华为昇腾-昇思MindSpore《AI计算系统》精品课程

HCIA-AI计算系统

精品课程

实验指导手册

版本:1.0



华为技术有限公司

|  |
| --- |
| 版权所有 © 华为技术有限公司 2021。 保留一切权利。  非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。  商标声明  C:\Users\jwx341670\Desktop\华为标志 Huawei Logo 2018\竖版标志Vertical Version\PNG\HW_POS_RBG_Vertical-150ppi.png 和其他华为商标均为华为技术有限公司的商标。  本文档提及的其他所有商标或注册商标，由各自的所有人拥有。  注意  您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。  由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。 |

|  |  |
| --- | --- |
| 华为技术有限公司 | |
| 地址： | 深圳市龙岗区坂田华为总部办公楼 邮编：518129 |
| 网址： | http://[e](http://e.huawei.com/).huawei.com |

目录

[8 全流程垃圾分类应用 4](#_Toc1473852800)

[8.1实验介绍 4](#_Toc1967625102)

[8.1.1 数据集的介绍 4](#_Toc774409161)

[8.1.2 模型知识点的介绍 5](#_Toc1743868107)

[8.2 实验环境要求 8](#_Toc334460093)

[8.3 实验详细设计与实现 8](#_Toc1306078852)

[8.3.1 实验步骤 8](#_Toc1836909577)

[8.4 实验总结 40](#_Toc714351367)

[8.5 实验任务与参考解答任务 40](#_Toc1669838439)

[8.5.1 实验任务 40](#_Toc1658345277)

# 全流程垃圾分类应用

## 8.1实验介绍

本实验实现的是基于SENet实现垃圾分类从训练到部署的全流程应用案例。整体流程如下：

* 处理需要的数据集，这里使用了垃圾分类图像数据集。
* 搭建SENet50模型结构。
* 定义损失函数和优化器。
* 加载数据集并进行训练，训练完成后，利用测试集进行评估。
* 将模型导出，并用ACL实现部署开发，搭建实时垃圾分类应用。

### 数据集的介绍

我们示例中用到的垃圾分类数据集，总共包括40种垃圾类型：分别是0（一次性快餐盒，张），1（污损塑料，张），2（烟蒂，张），3（牙签，张），4（破碎花盆及碟碗，张），5（竹筷，张），6（剩饭剩菜，张），7（大骨头，张），8（水果果皮，张），9（水果果肉，张），10（茶叶渣，张），11（菜叶菜根，张），12（蛋壳，张），13（鱼骨，张），14（充电宝，张），15（包，张），16（化妆品瓶，张），17（塑料玩具，张），18（塑料碗盆，张），19（塑料衣架，张），20（快递纸袋，张），21（插头电线，张），22（旧衣服，张），23（易拉罐，张），24（枕头，张），25（毛绒玩具，张），26（洗发水瓶，张），27（玻璃杯，张），28（皮鞋，张），29（砧板，张），30（纸板箱，张），31（调料瓶，张），32（酒瓶，张），33（金属食品罐，张），34（锅，张），35（食用油桶，张），36（饮料瓶，张），37（干电池，张），38（软膏，张），39（过期药物，张）保存在40个文件夹当中，总共14802张，大小大概在571M左右。为了在模型部署上线之后进行测试，数据集在这里分成了data/train和data/test两部分。

目录结构如下：

data

├── train

├ ├── 0 //分类编号顺次向下，一共40个分类

├ ├── 1

├ ├── 2

├ ├── ...

├ ├── 39

├── test

├ ├── 0

├ ├── 1

├ ├── 2

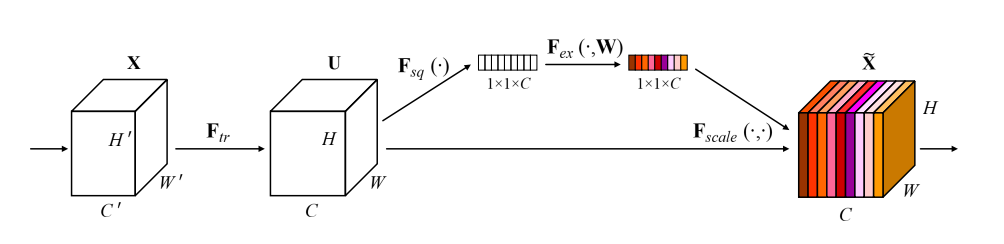
├ ├── ...

├ ├── 39

### 模型知识点的介绍

SENet（Squeeze-and-Excitation Networks ）是 ImageNet 2017 竞赛 Image Classification 任务的冠军。在ImageNet数据集上将top-5 错误率降低到2.251%，原先的最好成绩是2.991%。SENet网络创新点在于关注通道之间的关系，希望模型可以自动学习到不同通道特征的重要程度，也就是如下图所示的Squeeze-and-Excitation Module。

SE模块首先对卷积得到的特征图进行Squeeze操作，得到channel的全局特征，然后对全局特征进行Excitation操作，学习各个channel之间的关系，也得到不同的channel的权重，最后乘以原来的特征图得到加权特征。本质上，SE模块是channel维度上做attention或者gating操作，使模型可以更加关注更具有表征能力的通道特征，抑制那些不重要的通道特征。



**1.Squeeze-and-Excitation Module**

SE模块主要包括Squeeze和Excitation两个操作，可以适用任何映射

，以卷积为例，卷积核为，其中代表第c个卷积核，那么输出，其中\*代表卷积操作，而代表一个s通道的二维卷积核，其输入一个通道上的空间特征，它学习特征空间关系，但是由于对各个通道的卷积结果做了求和，所以通道特征关系与卷积核学习到的空间关系混合在一起。而SE模块就是为了抽离这种混杂，使得模型直接学习到通道特征关系。

**2.Squeeze操作**

由于卷积只是在一个局部空间内进行操作，$U$很难获得足够的信息来提取通道之间的关系，对于网络中前面的层这更严重，因为感受野比较小。为了，SENet提出Squeeze操作，将一个通道上整个空间特征编码为一个全局特征，采用global average pooling来实现：

**3.Excitation操作**

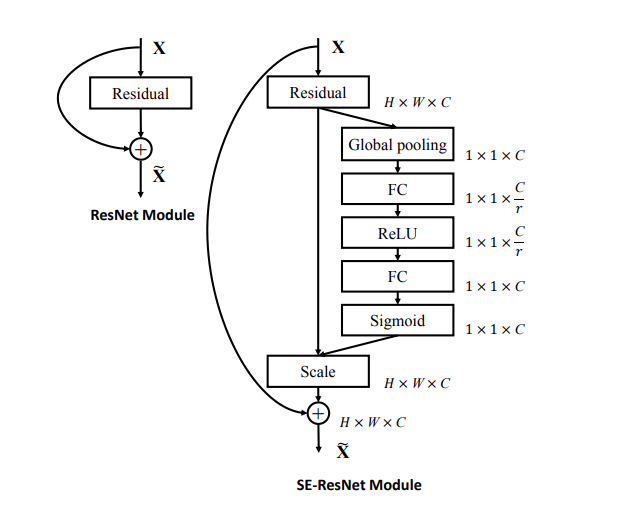
Sequeeze操作得到了全局描述特征，接下来需要另外一种运算来抓取通道之间的关系。这个操作需要满足两个准则：首先要灵活，它要可以学习到各个channel之间的非线性关系；第二点是学习的关系不是互斥的，因为这里允许多channel特征，而不是one-hot形式。基于此，这里采用sigmoid形式的gating机制：

