华为认证 AI 系列教程

HCIP-AI-MindSpore Developer

MindSpore简介与API详解

实验指导手册

版本：1.0

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

华为认证体系介绍

华为认证是华为公司基于“平台+生态”战略，围绕“云-管-端”协同的新ICT技术架构，打造的覆盖ICT（Information and Communications Technology 信息通信技术）全技术领域的认证体系，包含ICT技术架构与应用认证、云服务与平台认证两类认证。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

华为认证覆盖ICT全领域，符合ICT融合的技术趋势，致力于提供领先的人才培养体系和认证标准，培养数字化时代新型ICT人才，构建良性ICT人才生态。

HCIP-AI-MindSpore Developer V1.0 定位于培养和认证具备使用华为深度学习框架MindSpore进行人工智能开发能力的算法工程师。

通过HCIP-AI-MindSpore Developer V1.0认证，您将掌握MindSpore和MindSpore Lite的架构与特性相关知识，理解MindSpore网络迁移、分布式训练、端云侧推理与部署等相关特性；具备使用MindSpore完成图像处理任务和自然语言处理任务的能力；能够胜任人工智能算法工程师、深度学习工程师、图像算法工程师、自然语言处理算法工程师等岗位。


Huawei Certification
Huawei Certified ICT Expert
Huawei Certified ICT Professional
Huawei Certified ICT Associate
HCIE
HCIP
HCIA
Cloud Service & Platform
云服务与平台
ICT Infrastructure
ICT技术架构与应用
HarmonyOS(To Be)
IoT
GaussDB
openGauss (To Be)
openEuler
Cloud(Cloud Computing + Cloud Service)
Intelligent Driving (To Be)
Big Data
AI
Kunpeng Application Developer
Storage
Datacom
Transmission
Intelligent Computing
WLAN
Access
Enterprise Communication
Security
LTE
Intelligent Vision
SDN
5G
Data Center

前 言

简介

本书为 HCIP-AI-MindSpore V1.0 认证培训教程，适用于准备参加 HCIP-AI-MindSpore 考试的学员或者希望了解昇腾 AI 体系相关知识、MindSpore 开发框架使用、转换与跨平台部署的读者。

内容描述

本实验指导书共包含 5 个实验，基于 MindSpore 部署。从数据定义，再到损失函数的学习，最后为评价指标和训练，完成基本训练流程。

- 实验一为 API 基础，帮助读者掌握典型 API 模块定义方法。
- 实验二为自定义数据，通过将数据集转化为 MindRecord，帮助读者掌握转化方法。
- 实验三介绍了损失函数的定义，损失函数可以继承自 cell 和 loss 两个基类。
- 实验四为评价指标，通过 Class 实现一个自定义的 Metric 功能，帮助读者掌握如何使用 Metric 的方法。
- 实验五介绍了采用 TrainOneStepCell，实现更加灵活的训练。

读者知识背景

本课程为华为认证基础课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有基本的人工智能知识背景，同时熟悉华为 MindSpore 框架，了解人工智能开发流程。

实验环境说明

- MindSpore：实验一环境是基于 MindSpore1.3.0 进行的。
- 实验平台：个人便携笔记本电脑（ Windows)。

目录

1 MindSpore 框架和特性

1.1 实验介绍

1.1.1 关于本实验

本实验通过 API 算法，自定义数据，损失函数，评价指标和训练，帮助读者掌握模型训练的基本流程和使用的定义方法。

1.1.2 实验目的

- 掌握典型 API 模块定义。
- 理解 MindRecord 性能优化。
- 掌握定义数据集与网络的方法。
- 掌握基于 Loss 的评价指标。

1.1.3 MindSpore 实验介绍

MindSpore 是一个全场景深度学习框架，旨在实现易开发、高效执行、全场景覆盖三大目标，其中易开发表现为 API 友好、调试难度低，高效执行包括计算效率、数据预处理效率和分布式训练效率，全场景则指框架同时支持云、边缘以及端侧场景。

MindSpore 提供了 Python 编程范式，用户使用 Python 原生控制逻辑即可构建复杂的神经网络模型，AI 编程变得简单。

MindSpore 总体架构如下图所示，下面介绍主要的扩展层（MindSpore Extend）、前端表达层（MindExpress，ME）、编译优化层（MindCompiler）和全场景运行时（MindRE）四个部分。

