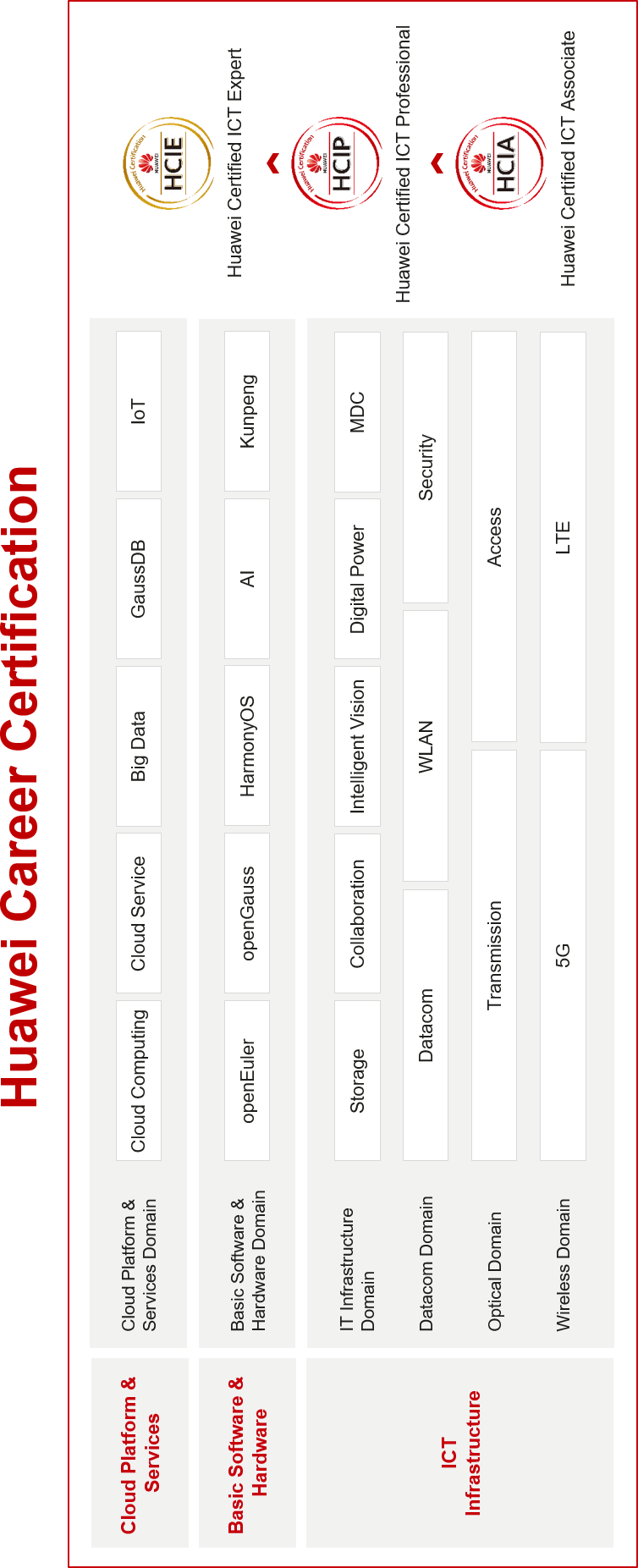
###### An HCIA-AI V3.5 certificate proves that you:

###### Have understood the development history of AI, Huawei Ascend AI system, and Huawei full-stack AI strategy in all scenarios;

###### Have mastered traditional machine learning and deep learning

###### Are able to use the MindSpore framework to build, train, and deploy neural networks;

###### Are competent in sales, marketing, product manager, project management, and technical support positions in the AI field.



**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**

[About This Document 3](#_bookmark17)

1. [MindSpore Basics 6](#_bookmark18)
   1. [Introduction 6](#_bookmark19)
      1. [About This Lab 6](#_bookmark20)
      2. [Objectives 6](#_bookmark21)
      3. [Lab Environment 6](#_bookmark22)
   2. [Procedure 6](#_bookmark23)
      1. [Introduction to Tensors 6](#_bookmark24)
      2. [Loading a Dataset 10](#_bookmark25)
      3. [Building the Network 14](#_bookmark26)
      4. [Training and Validating a Model 16](#_bookmark27)
      5. [Saving and Loading a Model 18](#_bookmark28)
      6. [Automatic Differentiation 19](#_bookmark29)
   3. [Question 20](#_bookmark30)
2. [MNIST Handwritten Character Recognition 21](#_bookmark31)
   1. [Introduction 21](#_bookmark32)
      1. [About This Exercise 21](#_bookmark33)
   2. [Preparations 21](#_bookmark34)
   3. [Detailed Design and Implementation 21](#_bookmark35)
      1. [Data Preparation 21](#_bookmark36)
      2. [Procedure 22](#_bookmark37)
3. [MobileNetV2 Image Classification 27](#_bookmark38)
   1. [Introduction 27](#_bookmark39)
   2. [Preparations 27](#_bookmark40)
   3. [Detailed Design and Implementation 27](#_bookmark41)
      1. [Data Preparation 27](#_bookmark42)
      2. [Procedure 28](#_bookmark43)
   4. [Question 37](#_bookmark44)
4. [ResNet-50 Image Classification 38](#_bookmark45)
   1. [Introduction 38](#_bookmark46)
   2. [Preparations 38](#_bookmark47)
   3. [Detailed Design and Implementation 38](#_bookmark48)
      1. [Data Preparation 38](#_bookmark49)
      2. [Procedure 39](#_bookmark50)
   4. [Question 51](#_bookmark51)
5. [TextCNN Sentiment Analysis 52](#_bookmark52)
   1. [Introduction 52](#_bookmark53)
   2. [Preparations 52](#_bookmark54)
   3. [Detailed Design and Implementation 52](#_bookmark55)
      1. [Data Preparation 52](#_bookmark56)
      2. [Procedure 53](#_bookmark57)
   4. [Question 62](#_bookmark58)

# 1 MindSpore Basics

## Introduction

* + 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. Objectives
       - Master the method of creating tensors.
       - Master the attributes and methods of tensors.
    2. 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.

## Procedure

* + 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.

#### 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.]]

#### 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

#### 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]])

#### 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. 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 [https://certification-data.obs.cn-north-](https://certification-data.obs.cn-north-4.myhuaweicloud.com/CHS/HCIA-AI/V3.5/chapter4/MNIST.zip) [4.myhuaweicloud.com/CHS/HCIA-AI/V3.5/chapter4/MNIST.zip](https://certification-data.obs.cn-north-4.myhuaweicloud.com/CHS/HCIA-AI/V3.5/chapter4/MNIST.zip) 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:

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.