其中。为了降低模型复杂度以及提升泛化能力，这里采用包含两个全连接层的bottleneck结构，其中第一个FC层起到降维的作用，降维系数为r是个超参数，然后采用ReLU激活。最后的FC层恢复原始的维度。最后将学习到的各个通过的激活值乘以U上的原始特征：

其实整个操作可以看成学习到了各个通道的权重系数，从而使得模型对各个通道的特征更有辨别能力，这应该也算一种注意力机制。

**4.SE模块在ResNet上的应用**

SE模块的灵活性在于它可以直接应用现有的网络结构中。对于ResNet，SE模块嵌入到残差结构中的残差学习分支中。具体如下图所示：



**5.SEResNet-50**

此实验使用SEResNet-50网络实现垃圾分类应用，具体网络结构如下所示：

|  |  |  |
| --- | --- | --- |
| **Input Size** | **SE-ResNet-50** | **Output Size** |
| 3×224×224 | [conv, 7 × 7, 64, stride 2], [max pool, 3 × 3, stride 2] | 64x112×112 |
| 64×112×112 | [[conv, 1×1, 64], [conv, 3×3, 64], [conv, 1×1, 256] [fc, 16, 256] × 3 | 256×56×56 |
| 256×56×56 | [[conv, 1×1, 128], [conv, 3×3, 128], [conv, 1×1, 512] [fc, 32, 512] × 4 | 512×28×28 |
| 512×28×28 | [[conv, 1×1, 256], [conv, 3×3, 256], [conv, 1×1, 1024] [fc, 64, 1024] × 6 | 1024×14×14 |
| 1024×14×14 | [[conv, 1×1, 512], [conv, 3×3, 512], [conv, 1×1, 2048] [fc, 128, 2048] × 3 | 20487×7 |
| 2048×7×7 | global average pool | 2048 |
| 2048 | 40-d fc,softmax | 40 |

## 

## 实验环境要求

在动手进行实践之前，确保你已经正确安装了MindSpore。如果没有，可以通过MindSpore官网安装页面：<https://www.mindspore.cn/install/>，将MindSpore安装在你的电脑当中。

同时希望你拥有Python编码基础和概率、矩阵等基础数学知识。

推荐环境：

版本：MindSpore 1.7

编程语言：Python 3.7

## 实验详细设计与实现

### 实验步骤

导入Python库&模块并配置运行信息

在使用前，导入需要的Python库。详细的MindSpore的模块说明，可以在MindSpore API页面中搜索查询。可以通过context.set\_context来配置运行需要的信息，譬如运行模式、后端信息、硬件等信息。

导入context模块，配置运行需要的信息。

代码：

# os模块主要用于处理文件和目录

import os

# 科学计算库

import numpy as np

# 随机库

import random

# 进度条

from tqdm import tqdm

# 处理图片模块

import cv2

# 绘图库

import matplotlib.pyplot as plt

# 读取图片模块

from PIL import Image, ImageFile

# MindSpore库

import mindspore

# 数据集处理模块

import mindspore.dataset as ds

# 图像增强模块

from mindspore.dataset.vision import c\_transforms as vision

# 环境设置模块

from mindspore import context

# 神经网络模块

import mindspore.nn as nn

# 模型编译

from mindspore.train import Model

# 动量优化器

from mindspore.nn.optim.momentum import Momentum

# 模型保存设置

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

# 张量

from mindspore import Tensor

# 模型导出

from mindspore.train.serialization import export

# 损失值平滑处理

from mindspore.train.loss\_scale\_manager import FixedLossScaleManager

# 模型加载

from mindspore.train.serialization import load\_checkpoint, load\_param\_into\_net

# 常见算子操作

import mindspore.ops as ops

定义参数变量

garbage\_config.yaml中存放的是模型训练和测试中所需要的各种参数配置。

代码：

enable\_modelarts: False # 是否适配modelArts

data\_url: "" # 数据url地址

train\_url: "" # 训练url地址

checkpoint\_url: "" # 权重url地址

data\_path: "/cache/data" # 数据路径

output\_path: "/cache/train" # 输出结果保存路径

load\_path: "/cache/checkpoint\_path" # 加载权重路径

device\_target: "Ascend" # 设备类型

need\_modelarts\_dataset\_unzip: False # 是否需要数据集解压

# 参数配置项

num\_classes: 40 # 类别数

lr: 0.001 # 学习率

lr\_init: 0.0 # 初始学习率

lr\_end: 0.001 # 中止学习率

lr\_max: 0.3 # 学习率最大值

lr\_epochs: '30,60,90,120' # 学习率调整轮次

lr\_scheduler: "step" # 学习率条件方式

warmup\_epochs: 5 # 热启动轮次

batch\_size: 32 # 训练批次大小

max\_epoch: 100 # 训练多少轮次

momentum: 0.9 # 学习率动量参数

weight\_decay: 0 # 学习率权重衰减系数

loss\_scale: 1.0 # 损失缩放值

label\_smooth: 0 # 标签平滑

label\_smooth\_factor: 0 # 标签平滑因子

buffer\_size: 10 # 缓存大小

image\_size: '288,288' # 图片尺寸大小

pad\_mode: 'same' # 填充方式

padding: 0 # 填充值

reduction: 16 # 最小尺寸倍数

has\_bias: False # 是否含偏置

batch\_norm: True # 是否进行标准化

keep\_checkpoint\_max: 10 # 保存权重数量最大数

initialize\_mode: "XavierUniform" # 权重初始化方式

has\_dropout: False # 是否包含dropout

# 训练参数

dataset: "garbage\_photos" # 数据集名称

data\_dir: "" # 数据集目录

pre\_trained: "" # 预训练权重路径

lr\_gamma: 0.1 # 学习率调节参数

eta\_min: 0.0 # 学习率调节参数

T\_max: 90 # 学习率调节参数

log\_interval: 100 # log打印频次

ckpt\_path: "./outputs\_garbages/" # 权重保存目录

ckpt\_interval: 5 # 权重保存轮次频率

is\_save\_on\_master: 1 # 是否只在主节点上保存权重

is\_distributed: 0 # 是否分布式训练

# 验证参数

per\_batch\_size: 32 # 验证数据批次

graph\_ckpt: 1 # 使用当前图的权重

log\_path: "./outputs\_garbages/" # 日志保存目录

# 后处理参数

result\_dir: "" # 结果保存目录

label\_dir: "" # 标签保存目录

dataset\_name: "garbage" # 数据集名称

# 预处理参数

result\_path: "./preprocess\_Result/" # 预处理结果保存目录

# 导出参数

ckpt\_file: "" # 使用权重路径

file\_name: "seresnet" # 导出模型名称

file\_format: "MINDIR" # 导出模型格式

数据分析

数据的好坏决定了模型的上下限，对于基本的图片分类任务，首先需要对数据集做一个数据分析，查看数据分布状况以及数据质量状态，并针对不同的数据集状态选择不同的处理方式，从而达到提升模型性能的效果。

这里统计训练集下每个类别下的图片数量关系以及训练集下图片大小的分布状态，代码如下：

# 绘制各类别数量函数

# 输入参数为：数据集目录，绘制类型

def draw\_data\_count(image\_dir, dtype="bar"):

# 获取当前图片目录下子目录

sub\_folders = next(os.walk(image\_dir))[1]

# 根据类别进行排序

sub\_folders.sort(key=lambda x:int(x))

# 创建子目录下的信息列表

sub\_folders\_info = []

# 所有图片数

all\_image\_nums = 0

# 遍历子目录

for sub\_folder in tqdm(sub\_folders):

# 获得子目录下所有图片

image\_ids = next(os.walk(os.path.join(image\_dir, sub\_folder)))[2]