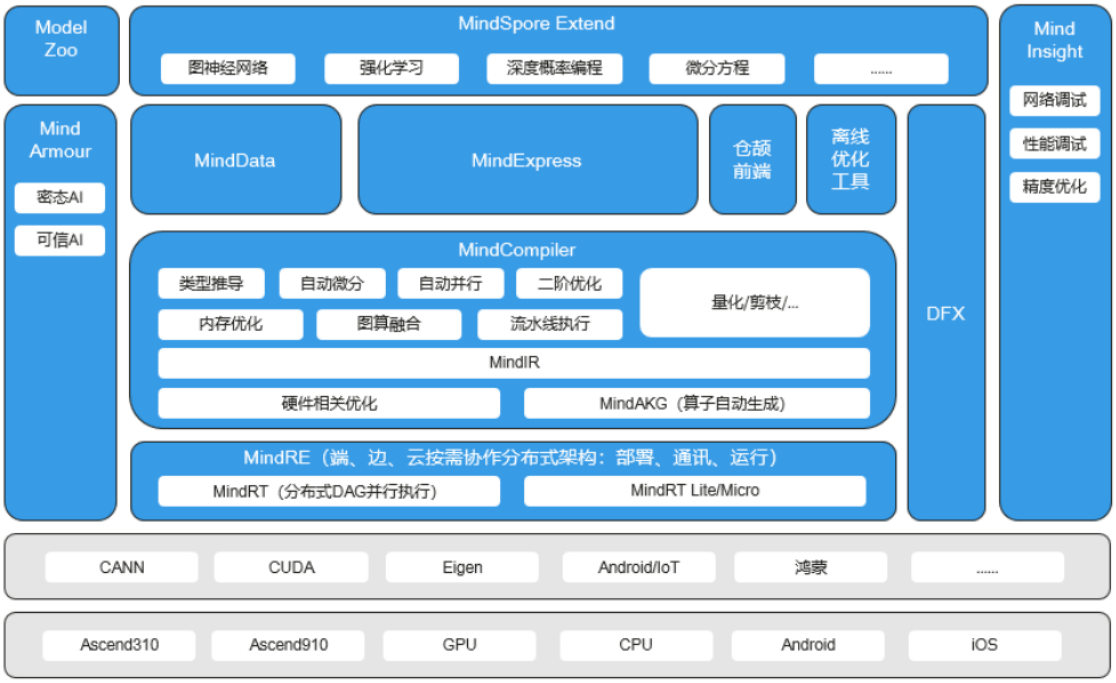

图1-1 MindSpore 总体架构图

MindSpore Extend（扩展层）：MindSpore 扩展包，期待更多开发者来一起贡献和构建。

MindExpress（表达层）：基于 Python 的前端表达，未来计划陆续提供 C/C++、Java 等不同的前端；MindSpore 也在考虑支持华为自研编程语言前端-仓颉，目前还处于预研阶段；同时也在做与 Julia 等第三方前端的对接工作，引入更多的第三方生态。

MindCompiler（编译优化层）：图层的核心编译器，主要基于端云统一的 MindIR 实现三大功能，包括硬件无关的优化（类型推导、自动微分、表达式化简等）、硬件相关优化（自动并行、内存优化、图算融合、流水线执行等）、部署推理相关的优化（量化、剪枝等）；其中，MindAKG 是 MindSpore 的自动算子生成编译器，目前还在持续完善中。

MindRE（全场景运行时）：这里含云侧、端侧以及更小的 IoT。

1.2 MindSpore1.3.0 版本安装

1.2.1 安装过程

MindSpore 是华为推出的 AI 开源开发框架，本实验基于 1.3.0 版本。安装默认为采用 Windows 系统下的 CPU 版本。

步骤 1　下载 Anaconda

登陆 Anaconda 官网下载安装包 https://www.anaconda.com/distribution/#download-section。

根据系统不同，可以选择 Windows、Mac、Linux 版本，这里选择 Windows，点击 64-Bit Graphical Installer 下载。

步骤 2　安装 Anaconda

双击下载好的 Anaconda3-x.x.x-Windows-x86_64.exe 文件，进行安装。

在环境变量配置页（图 1-2），建议两个都勾选，第一个是加入环境变量，第二个是默认使用 Anaconda 对应的 Python 包，可以减少我们后续配置的工作，点击 Install 开始安装。

