昇腾创新实践课

毒蘑菇图像识别实验手册

华为技术有限公司

商标声明

 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.huawei.com

目录

1 实验环境介绍

1.1 实验介绍

1.1.1 关于本实验

本实验将使用 kaggle 的蘑菇数据集，在 ModelArts 平台上训练一个蘑菇分类器。

1.1.2 实验环境介绍

实验、介绍、难度、软件环境、硬件环境：

表 1-1 实验环境介绍

实验	实验介绍	难度	软件环境	开发环境
毒蘑菇图像识别识别实验	本实验将使用kaggle的蘑菇数据集，在ModelArts平台上训练一个蘑菇分类器。	困难	Python3.7.5 MindSpore 1.2.1 Numpy 1.17.5 matplotlib 3.3.4 easydict 1.9	华为云 ModelArts

华为云 ModelArts 和 OBS 的使用方法请参考：

1. ModelArts快速入门：https://bbs.huaweicloud.com/forum/thread-147236-1-1.html
2. ModelArts成长地图：https://support.huaweicloud.com/modelarts/index.html

2 毒蘑菇图像识别实验

2.1 实验介绍

2.1.1 关于本实验

全世界已知有 2000 多种野生食用菌的种类，云南占全国的 80%、全世界的 40%以上，全省境内有 126 个县城出产野生菌，每年吃菌的时间长达半年。

据说，每个云南人都有一个因吃菌中过毒的朋友。了解蘑菇种类，能帮助食客们分别毒蘑菇与可食用蘑菇。

本实验将使用蘑菇数据集，训练一个能分辨蘑菇种类的模型。

数据集介绍：

- 蘑菇数据集包含 9 个类，每个类有 300~1600 个长宽不等的彩色图像。
- 9 个类完全相互排斥，且类之间没有重叠。
- 本实验使用 ImageFolder 格式管理数据集，每一类图片整理成单独的一个文件夹，数据集结构如下：

```
└──ImageFolder
   ├──train
   │    class1Folder
   │    class2Folder
   │    ......
   └──test
        class1Folder
        class2Folder
        ......
```

- 数据来源：
  https://www.kaggle.com/maysee/mushrooms-classification-common-genuss-images

图 2-1

数据集包含以下 9 种蘑菇：

- Agaricus：双孢蘑菇，伞菌目，蘑菇科，蘑菇属，广泛分布于北半球温带，无毒。
- Amanita：毒蝇伞，伞菌目，鹅膏菌科，鹅膏菌属，主要分布于我国黑龙江、吉林、四川、西藏、云南等地，有毒。
- Boletus：丽柄牛肝菌，伞菌目，牛肝菌科，牛肝菌属，分布于云南、陕西、甘肃、西藏等地，有毒。
- Cortinarius：掷丝膜菌，伞菌目，丝膜菌科，丝膜菌属，分布于湖南等地(夏秋季在山毛等阔叶林地上生长)，有毒。
- Entoloma：霍氏粉褶菌，伞菌目，粉褶菌科，粉褶菌属，主要分布于新西兰北岛和南岛西部，有毒。
- Hygrocybe：浅黄褐湿伞，伞菌目，蜡伞科，湿伞属，分布于香港(见于松仔园)，有毒。
- Lactarius：松乳菇，红菇目，红菇科，乳菇属，广泛分布于亚热带松林地，无毒。
- Russula：褪色红菇，伞菌目，红菇科，红菇属，分布于河北、吉林、四川、江苏、西藏等地，无毒。
- Suillus：乳牛肝菌，牛肝菌目，乳牛肝菌科，乳牛肝菌属，分布于吉林、辽宁、山西、安徽、江西、浙江、湖南、四川、贵州等地，无毒。

2.1.2 实验目的

- 熟悉华为云 ModelArts 的使用。
- 掌握华为云 ModelArts 创建 MindSpore 开发环境。
- 掌握华为云 ModelArts 开发 MindSpore 实验流程。

2.1.3 背景知识

MindSpore 基础知识，MindSpore 进阶知识，图像数据预处理，卷积神经网络。

2.1.4 实验设计

1. 数据文件准备
2. 环境准备
3. 数据集下载和解压缩
4. 参数设置
5. 数据展示
6. 数据处理
7. 网络定义
8. 模型训练
9. 模型评估
10. 效果展示
11. 模型保存