# 计算当前图片数

image\_counts = len(image\_ids)

# 将子目录名和图片数添加至信息列表

sub\_folders\_info.append((sub\_folder, image\_counts))

# 统计所有的图片数

all\_image\_nums += image\_counts

# 获取所有的类别

x = [i[0] for i in sub\_folders\_info]

# 获取所有类别的数目

y = [i[1] for i in sub\_folders\_info]

# 创建画布

plt.figure(figsize=(16, 8))

if dtype == "bar": # 绘制条形图

plt.bar(x, y, width=0.8)

for i,j in enumerate(y):

plt.text(i, j+4, j,ha='center',fontsize=8)

elif dtype == "pie": # 绘制饼状图

plt.pie(y, labels=x)

plt.axis('equal')

else:

raise ValueError

# 绘制各类别图片大小函数

# 输入参数为：数据集目录

def draw\_data\_size(image\_dir):

# 获取当前图片目录下子目录

sub\_folders = next(os.walk(image\_dir))[1]

# 根据类别进行排序

sub\_folders.sort(key=lambda x:int(x))

# 创建数据大小列表

data\_size\_list = []

# 遍历子目录

for sub\_folder in tqdm(sub\_folders):

# 获取当前类别下的所有图片

image\_ids = next(os.walk(os.path.join(image\_dir, sub\_folder)))[2]

# 这里随机取样十分之一样本统计尺寸大小

random.shuffle(image\_ids)

image\_shapes = [cv2.imread(os.path.join(image\_dir, sub\_folder, image\_ids[i])).shape[:2] for i in range(len(image\_ids)//10)]

data\_size\_list.extend(image\_shapes)

# 获取所有采样图片的高和宽

h = [i[0] for i in data\_size\_list]

w = [i[1] for i in data\_size\_list]

# 创建画布

plt.figure(figsize=(16, 8))

# 绘制散点图

plt.scatter(h, w)

# 统计训练集每个类别的数量

draw\_data\_count("../data/train/", dtype="bar")

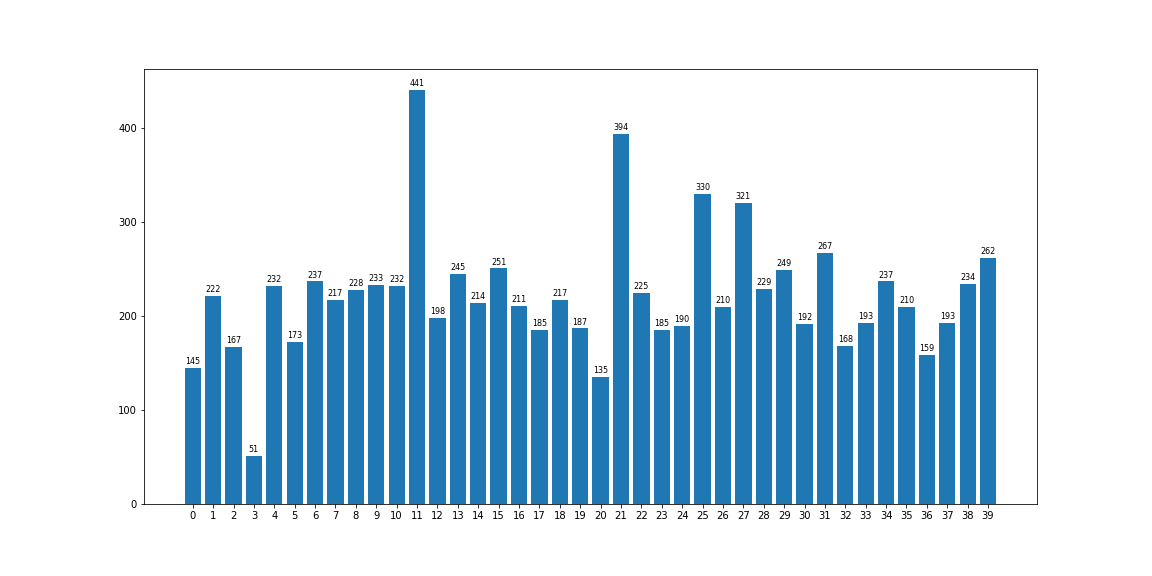
plt.show()

# 统计训练集尺寸大小分布

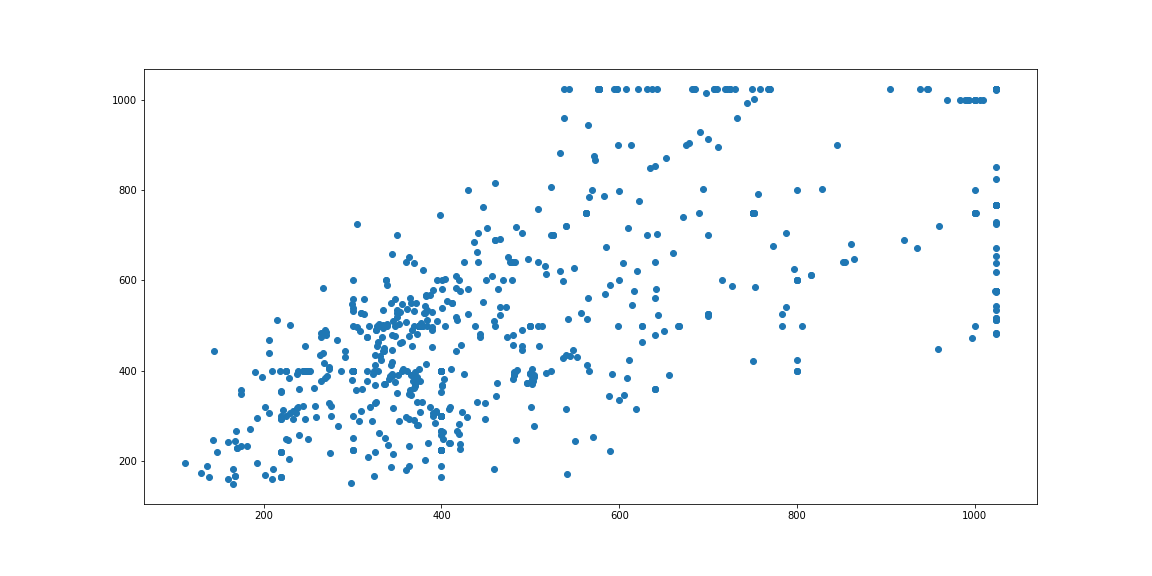
draw\_data\_size("../data/train/")

plt.show()

训练样本统计结果如下：40个类别的训练图片数量如下所示，可以看出40个类别中各个数据量比较均匀，但类别为"3"的数据量比较少，在训练过程中可能由于数据量不足导致对类别3的识别效果较差，若结果符合预期，则可以采用增加类别3样本的手段，如数据扩充，数据增强等，提升模型对类别3样本的识别效果。



训练样本尺寸分布如下：可以看出图片的尺寸大小不一，需要对图片做缩放操作后再送入神经网络进行训练。图片尺寸是决定网络模型的一个重要因素，图片尺寸大则具有更多的信息量，能够融合更多信息；图片尺寸小则具有更快的训练速度。



数据的读取和处理

数据集对于训练非常重要，好的数据集可以有效提高训练精度和效率。在加载数据集前，我们通常会对数据集进行一些处理。

定义数据集及数据操作

我们定义一个函数create\_dataset来创建数据集。在这个函数中，我们定义好需要进行的数据增强和处理操作：

* 读取数据集。
* 定义进行数据增强和处理所需要的一些参数。
* 根据参数，生成对应的数据增强操作。

代码：

# 创建数据集

# 输入参数为：数据集目录，图片大小，批次大小，节点ID，节点数，是否为训练集，是否为验证集

def resnet\_create\_dataset(data\_home, image\_size, batch\_size, rank\_id=0, rank\_size=1, training=True,valid=False):