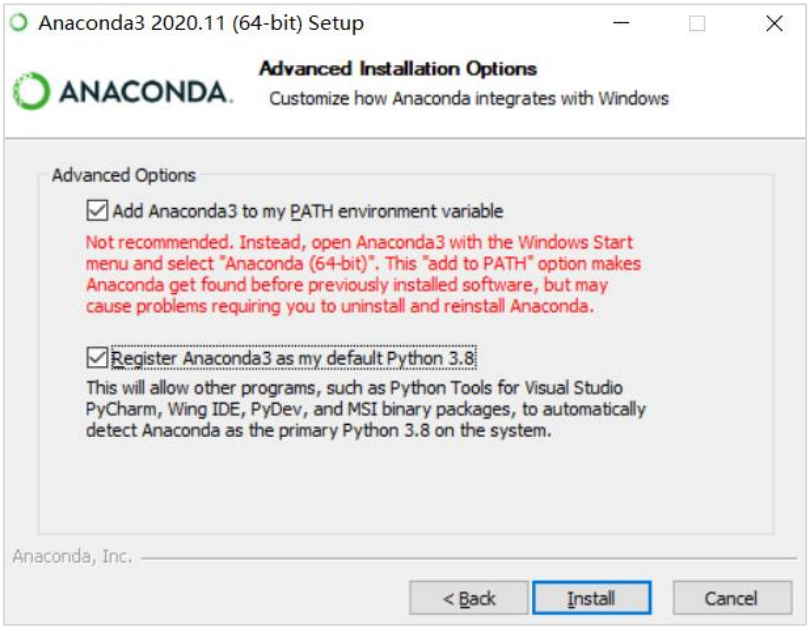

图1-2 环境变量选项

步骤 3　配置 Conda 与 Pip 源

通常情况下，使用 conda 命令或者 pip 命令都是从国外的服务器上下载我们需要的模块包的，这在网速不佳的情况下会消耗大量的时间。所以这里我们建议更换国内的源来进行模块下载，速度大大提升。

打开命令行 cmd 工具，输入以下命令。

Conda 换源

```
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/
```

Pip 换源

```
pip config set global.index-url https://pypi.douban.com/simple
```

步骤 4　创建虚拟环境并激活

打开命令行，输入命令

```
conda create -n ms python=3.7.5
```

输入 y 后回车

安装完成后激活虚拟环境

```
conda activate ms
```

步骤 5　安装 MindSpore1.3.0

安装 MindSpore 参考官网安装页面：

https://www.mindspore.cn/install

https://www.mindspore.cn/versions

图1-3 1.3.0 版本下载地址

输入以下命令进行安装：

```
pip install https://ms-release.obs.cn-north-4.myhuaweicloud.com/1.3.0/MindSpore/cpu/x86_64/mindspore-1.3.0-cp37-cp37m-win_amd64.whl --trusted-host ms-release.obs.cn-north-4.myhuaweicloud.com -i https://pypi.tuna.tsinghua.edu.cn/simple
```

步骤 6　验证安装

输入命令

```
python
import mindspore as ms
print(ms.__version__)
```

若安装成功结果显示应如下图：

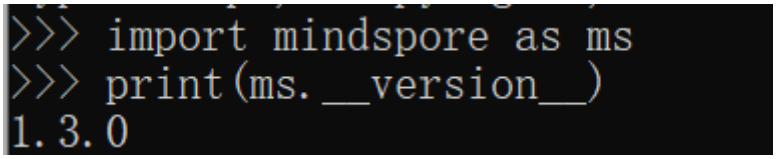

图1-4 安装成功结果显示

步骤 7　安装 Jupyter Notebook

输入以下命令：

```
pip install jupyter#码包下载，Jupyter Notebook 打开可直接输入命令 jupyter notebook
```

1.3 下载源码包

当我们需要对 MindSpore 底层进行学习与理解时可以登录 MindSpore GitHub 网站对源码进行下载：https://github.com/mindspore-ai/mindspore

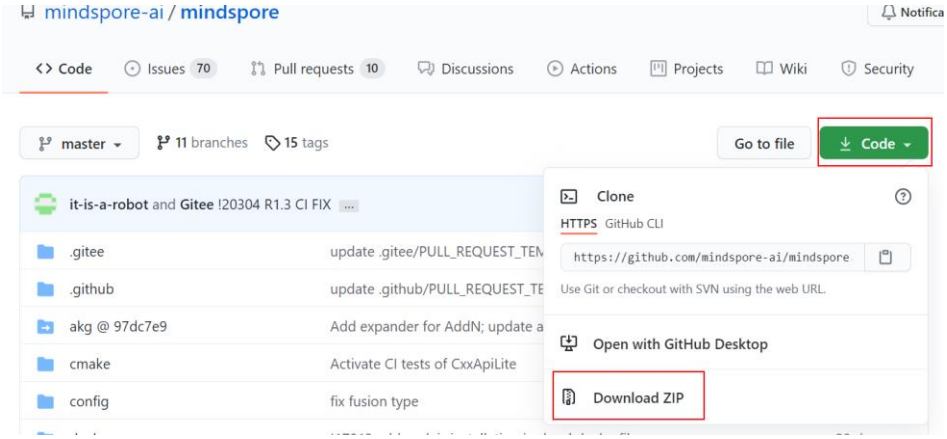

图1-5 Mindspore 代码下载

1.4 API 基础

1.4.1 Dataset

该模块提供加载和处理各种通用数据集的 API，如 MNIST、CIFAR-10、CIFAR-100、VOC、COCO、ImageNet、CelebA、CLUE 等。它还支持标准格式的数据集，包括 MindRecord、TFRecord、清单等。用户还可以使用此模块定义自己的数据集。