2.2 实验过程

2.2.1 数据文件准备

本实验将使用 kaggle 的蘑菇数据集，在 ModelArts 平台上训练一个蘑菇分类器。

我们需要先将下载好的数据文件处理好，然后上传到 OBS 内。

本实验需要以下第三方库：

split-folders 0.4.3

步骤 1 数据下载

下载并解压 kaggle 的蘑菇数据集。原始数据集中目录如下，其中内层 Mushrooms 文件夹内的数据重复，手动删除。

```
Mushrooms
├── Agaricus
├── Amanita
├── Boletus
├── Cortinarius
├── Entoloma
├── Hygrocybe
├── Lactarius
├── Mushrooms # 重复数据，删除该文件夹
├── Russula
└── Suillus
```

步骤 2 数据清洗

Russula 类别下的 092_43B354vYxm8.jpg 为损坏数据：

图 2-2

图片整改要求：

- 不能出现模糊区域；
- 可通过裁剪、删除操作进行修改。

代码：

```
# 删除损坏的数据
import os
bad_data = os.path.join('Mushrooms','Russula','092_43B354vYxm8.jpg')
if os.path.exists(bad_data):
    os.remove(bad_data)
```

步骤 3 数据切分

使用 split-folders 切分文件夹形式数据。

【重要】切分前请确保数据文夹内仅包含以下 9 个文件夹：

- Agaricus
- Amanita
- Boletus
- Cortinarius
- Entoloma
- Hygrocybe
- Lactarius
- Russula
- Suillus

代码：

```
import splitfolders
splitfolders.ratio('Mushrooms', output="data", seed=1706, ratio=(.8, .0, .2)) # 这里 Mushrooms 是原始数据文件夹，data 是切分后的数据文件夹
```

步骤 4 打包并上传

将 data 文件夹打包成 data.zip，然后上传到 OBS 桶内。OBS 使用方法请参考 ModelArts 快速入门、ModelArts 成长地图。

2.2.2 环境准备

在华为云 ModelArts 创建 Notebook 工作环境：MindSpore1.2。Notebook 使用方法请参考 ModelArts 快速入门、ModelArts 成长地图。

步骤 1 Python 环境导入

代码：

```
import mindspore.dataset as ds # 数据集载入
import mindspore.nn as nn # 各类网络层都在 nn 里面
from mindspore.train.callback import ModelCheckpoint, CheckpointConfig, LossMonitor, TimeMonitor # 回调函数
from mindspore.train import Model # 承载网络结构
from mindspore import load_checkpoint # 读取最佳参数
from mindspore import context # 设置 mindspore 运行的环境

from easydict import EasyDict as ed # 超参数保存
import numpy as np # numpy
import matplotlib.pyplot as plt # 可视化

# 文件处理相关
import os

# 华为云文件传输相关
import moxing
```

步骤 2 MindSpore 环境设置

代码：

```
device_target = context.get_context('device_target') # 获取运行装置（CPU，GPU，Ascend）
dataset_sink_mode = True if device_target in ['Ascend','GPU'] else False # 是否将数据通过 pipeline 下发到装置上
context.set_context(mode = context.GRAPH_MODE, device_target = device_target) # 设置运行环境，静态图
context.GRAPH_MODE 指向静态图模型，即在运行之前会把全部图建立编译完毕

print(f'device_target: {device_target}')
print(f'dataset_sink_mode: {dataset_sink_mode}')
```

输出：

```
device_target: Ascend
dataset_sink_mode: True
```

2.2.3 数据集下载和解压缩

定义数据预处理函数。

函数功能包括：

1. 加载数据集
2. 打乱数据集
3. 图像特征处理（标准化、通道转换等）
4. 批量输出数据
5. 重复

代码：

```
# 将数据包从 OBS 下载到 ModelArts
```

```
print('输入数据路径（例：obs://桶名/目录/data.zip）')
src = input('数据路径：')
moxing.file.copy_parallel(src_url=src, dst_url='data.zip') #  下载数据

#  解压缩数据包
os.system('unzip data.zip')
```

输出：

```
输入数据路径（例：obs://桶名/目录/data.zip）
数据路径：obs://mindspore1706/mushroom/data.zip
0
```

2.2.4 参数设置

代码：

```
#  数据路径
train_path = os.path.join('data', 'train')
test_path = os.path.join('data', 'test')

#  超参数
config = ed({
    #  训练参数
    'batch_size': 32,
    'epochs': 150,

    #网络参数
    'class_num': 9,

    #  动态学习率调节
    'warmup_epochs': 5,
    'lr_init': 0.01,
    'lr_max': 0.1,

    #  优化器参数
    'momentum': 0.9,
    'weight_decay': 4e-5})
```