# 指明对应数据目录

data\_dir = data\_home

data\_dir = os.path.join(data\_home, "train")

if not training:

data\_dir = os.path.join(data\_home, "test")

if valid:

data\_dir = os.path.join(data\_home, "val")

# 构建标签字典

label\_list={}

for i in range(40):

label\_list[str(i)]=i

# 从对应目录中读取图像的源数据集。

data\_set = de.ImageFolderDataset(data\_dir,

class\_indexing=label\_list,

shuffle=False, num\_shards=rank\_size,

shard\_id=rank\_id)

# 图像调整大小算子

transform\_img = vision.RandomCropDecodeResize([288,288],

scale=(0.08, 1.0),

ratio=(0.75, 1.333))

# 图片通道转化算子

changeswap\_op = vision.HWC2CHW()

# 类别转换算子

type\_cast\_op = C.TypeCast(mstype.float32)

# 图像调整大小算子

resize\_op = vision.Resize(image\_size) # interpolation default BILINEAR

# 随机水平翻转算子

random\_horizontal\_op = vision.RandomHorizontalFlip()

# 标准化算子

normalize\_op = vision.Normalize((0.4465, 0.4822, 0.4914), (0.2010, 0.1994, 0.2023))

if training:# 只对训练集做的预处理操作

data\_set = data\_set.map(input\_columns="image", operations=transform\_img)

data\_set = data\_set.map(input\_columns="image", operations=random\_horizontal\_op)

else: # 只对测试集做的预处理操作

decode\_op = vision.Decode()

data\_set = data\_set.map(input\_columns="image", operations=decode\_op)

data\_set = data\_set.map(input\_columns="image", operations=resize\_op)

# 对全部数据集做的预处理操作

data\_set = data\_set.map(input\_columns="image", operations=normalize\_op)

data\_set = data\_set.map(input\_columns="image", operations=type\_cast\_op)

data\_set = data\_set.map(input\_columns="image", operations=changeswap\_op)

# 随机打乱

data\_set = data\_set.shuffle(buffer\_size=data\_set.get\_dataset\_size())

# 设置图片加载批次

data\_set = data\_set.batch(batch\_size=batch\_size, drop\_remainder=True)

return data\_set

模型构建训练

代码：

# 定义卷积参数初始化

def conv\_variance\_scaling\_initializer(in\_channel, out\_channel, kernel\_size):

fan\_in = in\_channel \* kernel\_size \* kernel\_size

scale = 1.0

scale /= max(1., fan\_in)

stddev = (scale \*\* 0.5) / .87962566103423978

mu, sigma = 0, stddev

weight = truncnorm(-2, 2, loc=mu, scale=sigma).rvs(out\_channel \* in\_channel \* kernel\_size \* kernel\_size)

weight = np.reshape(weight, (out\_channel, in\_channel, kernel\_size, kernel\_size))

return Tensor(weight, dtype=mstype.float32)

# 定义权重初始化函数

def \_weight\_variable(shape, factor=0.01):

init\_value = np.random.randn(\*shape).astype(np.float32) \* factor

return Tensor(init\_value)

# 计算增益

def calculate\_gain(nonlinearity, param=None):

"""calculate\_gain"""

linear\_fns = ['linear', 'conv1d', 'conv2d', 'conv3d', 'conv\_transpose1d', 'conv\_transpose2d', 'conv\_transpose3d']

res = 0

if nonlinearity in linear\_fns or nonlinearity == 'sigmoid':

res = 1

elif nonlinearity == 'tanh':

res = 5.0 / 3

elif nonlinearity == 'relu':

res = math.sqrt(2.0)

elif nonlinearity == 'leaky\_relu':

if param is None:

neg\_slope = 0.01

elif not isinstance(param, bool) and isinstance(param, int) or isinstance(param, float):

neg\_slope = param

else:

raise ValueError("neg\_slope {} not a valid number".format(param))

res = math.sqrt(2.0 / (1 + neg\_slope \*\* 2))

else:

raise ValueError("Unsupported nonlinearity {}".format(nonlinearity))

return res

# 计算张量输入输出

def \_calculate\_fan\_in\_and\_fan\_out(tensor):

"""\_calculate\_fan\_in\_and\_fan\_out"""

dimensions = len(tensor)

if dimensions < 2:

raise ValueError("Fan in and fan out can not be computed for tensor with fewer than 2 dimensions")

if dimensions == 2: # Linear

fan\_in = tensor[1]

fan\_out = tensor[0]

else:

num\_input\_fmaps = tensor[1]

num\_output\_fmaps = tensor[0]

receptive\_field\_size = 1

if dimensions > 2:

receptive\_field\_size = tensor[2] \* tensor[3]

fan\_in = num\_input\_fmaps \* receptive\_field\_size

fan\_out = num\_output\_fmaps \* receptive\_field\_size

return fan\_in, fan\_out

def \_calculate\_correct\_fan(tensor, mode):

mode = mode.lower()

valid\_modes = ['fan\_in', 'fan\_out']

if mode not in valid\_modes:

raise ValueError("Unsupported mode {}, please use one of {}".format(mode, valid\_modes))

fan\_in, fan\_out = \_calculate\_fan\_in\_and\_fan\_out(tensor)

return fan\_in if mode == 'fan\_in' else fan\_out

def kaiming\_normal(inputs\_shape, a=0, mode='fan\_in', nonlinearity='leaky\_relu'):

fan = \_calculate\_correct\_fan(inputs\_shape, mode)

gain = calculate\_gain(nonlinearity, a)

std = gain / math.sqrt(fan)

return np.random.normal(0, std, size=inputs\_shape).astype(np.float32)

def kaiming\_uniform(inputs\_shape, a=0., mode='fan\_in', nonlinearity='leaky\_relu'):

fan = \_calculate\_correct\_fan(inputs\_shape, mode)

gain = calculate\_gain(nonlinearity, a)

std = gain / math.sqrt(fan)

bound = math.sqrt(3.0) \* std # Calculate uniform bounds from standard deviation

return np.random.uniform(-bound, bound, size=inputs\_shape).astype(np.float32)

# 定义3X3卷积层函数

def \_conv3x3(in\_channel, out\_channel, stride=1, use\_se=False, res\_base=False):

if use\_se:

weight = conv\_variance\_scaling\_initializer(in\_channel, out\_channel, kernel\_size=3)

else:

weight\_shape = (out\_channel, in\_channel, 3, 3)

weight = Tensor(kaiming\_normal(weight\_shape, mode="fan\_out", nonlinearity='relu'))

if res\_base:

return nn.Conv2d(in\_channel, out\_channel, kernel\_size=3, stride=stride,

padding=1, pad\_mode='pad', weight\_init=weight)

return nn.Conv2d(in\_channel, out\_channel, kernel\_size=3, stride=stride,

padding=0, pad\_mode='same', weight\_init=weight)

# 定义1X1卷积层函数

def \_conv1x1(in\_channel, out\_channel, stride=1, use\_se=False, res\_base=False):

if use\_se:

weight = conv\_variance\_scaling\_initializer(in\_channel, out\_channel, kernel\_size=1)

else:

weight\_shape = (out\_channel, in\_channel, 1, 1)

weight = Tensor(kaiming\_normal(weight\_shape, mode="fan\_out", nonlinearity='relu'))

if res\_base:

return nn.Conv2d(in\_channel, out\_channel, kernel\_size=1, stride=stride,

padding=0, pad\_mode='pad', weight\_init=weight)

return nn.Conv2d(in\_channel, out\_channel, kernel\_size=1, stride=stride,

padding=0, pad\_mode='same', weight\_init=weight)