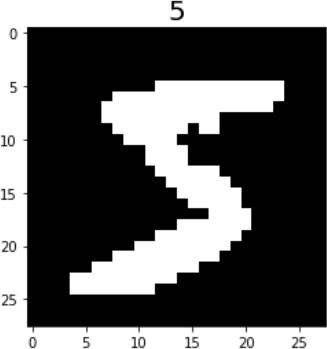
mnist\_dataset = ds.MnistDataset(DATA\_DIR, num\_samples=6, shuffle=False) # View the original image data.

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:



Code:

##### Figure 1-2 Data sample

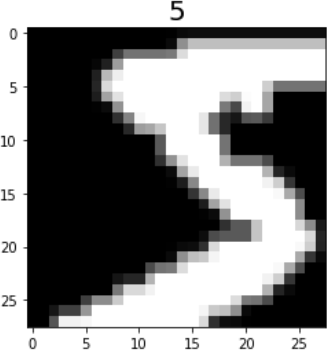
resize\_op = c\_vision.Resize(size=(40,40), interpolation=Inter.LINEAR) crop\_op = c\_vision.RandomCrop(28)

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. 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. 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. 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

1. 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. 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

[4.5099998 2.7

3.6000001]

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]),)

## 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

## Introduction

* + 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.

## 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.](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

## Detailed Design and Implementation

* + 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/>(OBS: https://certification- data.obs.cn-north-4.myhuaweicloud.com/CHS/HCIA-AI/V3.5/chapter4/MNIST.zip). 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

* + 1. Procedure

train-images.idx3-ubyte train-labels.idx1-ubyte

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.](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)

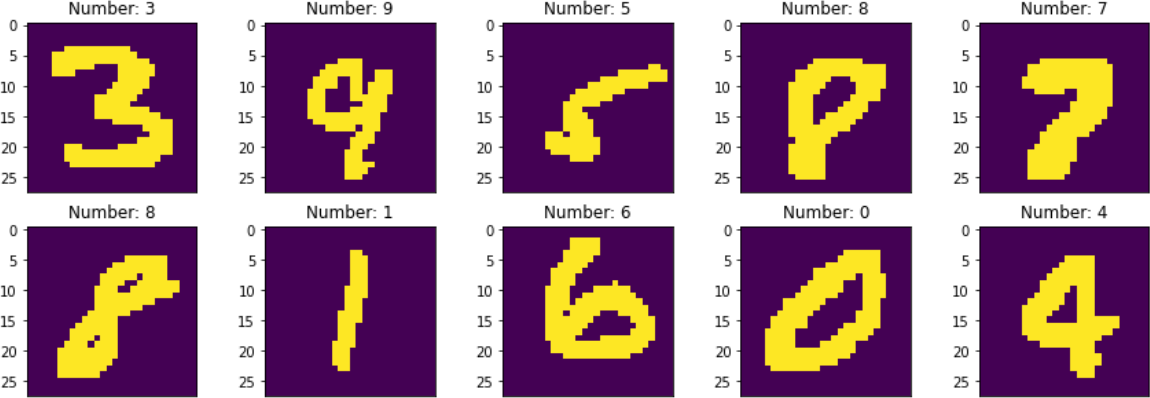
data = ds.create\_dict\_iterator(). next () images = data['image'].asnumpy()

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

## Introduction

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

## 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.](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

## Detailed Design and Implementation

* + 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

* + 1. 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:

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()

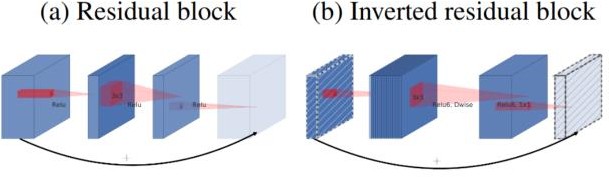
Output:

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 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

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)) 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\_officia l\_cv\_top1acc71.88.ckpt.

* + - 1. Read the pre-trained model.

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

* + - 1. 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 displaye

d 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:



## Question

What API is used to read pre-trained models?

# 4 ResNet-50 Image Classification

## 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.

## 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.](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

## Detailed Design and Implementation

* + 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

* + 1. 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()

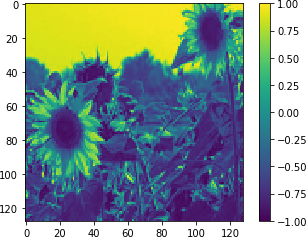
plt.grid(False) plt.show()

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

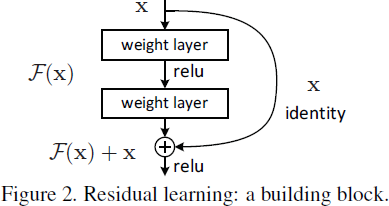
Data processing display:



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.



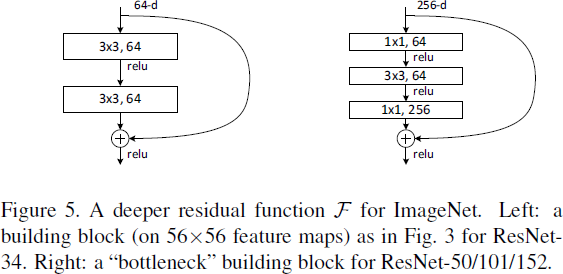
(https://arxiv.org/pdf/1512.03385.pdf) If the dimensions are the same:

If the dimensions are different: Bottleneck module

###### y = F(x, 𝑊𝑖 ) + 𝑥 F = 𝑊2𝜎(𝑊, 𝑥)

###### y = F(x, 𝑊𝑖) + 𝑊𝑠𝑥

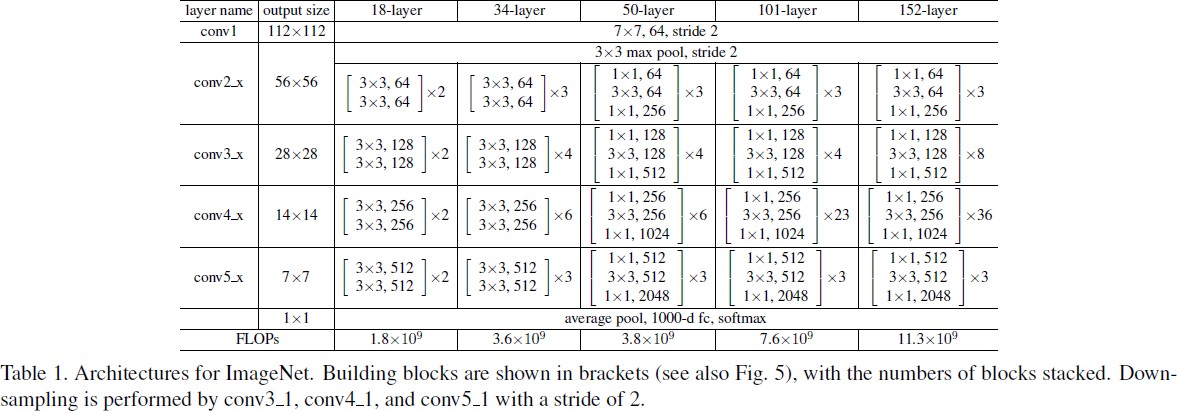
The bottleneck module uses the 1x1 convolutional layer to reduce or expand the feature map dimension so that the number of filters at the 3x3 convolutional layer is not affected by the input of the upper layer, and the output of the 3x3 convolutional layer does not affect the lower-layer module.