2.2.5 数据展示

代码：

```
#  创建图像标签列表
category_dict = {0:'Agaricus',1:'Amanita',2:'Boletus',3:'Cortinarius',4:'Entoloma',
                 5:'Hygrocybe',6:'Lactarius',7:'Russula',8:'Suillus'}

#  载入展示用数据
demo_ds = ds.ImageFolderDataset(test_path, decode=True)
```

```
# 设置图像大小
plt.figure(figsize=(6, 6))

# 打印 9 张子图
i = 1
for dic in demo_ds.create_dict_iterator():
    plt.subplot(3,3,i)
    plt.imshow(dic['image'].asnumpy()) # asnumpy：将 MindSpore tensor 转换成 numpy
    plt.axis('off')
    plt.title(category_dict[dic['label'].asnumpy().item()])
    i +=1
    if i > 9 :
        break

plt.show()
```

输出：

图 2-3

2.2.6 数据处理

步骤 1 计算数据集平均数和标准差

计算数据集平均数和标准差，数据标准化时使用

代码：

```
train_ds = ds.ImageFolderDataset(train_path, decode=True)
#计算数据集平均数和标准差，数据标准化时使用
tmp = np.asarray( [np.mean(x['image'], axis=(0, 1)) for x in train_ds.create_dict_iterator(output_numpy=True)] )
RGB_mean = tuple(np.mean(tmp, axis=(0)))
RGB_std = tuple(np.std(tmp, axis=(0)))

print(RGB_mean)
print(RGB_std)
```

输出：

```
(100.03388269705046, 94.57511259248079, 72.14921665851293)
(23.35913427414271, 20.336537235643164, 21.376613547858327)
```

步骤 2 自定义数据预处理 preprocess.py 文件

因蘑菇数据集各个类的样本数差异较大，为确保评估公正性，这里对验证集使用 PKSampler 进行平均采样（欠采样）。

代码：

```
import mindspore.dataset.vision.c_transforms as CV
import mindspore.dataset.transforms.c_transforms as C
import mindspore.common.dtype as mstype
import mindspore.dataset as ds

def create_dataset(data_path, mean=None, std=None, repeat_num=1, batch_size=32, usage='train'):
    """
    数据处理

    Args:
        dataset_path (str): 数据路径
        repeat_num (int): 数据重复次数
        batch_size (int): 批量大小
        usage (str): 训练或测试

    Returns:
        Dataset 对象
    """

    # 载入数据集
    if usage=='train':
        data = ds.ImageFolderDataset(data_path)
    else:
        # 每类取 63 个样本（最小类样本总数）
        sample_num = 63
        data = ds.ImageFolderDataset(data_path, sampler=ds.PKSampler(sample_num))

    # 打乱数据集
    data = data.shuffle(buffer_size=10000)
```

```
# 设定 resize 和 normalize 参数
image_size = 224
rgb_mean = mean
rgb_std = std

# 定义算子
if usage=='train':
    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.Normalize(mean=mean, std=std),
        CV.HWC2CHW()
    ]

type_cast_op = C.TypeCast(mstype.int32)


# 算子运算
data = data.map(operations=trans, input_columns="image")
data = data.map(operations=type_cast_op, input_columns="label")


# 批处理
if usage == 'train':
    drop_remainder = True
else:
    drop_remainder = False

data = data.batch(batch_size, drop_remainder=drop_remainder)

# 重复
data = data.repeat(repeat_num)

return data
```

步骤 3 调用 preprocess.py 文件

代码：

```
from preprocess import create_dataset
```

2.2.7 网络定义

本实验使用 ResNet50 网络。

ResNet 出自论文《Deep Residual Learning for Image Recognition》。

ResNet 又称残差网络，是由 Microsoft Research 提出的一种卷积神经网络，在 2015 年的 ImageNet 竞赛中获得图像分类和物体识别的优胜。ResNet 提出了"残差块"的概念，通过跳跃连接的方式缓解了在深度神经网络中增加深度带来的梯度消失问题，这使得 ResNet 能够通过增加相当的深度来提高准确率。

残差块示意图：

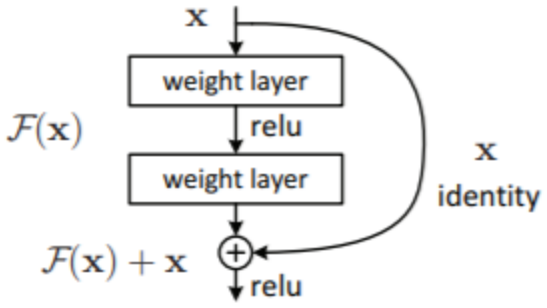

图 2-4

步骤 2 自定义残差网络 network.py 文件

代码：

```
import numpy as np
import mindspore.nn as nn
from mindspore.ops import operations as P
from mindspore.common.tensor import Tensor

# 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)

# 卷积层
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)

# 卷积层
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)

# 卷积层
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)

# 批量标准化层
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)

# 批量标准化层
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)

# 全连接层
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)

# 残差块
class ResidualBlock(nn.Cell):
    """
    ResNet V1 残差块定义

    Args:
        in_channel (int): 输入通道数
        out_channel (int): 输出通道数
        stride (int): 卷积层（3x3 卷积核）的步长

    Returns:
        Tensor, 输出张量

    Examples:
        >>> ResidualBlock(3, 256, stride=2)
    """
    expansion = 4

    def __init__(self,
                 in_channel,
                 out_channel,
```