我们以 MNIST 数据集为例，MNIST 手写数字数据库有一个 60,000 个示例的训练集和一个 10,000 个示例的测试集。它是 NIST 提供的较大集的子集。数字已被大小标准化，并在固定大小的图像中居中。

在 Jupyter Notebook 中运行以下命令来下载 MNIST 数据集的训练图像和标签并解压，存放在./datasets/MNIST_Data 路径中，目录结构如下:

```
#创建目录并下载相关数据集
!mkdir .\datasets\MNIST_Data\train .\datasets\MNIST_Data\test
!curl -O https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap2/train-labels-idx1-ubyte
!curl -O https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap2/train-images-idx3-ubyte
!curl -O https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap2/t10k-labels-idx1-ubyte
!curl -O https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap2/t10k-images-idx3-ubyte
```

执行结果：

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
```

```
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 60008  100 60008    0     0  60008      0  0:00:01 --:--:--  0:00:01 62967
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0 44.8M    0  392k    0     0   392k      0  0:01:57 --:--:--  0:01:57  738k
 25 44.8M   25 11.4M    0     0  11.4M      0  0:00:03  0:00:01  0:00:02 7725k
 51 44.8M   51 22.8M    0     0  11.4M      0  0:00:03  0:00:02  0:00:01 9316k
 77 44.8M   77 34.5M    0     0  11.5M      0  0:00:03  0:00:03 --:--:--  9.8M
100 44.8M  100 44.8M    0     0  11.2M      0  0:00:04  0:00:04 --:--:-- 10.2M
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 10008  100 10008    0     0  10008      0  0:00:01 --:--:--  0:00:01 45698
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
 61 7656k   61 4728k    0     0  4728k      0  0:00:01 --:--:--  0:00:01 5703k
100 7656k  100 7656k    0     0  7656k      0  0:00:01  0:00:01 --:--:-- 7312k
```

移动文件：

```
!move t* .\datasets\MNIST_Data\train
```

文件夹 PATH 列表：

```
!tree /F ./datasets/MNIST_Data
```

MindSpore 目前支持加载图像领域常用的经典数据集和多种数据存储格式下的数据集，用户也可以通过构建自定义数据集类实现自定义方式的数据加载。

```
#配置数据集目录，创建 MNIST 数据集对象
import mindspore.dataset as ds
DATA_DIR = './datasets/MNIST_Data/train'
mnist_dataset = ds.MnistDataset(DATA_DIR, num_samples=6, shuffle=False)
```

创建字典迭代器，通过迭代器获取一条数据，并将数据进行可视化。

```
import matplotlib.pyplot as plt

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()
```

执行结果：

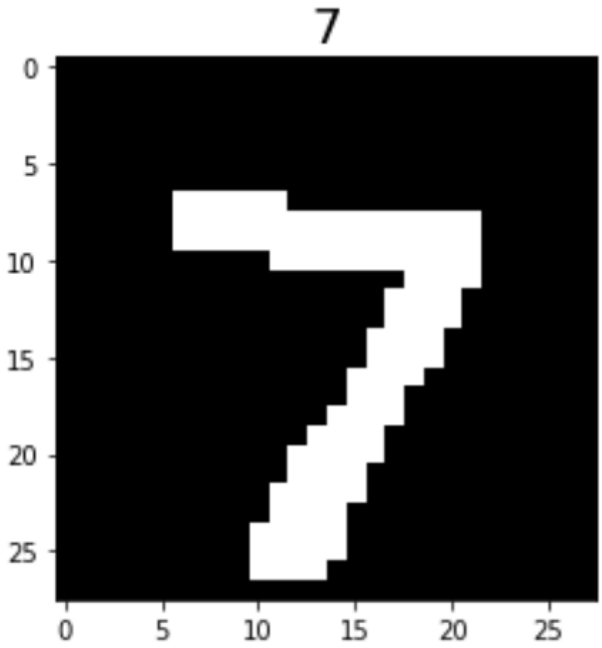

图1-6 Mnist 数据

1.4.2 Dataset.vision

本模块用于支持视觉增强。它包括两部分：C_transforms 和 Py_transforms。常见的功能比如 mindspore_dataset.vision.c_transforms.HWC2CHW 指的是将输入图像从形状（H、W、C）转置到形状（C、H、W）。输入的图片应该为三通道图片。

下面演示使用 c_transforms 模块对 MNIST 数据集进行数据增强。