(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.



(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

# Addition operator self.add = ops.Add()

\_bn(out\_channel)]) # Batch Norm

# 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,

# conv3\_x residual block

self.layer2 = self.\_make\_layer(block,

# conv4\_x residual block

self.layer3 = self.\_make\_layer(block,

# conv5\_x residual block

self.layer4 = self.\_make\_layer(block,

# Mean operator

layer\_nums[0], in\_channel=in\_channels[0], out\_channel=out\_channels[0], stride=strides[0])

layer\_nums[1], in\_channel=in\_channels[1], out\_channel=out\_channels[1], stride=strides[1])

layer\_nums[2], in\_channel=in\_channels[2], out\_channel=out\_channels[2], stride=strides[2])

layer\_nums[3], in\_channel=in\_channels[3], out\_channel=out\_channels[3], stride=strides[3])

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\_c v\_top1acc76.97\_top5acc93.44.ckpt.

* + - 1. Load the pre-trained model.

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

* + - 1. 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.c kpt")

# 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

## 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

## 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.

## 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.](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

## Detailed Design and Implementation

* + 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/ENG/HCIA-](https://certification-data.obs.cn-north-4.myhuaweicloud.com/ENG/HCIA-AI/V3.5/chapter4/TextCNN.zip) [AI/V3.5/chapter4/TextCNN.zip](https://certification-data.obs.cn-north-4.myhuaweicloud.com/ENG/HCIA-AI/V3.5/chapter4/TextCNN.zip)

The data directory structure is as follows:

└─data

* + 1. Procedure

├─ rt-polarity.pos

└─ rt-polarity.neg

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 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.](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):

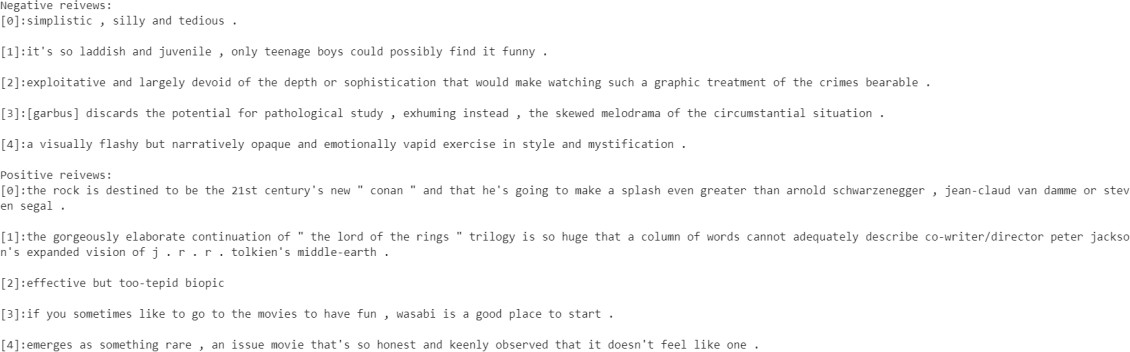
print("[{0}]:{1}".format(i,f.readline()))

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:



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: '''

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()

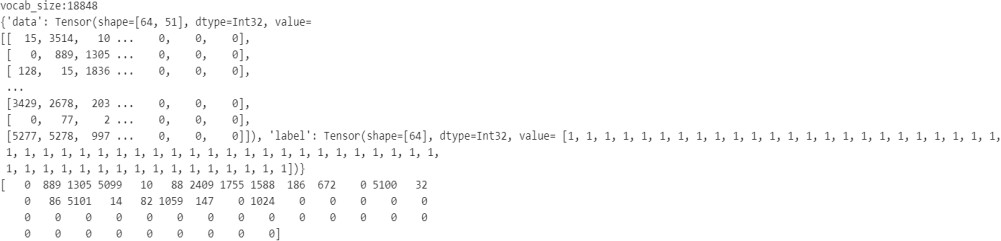
for i,data in enumerate(item): if i<1:

print(data) print(data['data'][1])

else:

break

Output the data processing result:



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))

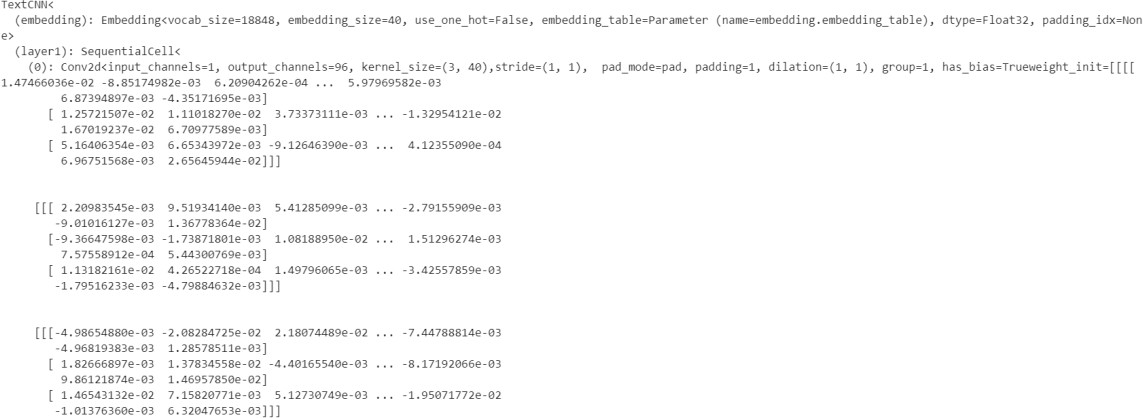
x = self.concat((x1, x2, x3)) x = self.drop(x)

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()

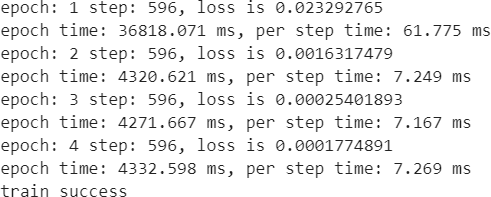
Step 6 Start training.