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

        channel = out_channel // self.expansion
        self.conv1 = _conv1x1(in_channel, channel, stride=1)
        self.bn1 = _bn(channel)

        self.conv2 = _conv3x3(channel, channel, stride=stride)
        self.bn2 = _bn(channel)

        self.conv3 = _conv1x1(channel, out_channel, stride=1)
        self.bn3 = _bn_last(out_channel)

        self.relu = nn.ReLU()

        self.down_sample = False

        if stride != 1 or in_channel != out_channel:
            self.down_sample = True
        self.down_sample_layer = None

        if self.down_sample:
            self.down_sample_layer = nn.SequentialCell([_conv1x1(in_channel, out_channel, stride),
                                                        _bn(out_channel)])
        self.add = P.Add()

    def construct(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.down_sample:
            identity = self.down_sample_layer(identity)

        out = self.add(out, identity)
        out = self.relu(out)

        return out
```

```
# ResNet backbone  结构配置
class ResNet_backbone(nn.Cell):
    """
    ResNet architecture.

    Args:
        block (Cell):  残差块
        layer_nums (list):  每个 ResNet 子模块的残差块数量
        in_channels (list):  每个 ResNet 子模块的输入通道数
        out_channels (list):  每个 ResNet 子模块的输出通道数
        strides (list):   每个 ResNet 子模块的卷积层（3x3 卷积核）步长
    Returns:
        Tensor,  输出张量

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

    def __init__(self,
                 block,
                 layer_nums,
                 in_channels,
                 out_channels,
                 strides):
        super(ResNet_backbone, 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.conv1 = _conv7x7(3, 64, stride=2)
        self.bn1 = _bn(64)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, pad_mode="same")

        self.layer1 = self._make_layer(block,
                                       layer_nums[0],
                                       in_channel=in_channels[0],
                                       out_channel=out_channels[0],
                                       stride=strides[0])
        self.layer2 = self._make_layer(block,
                                       layer_nums[1],
                                       in_channel=in_channels[1],
                                       out_channel=out_channels[1],
                                       stride=strides[1])
```

```
        self.layer3 = self._make_layer(block,
                                       layer_nums[2],
                                       in_channel=in_channels[2],
                                       out_channel=out_channels[2],
                                       stride=strides[2])
        self.layer4 = self._make_layer(block,
                                       layer_nums[3],
                                       in_channel=in_channels[3],
                                       out_channel=out_channels[3],
                                       stride=strides[3])


    def _make_layer(self, block, layer_num, in_channel, out_channel, stride):
        """
        建立 ResNet 子模块

        Args:
            block (Cell):  残差块
            layer_num (int):  残差块数量
            in_channel (int):  输入通道数
            out_channel (int):  输出通道数
            stride (int):    卷积层（3x3 卷积核）的步长

        Returns:
            SequentialCell, ResNet 子模块

        Examples:
            >>> _make_layer(ResidualBlock, 3, 128, 256, 2)
        """
        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)

    def construct(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        c1 = self.maxpool(x)

        c2 = self.layer1(c1)
        c3 = self.layer2(c2)
```

```
        c4 = self.layer3(c3)
        c5 = self.layer4(c4)

        #print(c5.shape)
        #out = self.mean(c5, (2, 3))

        return c5

# ResNet head  结构配置
class ResNet_head(nn.Cell):
    """
    ResNet architecture.

    Args:
        backbone_out (int): backbone 输出维度
        num_classes (int):  数据集分类数量
    Returns:
        Tensor,  输出张量

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

    def __init__(self,
                 backbone_out,
                 num_classes):
        super(ResNet_head, self).__init__()

        self.avgpool = nn.AvgPool2d(kernel_size=7)
        self.flatten = nn.Flatten()
        self.end_point = _fc(backbone_out, num_classes)



    def construct(self, x):
        x = self.avgpool(x)
        x = self.flatten(x)
        out = self.end_point(x)

        return out


# ResNet50_backbone 定义
```

```
def resnet50_backbone():
    """
    ResNet50

    Args:

    Returns:
        Cell, ResNet50 backbone 网络

    Examples:
        >>> net = resnet50(10)
    """
    return ResNet_backbone(ResidualBlock,
                           [3, 4, 6, 3],
                           [64, 256, 512, 1024],
                           [256, 512, 1024, 2048],
                           [1, 2, 2, 2])