# 定义7X7卷积层函数

def \_conv7x7(in\_channel, out\_channel, stride=1, use\_se=False, res\_base=False):

if use\_se:

weight = conv\_variance\_scaling\_initializer(in\_channel, out\_channel, kernel\_size=7)

else:

weight\_shape = (out\_channel, in\_channel, 7, 7)

weight = Tensor(kaiming\_normal(weight\_shape, mode="fan\_out", nonlinearity='relu'))

if res\_base:

return nn.Conv2d(in\_channel, out\_channel,

kernel\_size=7, stride=stride, padding=3, pad\_mode='pad', weight\_init=weight)

return nn.Conv2d(in\_channel, out\_channel,

kernel\_size=7, stride=stride, padding=0, pad\_mode='same', weight\_init=weight)

# 定义Batch Norm层函数

def \_bn(channel, res\_base=False):

if res\_base:

return nn.BatchNorm2d(channel, eps=1e-5, momentum=0.1,

gamma\_init=1, beta\_init=0, moving\_mean\_init=0, moving\_var\_init=1)

return nn.BatchNorm2d(channel, eps=1e-4, momentum=0.9,

gamma\_init=1, beta\_init=0, moving\_mean\_init=0, moving\_var\_init=1)

# 定义最后一层的Batch Norm函数

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)

# 定义全连接层函数

def \_fc(in\_channel, out\_channel, use\_se=False):

if use\_se:

weight = np.random.normal(loc=0, scale=0.01, size=out\_channel \* in\_channel)

weight = Tensor(np.reshape(weight, (out\_channel, in\_channel)), dtype=mstype.float32)

else:

weight\_shape = (out\_channel, in\_channel)

weight = Tensor(kaiming\_uniform(weight\_shape, a=math.sqrt(5)))

return nn.Dense(in\_channel, out\_channel, has\_bias=True, weight\_init=weight, bias\_init=0)

# 构建残差模块

class ResidualBlock(nn.Cell):

expansion = 4 # conv2\_x--conv5\_x中，前两层的卷积核的个数是第三层（也就是输出通道）的4分之一。

def \_\_init\_\_(self,

in\_channel,

out\_channel,

stride=1,

use\_se=False, se\_block=False):

super(ResidualBlock, self).\_\_init\_\_()

self.stride = stride

self.use\_se = use\_se

self.se\_block = se\_block

# 前两层的卷积核个数等于输出通道的四分之一

channel = out\_channel // self.expansion

# 第一层卷积

self.conv1 = \_conv1x1(in\_channel, channel, stride=1, use\_se=self.use\_se)

self.bn1 = \_bn(channel)

# 第二层卷积

if self.use\_se and self.stride != 1:

self.e2 = nn.SequentialCell([\_conv3x3(channel, channel, stride=1, use\_se=True), \_bn(channel),

nn.ReLU(), nn.MaxPool2d(kernel\_size=2, stride=2, pad\_mode='same')])

else:

self.conv2 = \_conv3x3(channel, channel, stride=stride, use\_se=self.use\_se)

self.bn2 = \_bn(channel)

# 第三层卷积，其中卷积核个数等于输出通道

self.conv3 = \_conv1x1(channel, out\_channel, stride=1, use\_se=self.use\_se)

self.bn3 = \_bn(out\_channel)

# 使用SE模块

if self.se\_block:

self.se\_global\_pool = ops.ReduceMean(keep\_dims=False)

self.se\_dense\_0 = \_fc(out\_channel, int(out\_channel / 4), use\_se=self.use\_se)

self.se\_dense\_1 = \_fc(int(out\_channel / 4), out\_channel, use\_se=self.use\_se)

self.se\_sigmoid = nn.Sigmoid()

self.se\_mul = ops.Mul()

# Relu激活层

self.relu = nn.ReLU()

self.down\_sample = False

# 当步长不为1、或输出通道不等于输入通道时，进行图像下采样，用来调整通道数

if stride != 1 or in\_channel != out\_channel:

self.down\_sample = True

self.down\_sample\_layer = None

# 用1X1卷积调整通道数

if self.down\_sample:

if self.use\_se:

if stride == 1:

self.down\_sample\_layer = nn.SequentialCell([\_conv1x1(in\_channel, out\_channel,stride, use\_se=self.use\_se), \_bn(out\_channel)])

else:

self.down\_sample\_layer = nn.SequentialCell([nn.MaxPool2d(kernel\_size=2, stride=2, pad\_mode='same'), \_conv1x1(in\_channel, out\_channel, 1,use\_se=self.use\_se), \_bn(out\_channel)])

else:

self.down\_sample\_layer = nn.SequentialCell([\_conv1x1(in\_channel, out\_channel, stride,use\_se=self.use\_se), \_bn(out\_channel)])

# 构建残差块

def construct(self, x):

# 输入

identity = x

# 第一层卷积 1X1

out = self.conv1(x)

out = self.bn1(out)

out = self.relu(out)

# 第二层卷积 3X3

if self.use\_se and self.stride != 1:

out = self.e2(out)

else:

out = self.conv2(out)

out = self.bn2(out)

out = self.relu(out)

# 第三层卷积 1X1

out = self.conv3(out)

out = self.bn3(out)

if self.se\_block: # 使用SE模块

out\_se = out

out = self.se\_global\_pool(out, (2, 3))

out = self.se\_dense\_0(out)

out = self.relu(out)

out = self.se\_dense\_1(out)

out = self.se\_sigmoid(out)

out = ops.reshape(out, ops.shape(out) + (1, 1))

out = self.se\_mul(out, out\_se)

# 改变网络的维度

if self.down\_sample:

identity = self.down\_sample\_layer(identity)

# 加上残差

out = out + identity

# Relu激活

out = self.relu(out)

return out

# 构建残差网络

class ResNet(nn.Cell):

# 输入参数为：残差块，残差块重复数，输入通道，输出通道，步长，图像类别数，启用SE模块，残差块是否使用偏置项

def \_\_init\_\_(self,

block,

layer\_nums,

in\_channels,

out\_channels,

strides,

num\_classes,

use\_se=False,

res\_base=False):

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!")

self.use\_se = use\_se

self.res\_base = res\_base

self.se\_block = False

if self.use\_se:

self.se\_block = True

# 第一层卷积，输入通道3，输出通道64

if self.use\_se:

self.conv1\_0 = \_conv3x3(3, 32, stride=2, use\_se=self.use\_se)

self.bn1\_0 = \_bn(32)

self.conv1\_1 = \_conv3x3(32, 32, stride=1, use\_se=self.use\_se)

self.bn1\_1 = \_bn(32)

self.conv1\_2 = \_conv3x3(32, 64, stride=1, use\_se=self.use\_se)

else:

self.conv1 = \_conv7x7(3, 64, stride=2, res\_base=self.res\_base)

self.bn1 = \_bn(64, self.res\_base)

self.relu = ops.ReLU()

# 3X3池化层，步长2

if self.res\_base:

self.pad = nn.Pad(paddings=((0, 0), (0, 0), (1, 1), (1, 1)))

self.maxpool = nn.MaxPool2d(kernel\_size=3, stride=2, pad\_mode="valid")

else:

self.maxpool = nn.MaxPool2d(kernel\_size=3, stride=2, pad\_mode="same")