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:



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:



## Question

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

Huawei AI Certification Training

HCIA-AI

ModelArts Lab Guide

ISSUE: 3.5



Huawei Technologies Co., Ltd.

**1**

Copyright © Huawei Technologies Co., Ltd. 2023. All rights reserved.

###### No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Technologies Co., Ltd.

Trademarks and Permissions

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

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

Notice

###### The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

###### The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

### Huawei Technologies Co., Ltd.

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

###### Website: https://[e](http://e.huawei.com/).huawei.com

**Huawei Certification System**

###### Huawei Certification is an integral part of the company's "Platform + Ecosystem" strategy, and it supports the ICT infrastructure featuring "Cloud-Pipe-Device". It evolves to reflect the latest trends of ICT development. Huawei Certification consists of three categories: ICT Infrastructure Certification, Basic Software & Hardware Certification and Cloud Platform & Services Certification, making it the most extensive technical certification program in the industry.

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

###### Huawei Certification covers all ICT fields and adapts to the industry trend of ICT convergence.

###### With its leading talent development system and certification standards, it is committed to fostering new ICT talent in the digital era, and building a sound ICT talent ecosystem.

###### HCIA-AI V3.5 certification is intended for cultivating and conducting qualification of engineers who are capable of creatively designing and developing AI products and solutions using machine learning and deep learning algorithms.

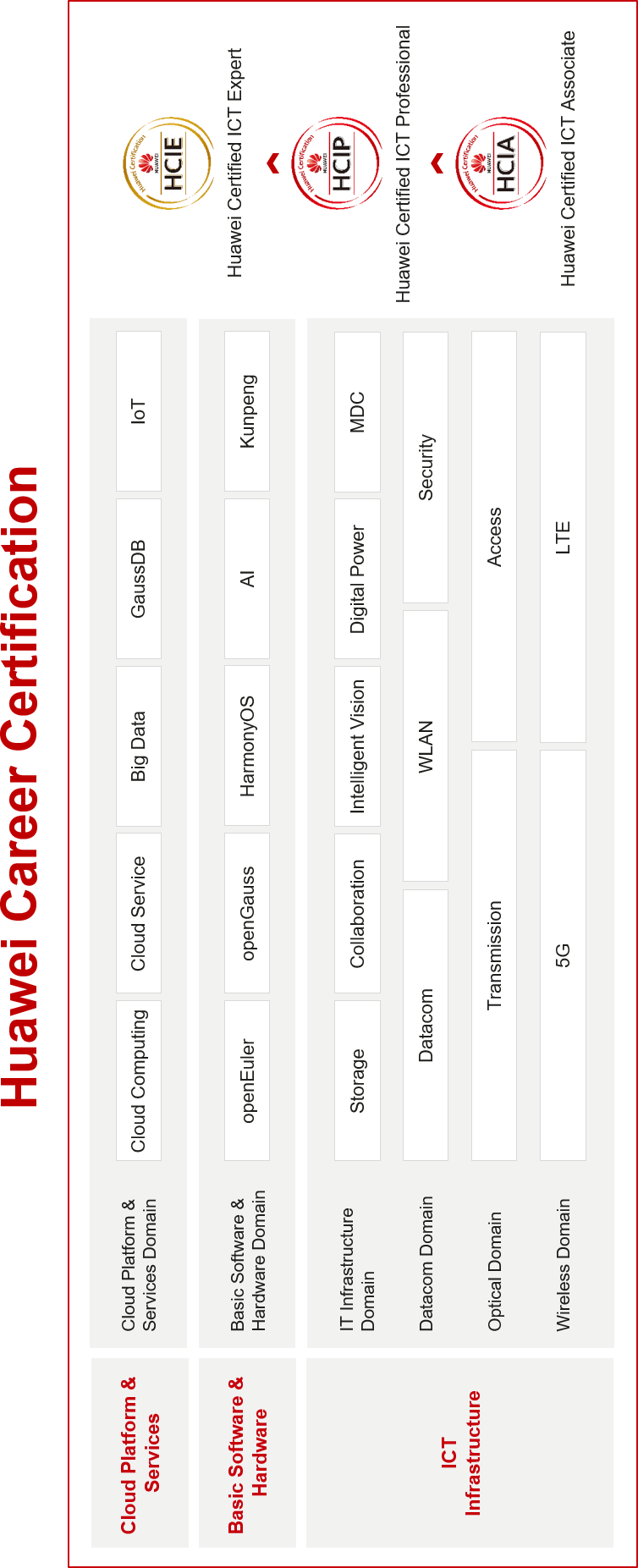
###### An HCIA-AI V3.5 certificate proves that you:

###### Have understood the development history of AI, Huawei Ascend AI system, and Huawei full-stack AI strategy in all scenarios;

###### Have mastered traditional machine learning and deep learning

###### Are able to use the MindSpore framework to build, train, and deploy neural networks;

###### Are competent in sales, marketing, product manager, project management, and technical support positions in the AI field.



**About This Document**

Overview

This document is intended for trainees who are preparing for the HCIA-AI certification examination or readers who want to achieve basic AI knowledge. After studying this document, you will be able to understand and use Huawei Cloud ModelArts ExeML.

Description

This document presents one exercise on Huawei Cloud ModelArts ExeML.

* + Exercise 1: How to use Huawei Cloud ModelArts ExeML

Knowledge Required

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

* + Basic computer operations.

Lab Environment

* + Accessible network

**Content**

##### [About This Document 3](#_bookmark59)

[Overview 3](#_bookmark60)

[Description 3](#_bookmark61)

[Knowledge Required 3](#_bookmark62)

[Lab Environment 3](#_bookmark63)

##### [ExeML 5](#_bookmark64)

* 1. [Introduction 5](#_bookmark65)
  2. [Objectives 5](#_bookmark66)
  3. [Lab Environment 5](#_bookmark67)
  4. [Flower Recognition Exercise 6](#_bookmark68)
     1. [Creating a Project 7](#_bookmark69)
     2. [Labeling Data 8](#_bookmark70)
     3. [Training a Model 9](#_bookmark71)
     4. [Deploying a Service and Performing Prediction 10](#_bookmark72)

**1 ExeML**

* 1. Introduction

ExeML, a service provided by ModelArts, automates model design, parameter tuning and training, and model compression and deployment using labeled data. ExeML is free of coding and does not require you to have experience in model development, meaning that it is beginner friendly. The exercise covered in this course will help you understand and use image classification, object detection, and predictive analysis.