# ResNet50_head 定义
def resnet50_head(class_num=10):
    """
    ResNet50

    Args:
        class_num (int): 数据集分类数

    Returns:
        Cell, ResNet50_head 网络

    Examples:
        >>> net = resnet50(10)
    """
    return ResNet_head(2048, class_num)
```

步骤 3 调用 network.py 文件

代码：

```
# 这里将网络分为 backbone 和 head，backbone 是 ResNet 包含残差块的部分，head 是最后的全连接层。
from network import resnet50_backbone, resnet50_head

# 最终网络由 backbone 和 head 组成。
class ResNet50(nn.Cell):
    """
    ResNet architecture.

    Args:
        backbone (Cell): ResNet50 backbone 网络
```

```
        head (Cell): ResNet50 head 网络
    Returns:
        Tensor，输出张量

    Examples:
        >>> ResNet50(resnet_backbone,
        >>>                 resnet_head)
    """

    def __init__(self, backbone, head):
        super(ResNet50, self).__init__()

        self.backbone = backbone
        self.head = head

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

2.2.8 模型训练

步骤 1 载入数据集

代码：

```
# 训练集
train_data = create_dataset(data_path=train_path,
                            mean=RGB_mean,
                            std=RGB_std,
                            batch_size=config.batch_size,
                            usage='train',
                            repeat_num=1)

# 测试集
test_data = create_dataset(data_path=test_path,
                           mean=RGB_mean,
                           std=RGB_std,
                           batch_size=config.batch_size,
                           usage='test',
                           repeat_num=1)
```

步骤 2 自定义动态学习率 lr_scheduler.py 文件

设定动态学习率，加速模型收敛。

本实验学习率函数图如下：

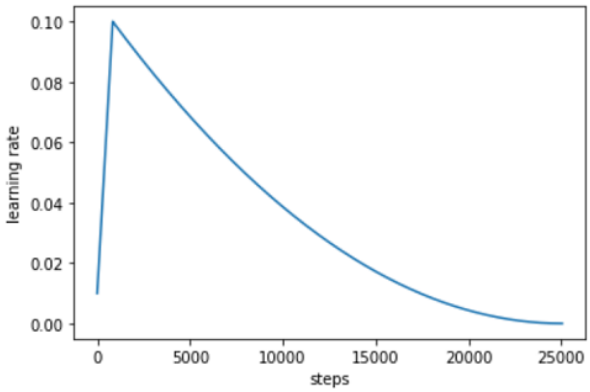

图 2-5

代码：

```
import numpy as np

def get_lr(total_epochs,
           steps_per_epoch,
           lr_init=0.01,
           lr_max=0.1,
           warmup_epochs=5):
    """
    生成学习率数组

    Args:
        total_epochs (int):  总 epoch 数
        steps_per_epoch (float):  每个 epoch 多少 step
        lr_init (float):  初始学习率
        lr_max (float):  最大学习率
        warmup_epochs (int):  预热 epoch 数

    Returns:
        numpy.ndarray，学习率数组
    """

    lr_each_step = [] #  学习率数组（回传）
    total_steps = steps_per_epoch * total_epochs #  总 step 数
    warmup_steps = steps_per_epoch * warmup_epochs #  预热 step 数

    #  计算预热阶段学习率递增值
    if warmup_steps != 0:
        inc_each_step = (float(lr_max) - float(lr_init)) / float(warmup_steps)
    else:
        inc_each_step = 0

    #  学习率调整
    for i in range(int(total_steps)):
        if i < warmup_steps:
```

```
            # 预热（学习率线性递增）
            lr = float(lr_init) + inc_each_step * float(i)
        else:
            # 衰减（学习率指数递减）
            base = ( 1.0 - (float(i) - float(warmup_steps)) / (float(total_steps) - float(warmup_steps)) )
            lr = float(lr_max) * base * base
            if lr < 0.0:
                lr = 0.0

        # 记录学习率
        lr_each_step.append(lr)

    lr_each_step = np.array(lr_each_step).astype(np.float32)

    return lr_each_step
```

步骤 3 调用 lr_scheduler.py 文件

代码：

```
from lr_scheduler import get_lr

# 训练 step 总数
train_step_size = train_data.get_dataset_size()

# 学习率数组
lr = get_lr(total_epochs=config.epochs,
            steps_per_epoch=train_step_size,
            lr_init=config.lr_init,
            lr_max=config.lr_max,
            warmup_epochs=config.warmup_epochs)
```