# conv2\_x残差块

self.layer1 = self.\_make\_layer(block,

layer\_nums[0],

in\_channel=in\_channels[0],

out\_channel=out\_channels[0],

stride=strides[0],

use\_se=self.use\_se)

# conv3\_x残差块

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

layer\_nums[1],

in\_channel=in\_channels[1],

out\_channel=out\_channels[1],

stride=strides[1],

use\_se=self.use\_se)

# conv4\_x残差块

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

layer\_nums[2],

in\_channel=in\_channels[2],

out\_channel=out\_channels[2],

stride=strides[2],

use\_se=self.use\_se,

se\_block=self.se\_block)

# conv5\_x残差块

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

layer\_nums[3],

in\_channel=in\_channels[3],

out\_channel=out\_channels[3],

stride=strides[3],

use\_se=self.use\_se,

se\_block=self.se\_block)

# 均值算子

self.mean = ops.ReduceMean(keep\_dims=True)

# Flatten层

self.flatten = nn.Flatten()

# 输出层

self.end\_point = \_fc(out\_channels[3], num\_classes, use\_se=self.use\_se)

# 输入参数为：残差块，残差块重复数，输入通道，输出通道，步长，使用SE模块，SE块

def \_make\_layer(self, block, layer\_num, in\_channel, out\_channel, stride, use\_se=False, se\_block=False):

# 搭建convn\_x的残差块

layers = []

resnet\_block = block(in\_channel, out\_channel, stride=stride, use\_se=use\_se)

layers.append(resnet\_block)

if se\_block:

for \_ in range(1, layer\_num - 1):

resnet\_block = block(out\_channel, out\_channel, stride=1, use\_se=use\_se)

layers.append(resnet\_block)

resnet\_block = block(out\_channel, out\_channel, stride=1, use\_se=use\_se, se\_block=se\_block)

layers.append(resnet\_block)

else:

for \_ in range(1, layer\_num):

resnet\_block = block(out\_channel, out\_channel, stride=1, use\_se=use\_se)

layers.append(resnet\_block)

return nn.SequentialCell(layers)

# 构建 SENet网络

def construct(self, x):

if self.use\_se: # 使用SE模块的输入层

x = self.conv1\_0(x)

x = self.bn1\_0(x)

x = self.relu(x)

x = self.conv1\_1(x)

x = self.bn1\_1(x)

x = self.relu(x)

x = self.conv1\_2(x)

else: # 卷积7X7，步长为2

x = self.conv1(x)

x = self.bn1(x) # 第一层的Batch Norm

x = self.relu(x) # Rule激活层

if self.res\_base:

x = self.pad(x)

c1 = self.maxpool(x) # 最大池化3X3，步长为2

c2 = self.layer1(c1) # conv2\_x残差块

c3 = self.layer2(c2) # conv3\_x残差块

c4 = self.layer3(c3) # conv4\_x残差块

c5 = self.layer4(c4) # conv5\_x残差块

out = self.mean(c5, (2, 3)) # 平均池化层

out = self.flatten(out) # Flatten层

out = self.end\_point(out) # 输出层

return out

# 构建SENet50 网络

def se\_resnet50(class\_num=1001,phase="train"):

return ResNet(ResidualBlock, # 残差块

[3, 4, 6, 3], # 残差块数量

[64, 256, 512, 1024], # 输入通道

[256, 512, 1024, 2048], # 输出通道

[1, 2, 2, 2], # 步长

class\_num, # 输出类别数

use\_se=True) # 使用SE模块

模型验证测试

模型验证是测试模型鲁棒性关键的一步。在训练过程中，模型会不断拟合训练数据，同时在验证数据集上不断验证，并保留在验证集上最佳准确率的模型权重，为了测试模型的鲁棒性，需要一批新的数据对模型进行测试，测试代码如下所示：

import os

import time

import datetime

import random

import mindspore

import glob

import numpy as np

import mindspore.nn as nn

from mindspore import Tensor, context

from mindspore.communication.management import init, get\_rank, get\_group\_size

from mindspore.train.model import Model

from mindspore.train.serialization import load\_checkpoint, load\_param\_into\_net

from mindspore.ops import operations as P

from mindspore.ops import functional as F

from mindspore.common import dtype as mstype

from mindspore.train.callback import ModelCheckpoint, CheckpointConfig, LossMonitor, TimeMonitor,Callback

from src.utils.logging import get\_logger

from src.seresnet import se\_resnet50

from src.dataset import vgg\_create\_dataset

from src.dataset import classification\_dataset

from model\_utils.moxing\_adapter import config

from model\_utils.moxing\_adapter import moxing\_wrapper

from model\_utils.device\_adapter import get\_device\_id, get\_rank\_id, get\_device\_num

class ParameterReduce(nn.Cell):

"""ParameterReduce"""

def \_\_init\_\_(self):

super(ParameterReduce, self).\_\_init\_\_()

self.cast = P.Cast()

self.reduce = P.AllReduce()

def construct(self, x):

one = self.cast(F.scalar\_to\_array(1.0), mstype.float32)

out = x \* one

ret = self.reduce(out)

return ret

def get\_top5\_acc(top5\_arg, gt\_class):

sub\_count = 0

for top5, gt in zip(top5\_arg, gt\_class):

if gt in top5:

sub\_count += 1

return sub\_count

def modelarts\_pre\_process():

'''modelarts pre process function.'''

def unzip(zip\_file, save\_dir):

import zipfile

s\_time = time.time()

if not os.path.exists(os.path.join(save\_dir, config.modelarts\_dataset\_unzip\_name)):

zip\_isexist = zipfile.is\_zipfile(zip\_file)

if zip\_isexist:

fz = zipfile.ZipFile(zip\_file, 'r')

data\_num = len(fz.namelist())

print("Extract Start...")

print("unzip file num: {}".format(data\_num))

data\_print = int(data\_num / 100) if data\_num > 100 else 1

i = 0

for file in fz.namelist():

if i % data\_print == 0:

print("unzip percent: {}%".format(int(i \* 100 / data\_num)), flush=True)

i += 1

fz.extract(file, save\_dir)

print("cost time: {}min:{}s.".format(int((time.time() - s\_time) / 60),

int(int(time.time() - s\_time) % 60)))

print("Extract Done.")

else:

print("This is not zip.")

else:

print("Zip has been extracted.")

if config.need\_modelarts\_dataset\_unzip:

zip\_file\_1 = os.path.join(config.data\_path, config.modelarts\_dataset\_unzip\_name + ".zip")

save\_dir\_1 = os.path.join(config.data\_path)

sync\_lock = "/tmp/unzip\_sync.lock"

# Each server contains 8 devices as most.

if config.device\_target == "GPU":

init()

device\_id = get\_rank()

device\_num = get\_group\_size()

elif config.device\_target == "Ascend":

device\_id = get\_device\_id()

device\_num = get\_device\_num()

else:

raise ValueError("Not support device\_target.")