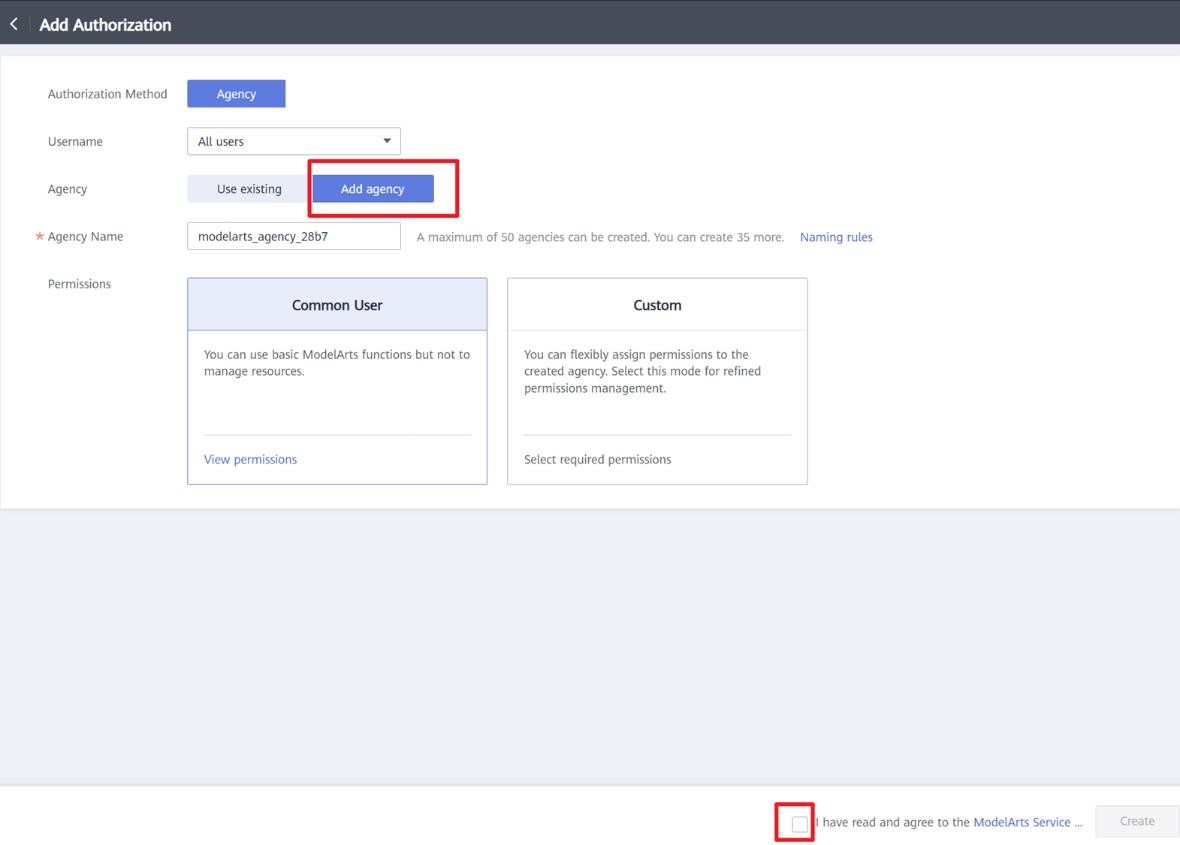
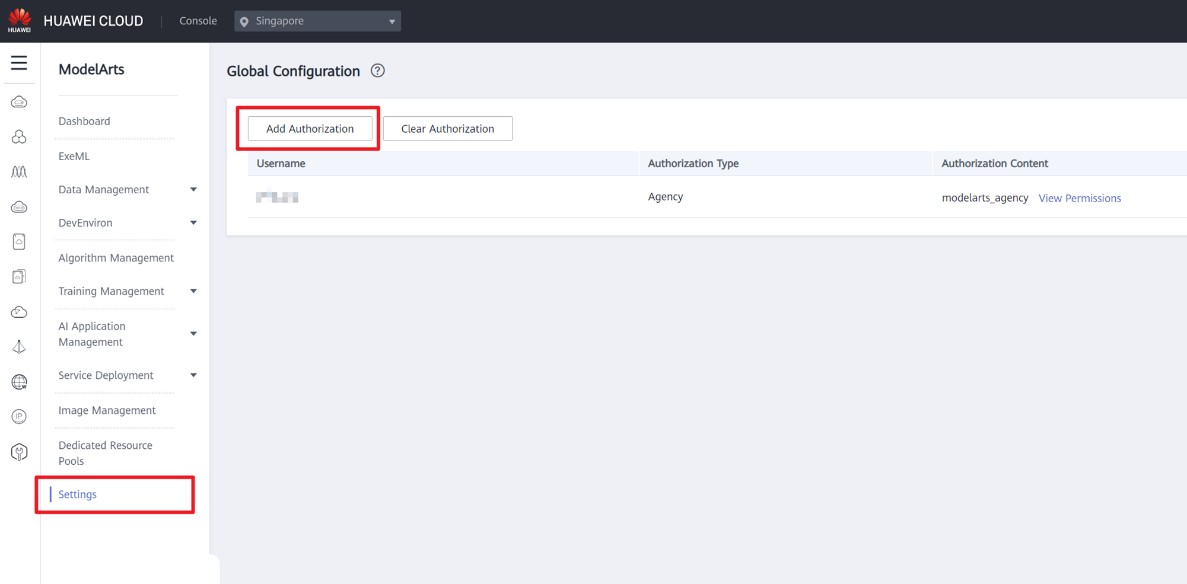
Image classification identifies a class of objects in images. An image classification model can predict the label of an image, and applies to scenarios in which image classes are distinguishable.

* 1. Objectives

The exercise in this course uses flower recognition as an example, to help you quickly create and learn about image classification models.

* 1. Lab Environment

If you are using ModelArts for the first time, authorize ModelArts to access Huawei Cloud Object Storage Service (OBS). Without doing so, you will not be able to create jobs on ModelArts. The procedure is as follows.