步骤 4 构建网络

选择网络、选择损失函数、优化器、模型。

代码：

```
# 网络
backbone_net = resnet50_backbone() # backbone 网络，保存后能提供后续迁移学习使用
head_net = resnet50_head(config.class_num) # head 网络，resnet50 最后的全连接层
net = ResNet50(backbone_net, head_net)

# 损失函数
net_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')

# 优化器
opt = nn.Momentum(net.trainable_params(), lr, momentum=config.momentum,
weight_decay=config.weight_decay)
```

```
# 模型
model = Model(net, loss_fn = net_loss,
                    optimizer = opt, metrics = {'accuracy','loss'})
```

步骤 5 自定义回调函数 callbacks.py 文件

自定义回调函数（EvalHistory 中新增保存最佳模型的 backbone 网络参数的功能）

代码：

```
from mindspore.train.callback import Callback
from mindspore import save_checkpoint
import os, stat, copy

# 记录模型 accuracy
class TrainHistroy(Callback):

    def __init__(self, history):
        super(TrainHistroy, self).__init__()
        self.history = history

    # 每个 epoch 结束时执行
    def epoch_end(self, run_context):
        cb_params = run_context.original_args()
        loss = cb_params.net_outputs.asnumpy()
        self.history.append(loss)


# 测试并记录模型在测试集的 loss 和 accuracy，每个 epoch 结束时进行模型测试并记录结果，跟踪并保存准确率最高的模型的网络参数
class EvalHistory(Callback):
    #保存 accuracy 最高的网络参数
    best_param = None
    best_param_backbone = None

    def __init__(self, model, backbone, loss_history, acc_history, eval_data):
        super(EvalHistory, self).__init__()
        self.model = model
        self.backbone = backbone
        self.loss_history = loss_history
        self.acc_history = acc_history
        self.eval_data = eval_data

    # 每个 epoch 结束时执行
    def epoch_end(self, run_context):
        cb_params = run_context.original_args()
        res = self.model.eval(self.eval_data, dataset_sink_mode=False)

        if len(self.acc_history)==0:
            self.best_param = copy.deepcopy(cb_params.network)
```

```
                self.best_param_backbone = copy.deepcopy(self.backbone)
            elif res['accuracy']>=max(self.acc_history):
                self.best_param = copy.deepcopy(cb_params.network)
                self.best_param_backbone = copy.deepcopy(self.backbone)

            self.loss_history.append(res['loss'])
            self.acc_history.append(res['accuracy'])

            print('acc_eval: ',res['accuracy'])

    # 训练结束后执行
    def end(self, run_context):
        # 保存最优网络参数
        if os.path.exists('best_param.ckpt'):
            os.chmod('best_param.ckpt', stat.S_IWRITE)
        if os.path.exists('best_param_backbone.ckpt'):
            os.chmod('best_param_backbone.ckpt', stat.S_IWRITE)
        save_checkpoint(self.best_param, 'best_param.ckpt')
        save_checkpoint(self.best_param_backbone, 'best_param_backbone.ckpt')
```

步骤 6 设置回调函数

代码：

```
time_cb = TimeMonitor(data_size=train_step_size) # 监控每次迭代的时间
loss_cb = LossMonitor() # 监控 loss 值
hist = {'loss':[], 'loss_eval':[], 'acc_eval':[]} # 训练过程记录

# 记录每次迭代的模型准确率
train_hist_cb = TrainHistroy(hist['loss'])

# 测试并记录模型在验证集的 loss 和 accuracy，并保存最优网络参数
eval_hist_cb = EvalHistory(model = model,
                           backbone = backbone_net,
                           loss_history = hist['loss_eval'],
                           acc_history = hist['acc_eval'],
                           eval_data = test_data)

cb = [time_cb, loss_cb, train_hist_cb, eval_hist_cb]
```

步骤 7 训练模型

代码：

```
model.train(config.epochs, train_data, callbacks=cb)
```

输出：

```
epoch: 1 step: 167, loss is 1.9619308
```

```
epoch time: 116120.809 ms, per step time: 695.334 ms
acc_eval:    0.23809523809523808
epoch: 2 step: 167, loss is 1.9744694
epoch time: 5230.776 ms, per step time: 31.322 ms
...
epoch: 149 step: 167, loss is 0.09291564
epoch time: 5655.406 ms, per step time: 33.865 ms
acc_eval:    0.7601410934744268
epoch: 150 step: 167, loss is 0.14568897
epoch time: 5622.981 ms, per step time: 33.671 ms
acc_eval:    0.7971781305114638
```