# Each server contains 8 devices as most.

if device\_id % min(device\_num, 8) == 0 and not os.path.exists(sync\_lock):

print("Zip file path: ", zip\_file\_1)

print("Unzip file save dir: ", save\_dir\_1)

unzip(zip\_file\_1, save\_dir\_1)

print("===Finish extract data synchronization===")

try:

os.mknod(sync\_lock)

except IOError:

pass

while True:

if os.path.exists(sync\_lock):

break

time.sleep(1)

print("Device: {}, Finish sync unzip data from {} to {}.".format(device\_id, zip\_file\_1, save\_dir\_1))

config.log\_path = os.path.join(config.output\_path, config.log\_path)

@moxing\_wrapper(pre\_process=modelarts\_pre\_process)

def run\_eval():

"""run eval"""

config.per\_batch\_size = config.batch\_size

config.image\_size = list(map(int, config.image\_size.split(',')))

config.rank = get\_rank\_id()

config.group\_size = get\_device\_num()

config.device\_id = get\_device\_id()

\_enable\_graph\_kernel = config.device\_target == "GPU"

context.set\_context(mode=context.PYNATIVE\_MODE, enable\_graph\_kernel=False,

device\_target=config.device\_target, save\_graphs=False)

if os.getenv('DEVICE\_ID') == str(config.device\_id):

pass

else:

if os.getenv('DEVICE\_ID', "not\_set").isdigit() and config.device\_target == "Ascend":

context.set\_context(device\_id=int(os.getenv('DEVICE\_ID')))

config.outputs\_dir = os.path.join(config.log\_path,

datetime.datetime.now().strftime('%Y-%m-%d\_time\_%H\_%M\_%S'))

config.logger = get\_logger(config.outputs\_dir, config.rank)

config.logger.save\_args(config)

if config.dataset == "garbage\_photos":

net = se\_resnet50(class\_num=config.num\_classes,phase="test")

loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')

model = Model(net, loss\_fn=loss, metrics={'acc'})

param\_dict = load\_checkpoint(config.pre\_trained)

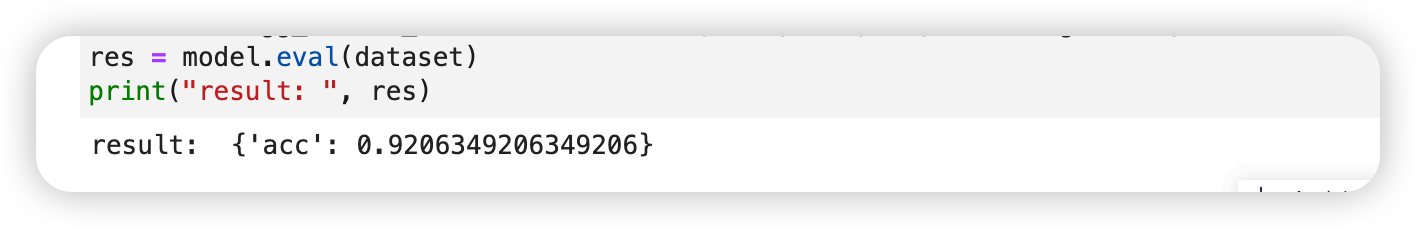
load\_param\_into\_net(net, param\_dict)

net.set\_train(False)

dataset = vgg\_create\_dataset(config.data\_dir, config.image\_size, config.per\_batch\_size, training=False,valid=False)

res = model.eval(dataset)

print("result: ", res)



模型导出

通过训练好的权重文件，调用mindspore.export()将模型导出成AIR模型，再使用ATC工具将AIR模型转为OM模型。

代码：

# 全局配置

weight\_ckpt\_path = "./pretrained/0-20\_277.ckpt"

deploy\_model\_filename = "deploy\_model/garbage\_deploy"

num\_classes = 40

image\_size = (224,224)

device\_target = "Ascend"

device\_id = 0

label\_list\_json\_path = "../data/label\_list.json"

# 导入依赖库

import os

import numpy as np

from mindspore import Tensor, context, export

import mindspore as ms

from mindspore import load\_checkpoint, load\_param\_into\_net

from src.seresnet import se\_resnet50

context.set\_context(mode=context.GRAPH\_MODE, device\_target=device\_target)

def run\_export(ckpt\_file, save\_filename, num\_classes, image\_size):

# 初始化模型

net = se\_resnet50(class\_num=num\_classes)

# 加载权重参数

load\_param\_into\_net(net, load\_checkpoint(ckpt\_file), strict\_load=True)

# 设置为eval模式

net.set\_train(False)

# 构建伪输入

input\_data = Tensor(np.zeros([1, 3, image\_size[0], image\_size[1]]), ms.float32)

# 导出模型

export(net, input\_data, file\_name=save\_filename, file\_format="AIR")

# 执行导出函数

run\_export(ckpt\_file=weight\_ckpt\_path,

save\_filename=deploy\_model\_filename,

num\_classes=num\_classes,

image\_size=image\_size)

模型转换

命令如下：

export PYTHONPATH=/home/ma-user/anaconda3/envs/MindSpore/lib/python3.7/site-packages:$PYTHONPATH

export install\_path=/usr/local/Ascend/ascend-toolkit/latest

export PATH=/usr/local/python3.7.5/bin:${install\_path}/atc/ccec\_compiler/bin:${install\_path}/atc/bin:$PATH

export PYTHONPATH=${install\_path}/atc/python/site-packages:$PYTHONPATH

export LD\_LIBRARY\_PATH=${install\_path}/atc/lib64:${install\_path}/acllib/lib64:$LD\_LIBRARY\_PATH

export ASCEND\_OPP\_PATH=${install\_path}/opp

export ASCEND\_AICPU\_PATH=/usr/local/Ascend/ascend-toolkit/latest

/usr/local/Ascend/ascend-toolkit/latest/atc/bin/atc --model='deploy\_model/garbage\_deploy.air' \

--framework=1 \

--output='deploy\_model/garbage\_deploy' \

--input\_format=NCHW \

--input\_shape='image:1,3,224,224' \

--log=error \

--soc\_version=Ascend310"

模型推理

#导入依赖库

import cv2

import numpy as np

import struct

import time

import acl

# 缩放图片

def resize\_image(image, target\_size):

h, w = image.shape[:2]

th, tw = target\_size

# 获取等比缩放后的尺寸

scale = min(th / h, tw / w)

oh, ow = round(h \* scale), round(w \* scale)

# 缩放图片，opencv缩放传入尺寸为（宽，高），这里采用线性差值算法

image = cv2.resize(image, (ow, oh), interpolation=cv2.INTER\_LINEAR).astype(np.uint8)

# 将剩余部分进行填充

new\_image = np.ones((th, tw, 3), dtype=np.uint8) \* 114

new\_image[:oh, :ow, :] = image

return new\_image

# 预处理函数

def process\_image(img\_path):

# 读取图片，opencv读图后格式是BGR格式，需要转为RGB格式

image = cv2.imread(img\_path, cv2.IMREAD\_COLOR)

image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

# 将图片等比resize至(224x224)

image = resize\_image(image, (224, 224))

image = np.array(image, dtype=np.float32)

# 将图片标准化

# (h,w,c) -> (c,h,w)

image = image.transpose((2, 0, 1))

return image

def softmax(f):

return np.exp(f) / np.sum(np.exp(f))

# 全局变量

ACL\_MEM\_MALLOC\_HUGE\_FIRST = 0

ACL\_MEMCPY\_HOST\_TO\_DEVICE = 1

ACL\_MEMCPY\_DEVICE\_TO\_HOST = 2

NPY\_BYTE = 1

# 构建ACL推理引擎类

class ACLEngine(object):

def \_\_init\_\_(self, model\_path, class\_map, device\_id=0):

self.device\_id = device\_id

self.context = None

self.model\_id = None

self.model\_desc = None

self.load\_input\_dataset, self.load\_output\_dataset = None, None

self.input\_data, self.output\_data = None, None

self.class\_map = class\_map

self.num\_classes = len(class\_map)

# --------------- pyACL初始化 ---------------

print('Start init resource')

self.\_init\_resource()

self.\_load\_model(model\_path)

self.\_prepare\_inputs()

def inference(self, img\_path):

""" 模型推理及后处理模块 """

# 1.读取并预处理图片

img = process\_image(img\_path)