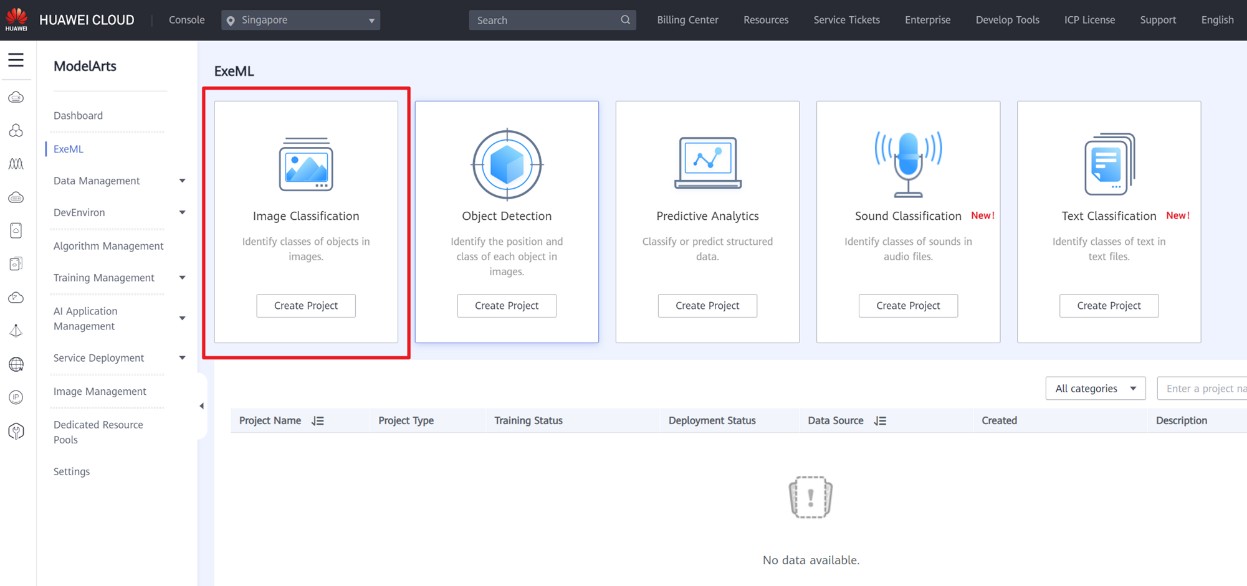
##### Figure 1-1 ModelArts console

* 1. Flower Recognition Exercise

The ModelArts ExeML page consists of two parts. The upper part lists the supported project types. You can click **Create Project** to create an ExeML project. The lower part displays the list of created ExeML projects. You can filter projects by project type in the upper right corner of the list, or enter



key words in the text box and click to search for projects that contain those key words.



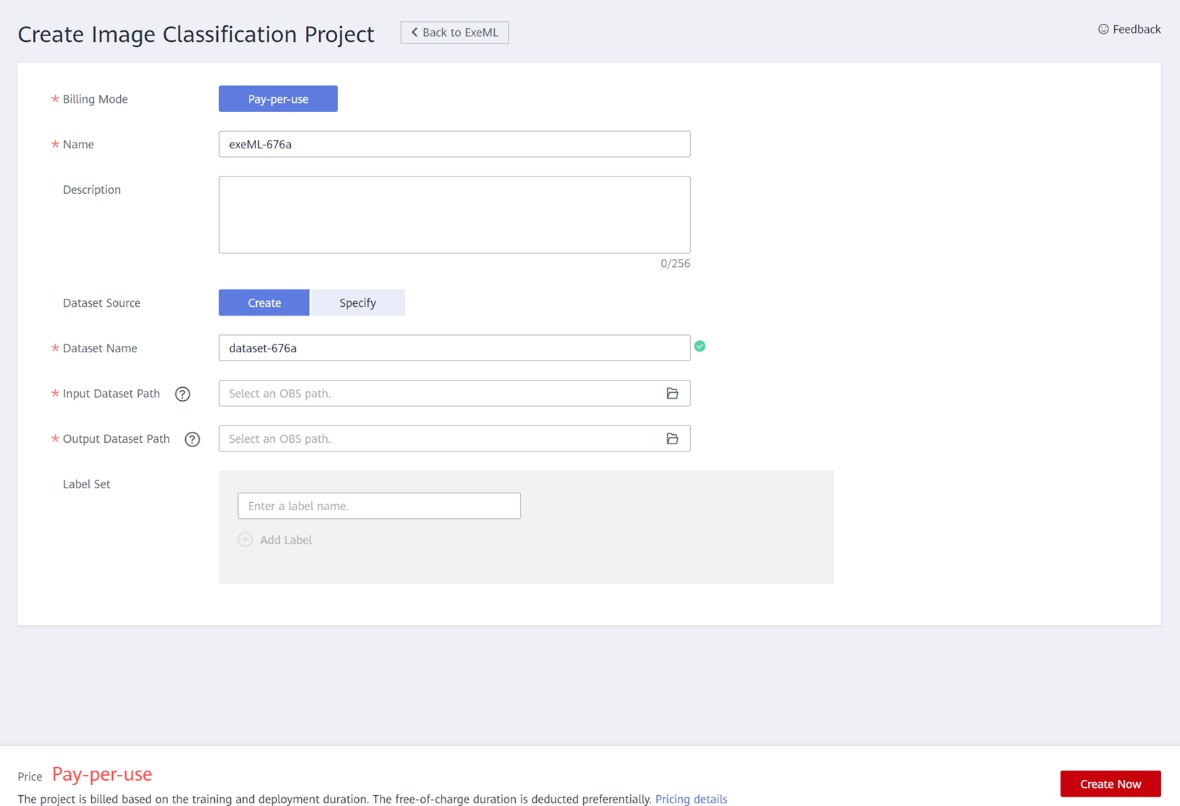
##### Figure 1-2 ExeML

The procedure of the exercise consists of four parts:

* + - Creating a project: To use ModelArts ExeML, first create an ExeML project.
    - Labeling data: Upload images and label them by class.
    - Training a model: After data labeling, train a model.
    - Deploying a service and performing prediction: Deploy the model as a service and perform real- time prediction.
    1. Creating a Project

Step 1 Set parameters.

On the **ExeML** page, click **Create Project** in **Image Classification**. The **Create Image Classification Project** page is displayed.



Parameters:

##### Figure 1-3 Creating a project

**Billing Mode**: **Pay-per-use** by default

**Name**: Enter a project name.

**Dataset Source**: Select the OBS path where the dataset is stored.

You must create an empty OBS folder and select this folder as the training data path. (Click the bucket name, and then click **Create Folder**.) Alternatively, import the data to OBS in advance. In this example, click the following data link and decompress the downloaded file package, and upload the data in the **train** folder to the **OBS/ExeML/flower/train** directory.

For details about how to upload data, see https://support.huaweicloud.com/intl/en- us/modelarts\_faq/modelarts\_05\_0013.html.