2.2.9 模型评估

步骤 1 可视化模型损失函数

代码：

```
# 定义 loss 记录绘制函数
def plot_loss(hist):
    plt.plot(hist['loss'], marker='.')
    plt.plot(hist['loss_eval'], marker='.')
    plt.title('loss record')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.grid()
    plt.legend(['loss_train', 'loss_eval'], loc='upper right')
    plt.show()
    plt.close()

plot_loss(hist)
```

输出：

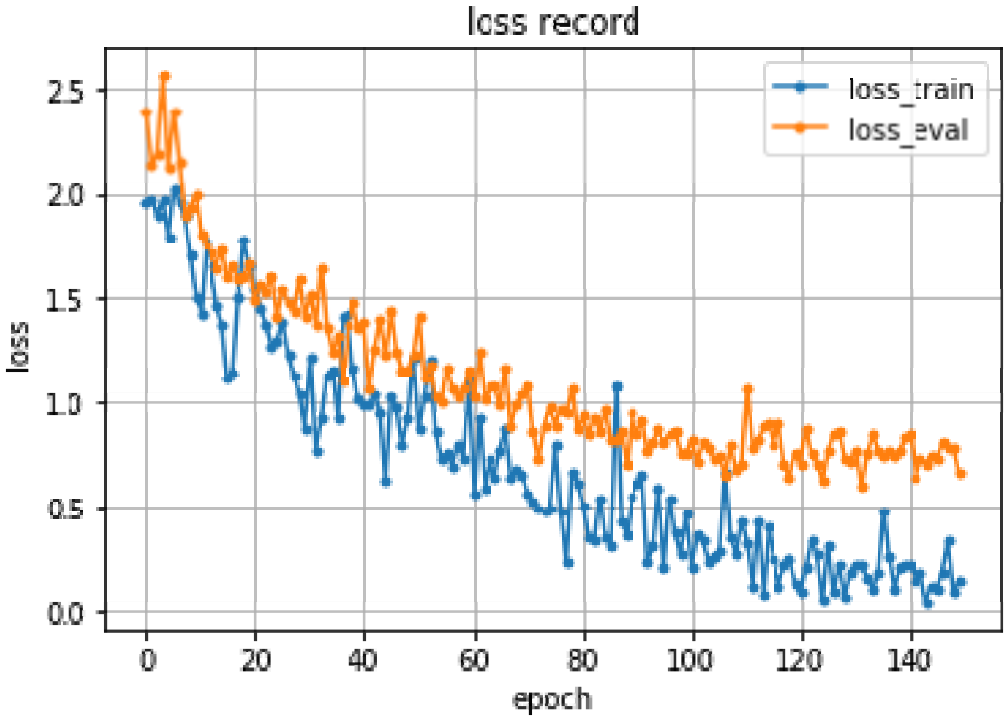

图 2-6

步骤 2 可视化模型准确率

代码：

```
def plot_accuracy(hist):
    plt.plot(hist['acc_eval'], marker='.')
    plt.title('accuracy history')
    plt.xlabel('epoch')
    plt.ylabel('acc_eval')
    plt.grid()
    plt.show()
    plt.close()

plot_accuracy(hist)
```