# 2.准备模型推理的输入数据，运行模式默认为运行模式为ACL\_HOST，当前实例代码中模型只有一个输入。

bytes\_data = img.tobytes()

np\_ptr = acl.util.bytes\_to\_ptr(bytes\_data)

start\_time = time.time()

# 将图片数据从Host传输到Device。

ret = acl.rt.memcpy(self.input\_data[0]["buffer"], self.input\_data[0]["size"], np\_ptr,

self.input\_data[0]["size"], ACL\_MEMCPY\_HOST\_TO\_DEVICE)

# 3.执行模型推理。

# self.model\_id表示模型ID，在模型加载成功后，会返回标识模型的ID。

ret = acl.mdl.execute(self.model\_id, self.load\_input\_dataset, self.load\_output\_dataset)

# 4.处理模型推理的输出数据，输出置信度的类别编号。

inference\_result = []

for i, item in enumerate(self.output\_data):

buffer\_host, ret = acl.rt.malloc\_host(self.output\_data[i]["size"])

# 将推理输出数据从Device传输到Host。

ret = acl.rt.memcpy(buffer\_host, self.output\_data[i]["size"], self.output\_data[i]["buffer"],

self.output\_data[i]["size"], ACL\_MEMCPY\_DEVICE\_TO\_HOST)

bytes\_out = acl.util.ptr\_to\_bytes(buffer\_host, self.output\_data[i]["size"])

data = np.frombuffer(bytes\_out, dtype=np.byte)

inference\_result.append(data)

tuple\_st = struct.unpack(f"{self.num\_classes}f", bytearray(inference\_result[0]))

vals = np.array(tuple\_st).flatten()

vals = softmax(vals)

top\_k = vals.argsort()[-1:-6:-1]

print("\n======== inference results: =============")

for i, j in enumerate(top\_k):

print("top %d: class:[%s]: probability:[%f]" % (i, self.class\_map[str(j)], vals[j]))

end\_time = time.time()

print('inference cost time: {:.1f}ms\n'.format((end\_time-start\_time)\*1000))

def release\_resource(self):

""" 资源释放模块 """

self.\_unload\_model()

self.\_unload\_picture()

self.\_destroy\_resource()

print('Resource destroyed successfully')

def \_init\_resource(self):

# pyACL初始化

ret = acl.init()

# 运行管理资源申请

# 指定运算的Device。

self.device\_id = 0

ret = acl.rt.set\_device(self.device\_id)

# 显式创建一个Context，用于管理Stream对象。

self.context, ret = acl.rt.create\_context(self.device\_id)

def \_load\_model(self, model\_path):

# 加载离线模型文件，返回标识模型的ID。

self.model\_id, ret = acl.mdl.load\_from\_file(model\_path)

# 根据加载成功的模型的ID，获取该模型的描述信息。

self.model\_desc = acl.mdl.create\_desc()

ret = acl.mdl.get\_desc(self.model\_desc, self.model\_id)

def \_prepare\_inputs(self):

# 1.准备模型推理的输入数据集。

# 创建aclmdlDataset类型的数据，描述模型推理的输入。

self.load\_input\_dataset = acl.mdl.create\_dataset()

# 获取模型输入的数量。

input\_size = acl.mdl.get\_num\_inputs(self.model\_desc)

self.input\_data = []

# 循环为每个输入申请内存，并将每个输入添加到aclmdlDataset类型的数据中。

for i in range(input\_size):

buffer\_size = acl.mdl.get\_input\_size\_by\_index(self.model\_desc, i)

# 申请输入内存。

buffer, ret = acl.rt.malloc(buffer\_size, ACL\_MEM\_MALLOC\_HUGE\_FIRST)

data = acl.create\_data\_buffer(buffer, buffer\_size)

\_, ret = acl.mdl.add\_dataset\_buffer(self.load\_input\_dataset, data)

self.input\_data.append({"buffer": buffer, "size": buffer\_size})

# 2.准备模型推理的输出数据集。

# 创建aclmdlDataset类型的数据，描述模型推理的输出。

self.load\_output\_dataset = acl.mdl.create\_dataset()

# 获取模型输出的数量。

output\_size = acl.mdl.get\_num\_outputs(self.model\_desc)

self.output\_data = []

# 循环为每个输出申请内存，并将每个输出添加到aclmdlDataset类型的数据中。

for i in range(output\_size):

buffer\_size = acl.mdl.get\_output\_size\_by\_index(self.model\_desc, i)

# 申请输出内存。

buffer, ret = acl.rt.malloc(buffer\_size, ACL\_MEM\_MALLOC\_HUGE\_FIRST)

data = acl.create\_data\_buffer(buffer, buffer\_size)

\_, ret = acl.mdl.add\_dataset\_buffer(self.load\_output\_dataset, data)

self.output\_data.append({"buffer": buffer, "size": buffer\_size})

def \_unload\_model(self):

# 卸载模型。

ret = acl.mdl.unload(self.model\_id)

# 释放模型描述信息。

if self.model\_desc:

ret = acl.mdl.destroy\_desc(self.model\_desc)

self.model\_desc = None

# 释放Context。

if self.context:

ret = acl.rt.destroy\_context(self.context)

self.context = None

def \_unload\_picture(self):

# 释放输出资源，包括数据结构和内存。

while self.output\_data:

item = self.output\_data.pop()

ret = acl.rt.free(item["buffer"])

output\_number = acl.mdl.get\_dataset\_num\_buffers(self.load\_output\_dataset)

for i in range(output\_number):

data\_buf = acl.mdl.get\_dataset\_buffer(self.load\_output\_dataset, i)

if data\_buf:

ret = acl.destroy\_data\_buffer(data\_buf)

ret = acl.mdl.destroy\_dataset(self.load\_output\_dataset)

def \_destroy\_resource(self):

# 释放Device。

ret = acl.rt.reset\_device(self.device\_id)

# pyACL去初始化。

ret = acl.finalize()

推理执行

import json

with open(label\_list\_json\_path, 'r', encoding='utf-8') as json\_path:

class\_map = json.load(json\_path)

# 初始化推理引擎

engine = ACLEngine(f'{deploy\_model\_filename}.om', class\_map, device\_id=device\_id)

# 推理单张图片

engine.inference('./img\_335.jpg')

# 释放资源

engine.release\_resource()

输出：

Start init resource

======== inference results: =============

top 0: class:[其他垃圾/污损塑料]: probability:[1.000000]

top 1: class:[有害垃圾/过期药物]: probability:[0.000000]

top 2: class:[可回收物/纸板箱]: probability:[0.000000]

top 3: class:[可回收物/化妆品瓶]: probability:[0.000000]

top 4: class:[其他垃圾/一次性快餐盒]: probability:[0.000000]

inference cost time: 17.1ms

Resource destroyed successfully

## 实验总结

本实验整体流程如下：

1.数据集分析：首先对垃圾分类数据集进行简单的数据分析，查看数据分布状况以及数据状态。

2.模型训练：使用MindSpore框架构建图片分类算法SEResNet-50，使用垃圾分类数据集在Ascend910加速卡上进行模型训练。

3.模型验证：在测试集上测试模型的鲁棒性、准确率并获取性能指标。

4.模型导出：将训练好的模型进行导出，使之脱离MindSpore框架。

5.模型部署：将训练的模型部署并打包成应用，构建单张图片推理脚本，端到端的进行图片推理。

## 实验任务与参考解答任务

### 实验任务

本实验要求同学们完成完成垃圾分类模型部署，精度不低于80%的同时且推理时间不大于30ms。有能力的同学，试着精度达到90%，推理时间缩短到15ms以内。