Data link:

[https://certification-data.obs.cn-north-4.myhuaweicloud.com/ENG/HCIA-](https://certification-data.obs.cn-north-4.myhuaweicloud.com/ENG/HCIA-AI/V3.5/chapter5/ExeML.zip) [AI/V3.5/chapter5/ExeML.zip](https://certification-data.obs.cn-north-4.myhuaweicloud.com/ENG/HCIA-AI/V3.5/chapter5/ExeML.zip)

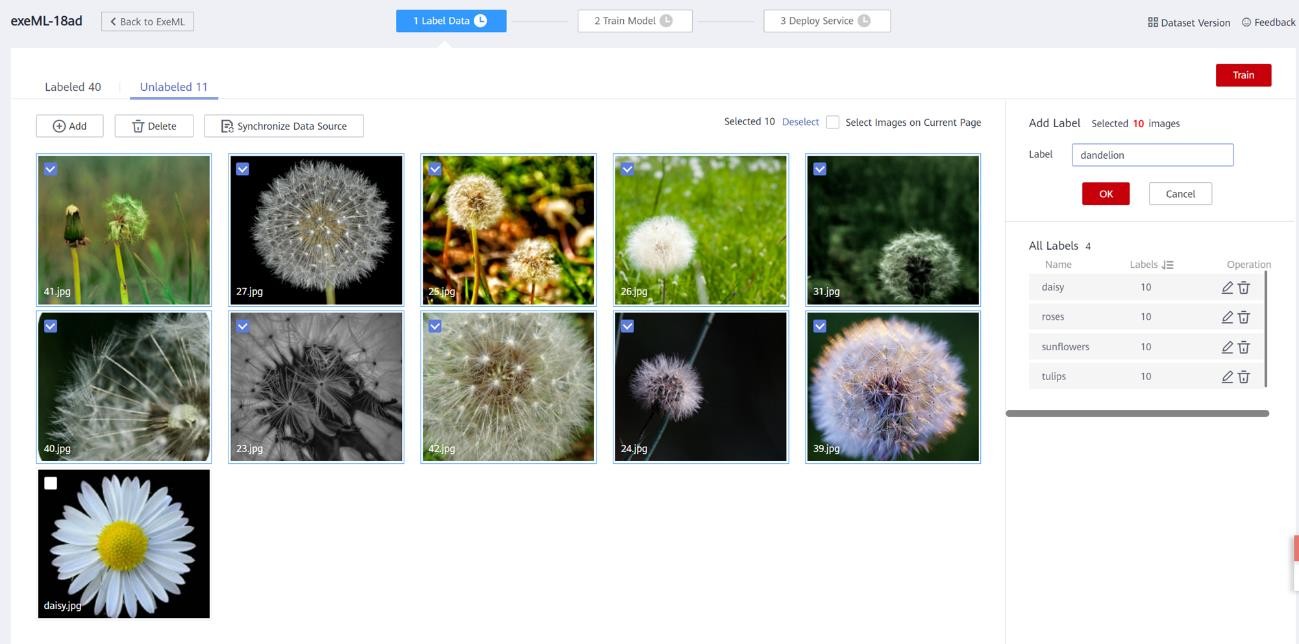
Step 2 Click **Create Now**.

Click **Create Now**. The ExeML project is then created.

* + 1. Labeling Data

Step 1 Upload images.

After an ExeML project is created, the data labeling page is automatically displayed. Images uploaded to the OBS directory configured during project creation are automatically loaded. Labeled images are displayed in the **Labeled** tab.



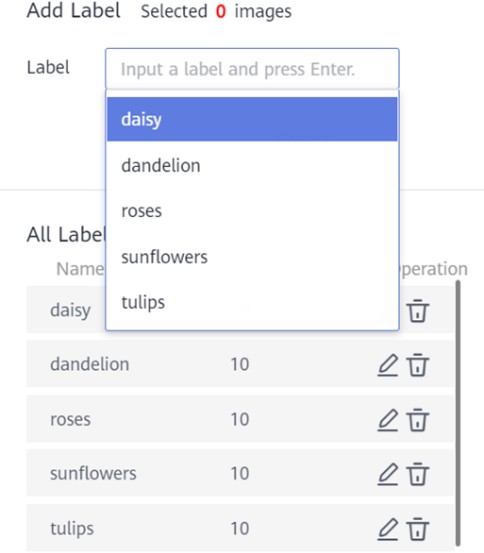
Notes:

##### Figure 1-4 Data labeling for image classification

* + - * The images to be trained must be classified into at least two classes, and each class must contain at least five images. That is, at least two labels are available and the number of images for each label is not fewer than five. In this example, the dataset contains five labels: tulips, daisy, sunflowers, roses, and dandelion.
      * You can add multiple labels to an image.

Step 2 Label the images.

Click **Unlabeled** and select the unlabeled images you want to label, or select **Select Images on Current Page** in the upper right corner to select all images on the page. Then, input a label or select an existing label from the drop-down list, and then press **Enter**. Then, click **OK**.



##### Figure 1-5 Image labeling for image classification

Step 3 Modify labels.

Click the **Labeled** tab, select an image, enter a new label name in the **Label** text box on the right, and click **OK**.

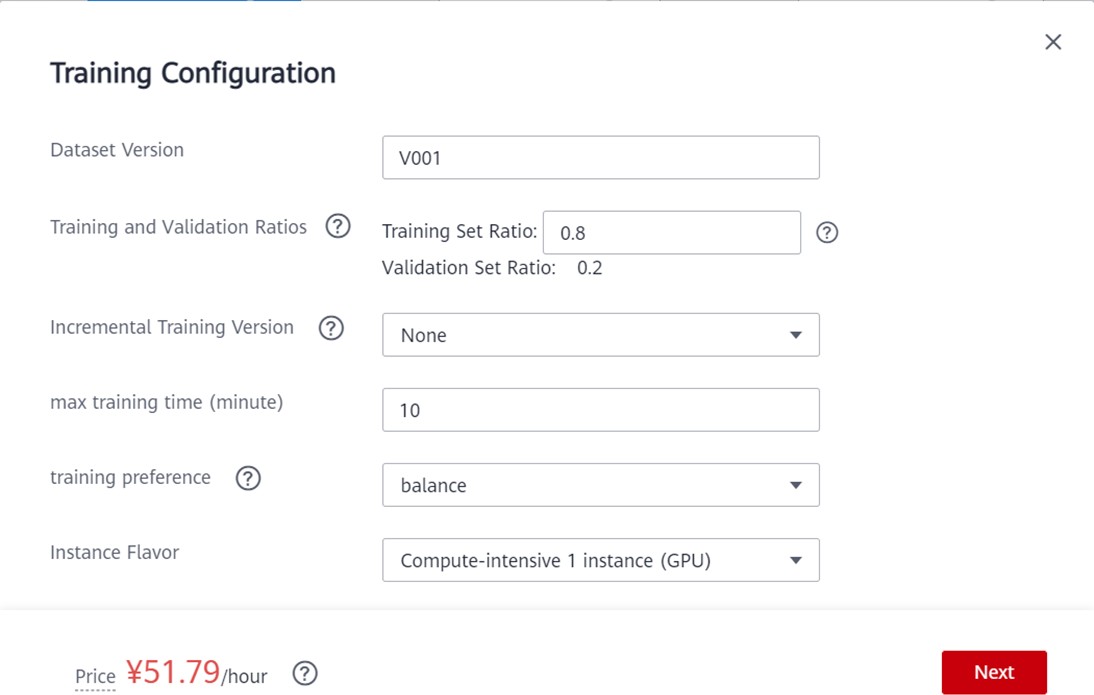
You can select multiple images at a time and modify them in batches.