输出：

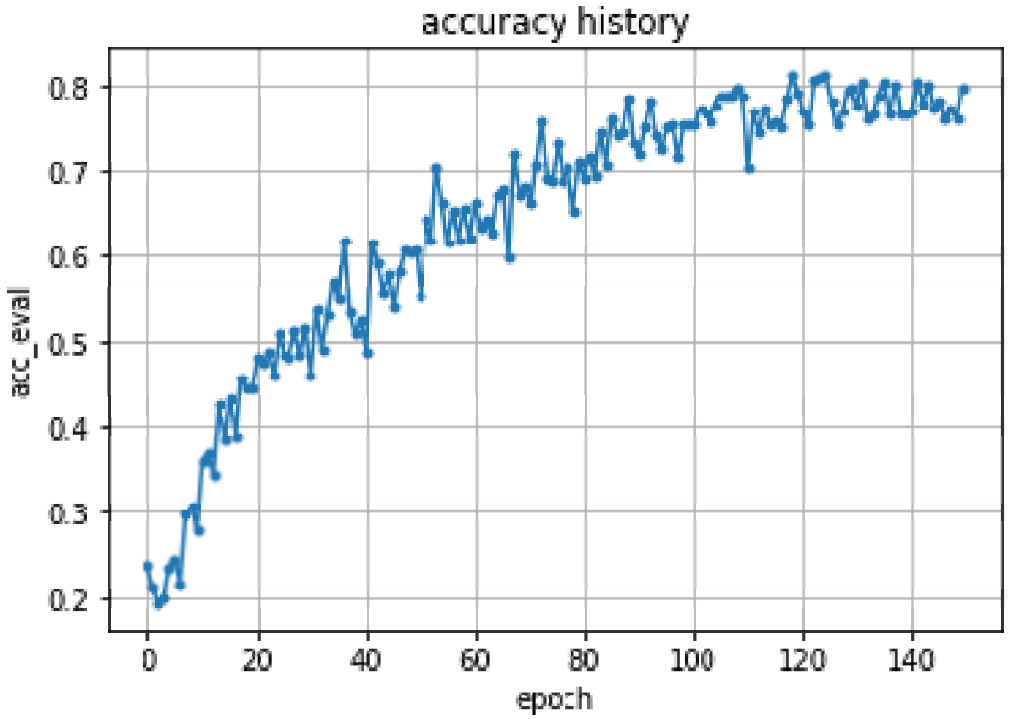

图 2-7

步骤 3 选择最优参数并验证

代码：

```
# 使用准确率最高的参数组合建立模型，并测试其在验证集上的效果
load_checkpoint('best_param.ckpt', net=net)
res = model.eval(test_data, dataset_sink_mode=False)
print(res)
```

输出：

```
{'loss': 0.773123272591167, 'accuracy': 0.7918871252204586}
```

2.2.10 效果展示

代码：

```
# 创建图像标签列表
category_dict = {0:'Agaricus',1:'Amanita',2:'Boletus',3:'Cortinarius',4:'Entoloma',
                 5:'Hygrocybe',6:'Lactarius',7:'Russula',8:'Suillus'}

ds_test_demo = create_dataset(test_path, mean=RGB_mean, std=RGB_std, batch_size=1, usage='test')

# 将数据标准化至 0~1 区间
def normalize(data):
    _range = np.max(data) - np.min(data)
    return (data - np.min(data)) / _range
```

```
# 设置图像大小
plt.figure(figsize=(10,10))
i = 1
# 打印 9 张子图
for dic in ds_test_demo.create_dict_iterator():
    # 预测单张图片
    input_img = dic['image']
    output = model.predict(input_img)
    predict = np.argmax(output.asnumpy(),axis=1)[0] # 反馈可能性最大的类别

    # 可视化
    plt.subplot(3,3,i)
    input_image = np.squeeze(input_img.asnumpy(),axis=0).transpose(1,2,0) # 删除 batch 维度
    input_image = normalize(input_image) # 重新标准化，方便可视化
    plt.imshow(input_image)
    plt.axis('off')
    plt.title('True: %s\n Predict: %s'%(category_dict[dic['label'].asnumpy().item()],category_dict[predict]))
    i +=1
    if i > 9 :
        break

plt.show()
```

输出：

图 2-8

2.2.11 模型保存

若要保存模型参数文件（ckpt），停止 ModelArts 环境前需先将保存在 ModelArts 暂存的模型参数文件上传到 OBS。

代码：

```
# 将 ckpt 上传到 OBS
print('输入目标路径（例：obs://桶名/目录/best_param_backbone.ckpt）')
dst = input('目标路径：')
moxing.file.copy_parallel(src_url='best_param_backbone.ckpt', dst_url=dst)
```

输出：

```
输入目标路径（例：obs://桶名/目录/best_param_backbone.ckpt）
目标路径：obs://mindspore1706/mushroom/best_param_backbone.ckpt
```

2.3 实验总结

本实验介绍了 MindSpore 在华为云 ModelArts 平台使用 Ascend 910 芯片完成图像分类的任务，使用大规模的蘑菇图片数据集，搭建复杂的 ResNet50 的神经网络，调用华为云 ModelArts 平台的云端算力进行快速训练，最终实现毒蘑菇图像分类的任务。

2.4 课后作业：蘑菇分类模型 Fine-Tune

上一个阶段使用 ModelArts 训练了一个蘑菇分类器，该分类器可以对 9 种蘑菇进行分类。

现在，请使用上一个阶段训练好的网络作为本作业的 backbone 网络，训练一个可以分类另外 2 种新蘑菇的分类器。

2.4.1 数据集

本作业使用 kaggle 的另一个蘑菇数据集，只需对其中的两种新蘑菇进行分类：

- Exidia：黑耳，又称为黑胶菌，分布于我国吉林、河北、山西、宁夏、青海、四川、广西、海南,西藏等地，无毒。
- Inocybe：丝盖伞，又称毛丝盖菌、毛锈伞，分布于我国河北、吉林、江苏、山西、四川、云南、甘肃、新疆等地，有毒。

2.4.2 实验设计要求

1. 环境准备
2. 参数设置
3. 数据处理
4. 网络定义
5. 特征提取
6. 模型训练（需要自定义动态学习率）
7. 模型评估（需要可视化 loss 和 accuracy）
8. 效果展示
9. 模型保存