* + 1. Training a Model

After labeling the images, train a model. Before training, set parameters. Images to be trained must be classified into at least two classes, and each class must contain at least five images. Therefore, before training, ensure that the labeled images meet the requirements. Otherwise, the **Train** button remains unavailable.

Step 1 Set parameters.

Use default settings or configure them.



Parameters:

##### Figure 1-6 Training Configuration

**max training time (minute)**: If the training is not completed within the maximum training duration, the training is forcibly stopped. Enter a larger value to avoid this situation.

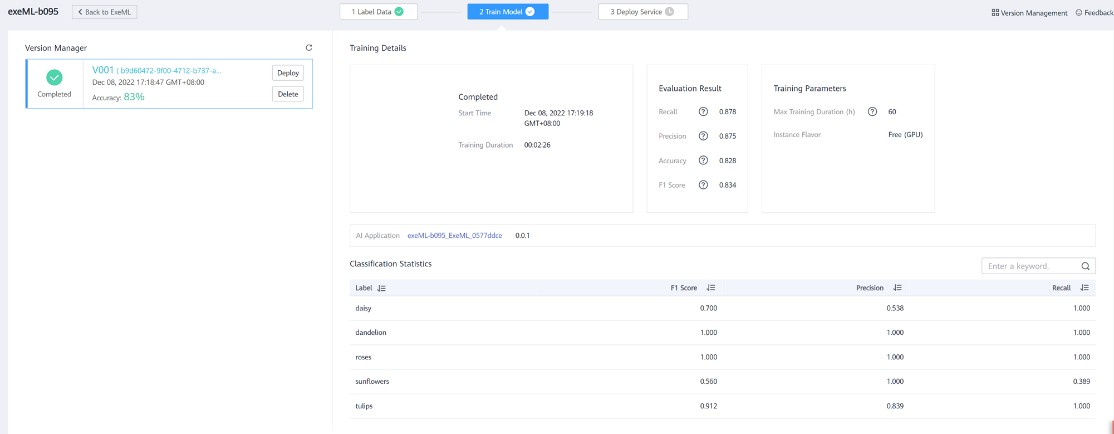
Step 2 Train a model.

After setting, click **Train**. After the training, you can view the training result on the **Train Model** tab page.

* + 1. Deploying a Service and Performing Prediction

Step 1 Deploy a service.

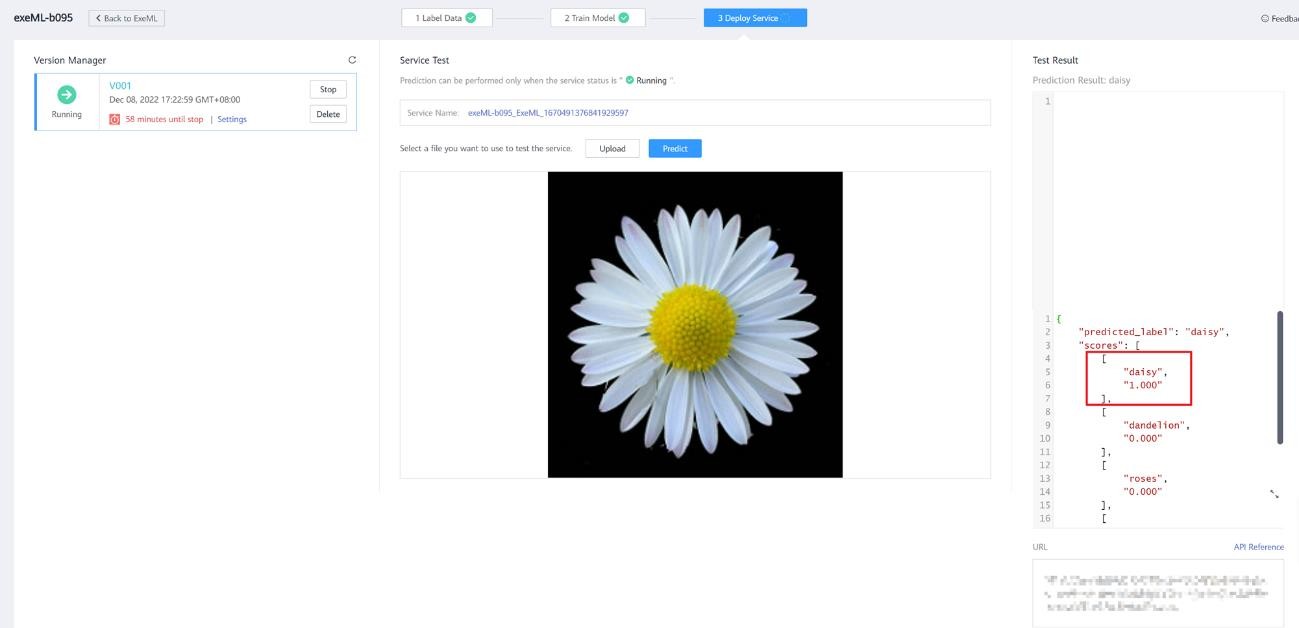
After model training, deploy a model version with satisfactory accuracy and in the **Completed** status as a service. To do so, click **Deploy** in **Version Manager** of the **Train Model** tab page. After the deployment, choose **Service Deployment** > **Real-Time Services** to view the deployed service.



##### Figure 1-7 Deploy

Step 2 Test the service.

After the model is deployed, test the service using an image. The test data is stored in **OBS/flower/test/daisy.jpg**. On the **Deploy Service** tab, click **Upload** and upload an image. Then, click **Predict**. The test result is displayed in the right pane. There are five labels for data labeling: tulip, daisy, sunflower, rose, and dandelion. In the prediction, "daisy" gets the highest score, so the classification result is "daisy".



##### Figure 1-8 Service test