```
#导入相关模块，重新加载数据集
from mindspore.dataset.vision import Inter
import mindspore.dataset.vision.c_transforms as transforms
import mindspore.dataset as ds
DATA_DIR = './datasets/MNIST_Data/train'
mnist_dataset = ds.MnistDataset(DATA_DIR, num_samples=6, shuffle=False)
```

定义数据增强算子，对数据集执行 Resize 和 RandomCrop 操作。

```
resize_op = transforms.Resize(size=(200,200), interpolation=Inter.LINEAR)
crop_op = transforms.RandomCrop(150)
transforms_list = [resize_op, crop_op]
ds4 = mnist_dataset.map(operations=transforms_list,input_columns='image')
```

查看数据增强效果：

```
mnist_it = ds4.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()
```

执行结果：

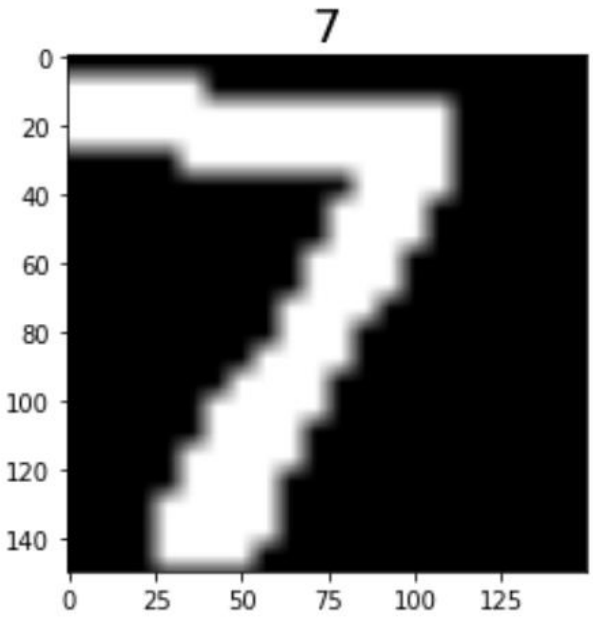

图1-7 数据增强

1.4.3 nn

神经网络细胞，用于构建神经网络的预定义构建块或计算单元。包括 cell，containers，convolution layers，损失函数，优化器函数，指标等。

我们以 ReduceLogSumExp 为例，它通过计算维度中所有元素的指数，然后计算和的对数，减少张量的维度。

```
import numpy as np
import mindspore.ops as ops
from mindspore import Tensor,nn
from mindspore.common import dtype as astype

x = Tensor(np.random.randn(3, 4, 5, 6).astype(np.float32))
op = nn.ReduceLogSumExp(1, keep_dims=True)#调用 nn 中的 ReduceLogSumExp 函数
output = op(x)
print(output.shape)
```

执行结果：

```
(3，1，5，6)
```

1.4.4 numpy

mindspore.numpy 大致分为四类：Array Gernation，Array Operation，Logic 和 Math。引入代码如下：

```
import mindspore.numpy as np
```

我们以 Array Generation 中的 arrange 为例，它可以返回给定间隔内均匀间隔的值。

```
#定义函数
mindspore.numpy.arrange(start, stop=None, step=None, dtype=None)
```

其中，参数 start 指的是间隔的开始，默认值为 0；stop 指的是间隔结束，间隔不包括此值；step 指的是值之间的间距，对于任何输出，这是两个相邻值之间的距离。

```
import mindspore.numpy as np
print(np.arange(0, 5, 1)) #start=0，stop=5，step=1
#执行结果
[0 1 2 3 4]
print(np.arange(3)) #start=0，stop=3，step=1
#执行结果
[0 1 2]
print(np.arange(start=0, stop=3)) #step=1
#执行结果
[0 1 2]
print(np.arange(0, stop=3, step=0.5))
#执行结果
[0.  0.5 1.  1.5 2.  2.5]
print(np.arange(stop=3)) # This will lead to TypeError
```

1.4.5 ops

运算符可用于构造 cell 函数。引入 mindspore 中的运算符代码如下：

```
import mindspore.ops as ops
```

我们以批量归一化函数为例，批量归一化在卷积神经网络中得到了广泛的应用。此操作对输入应用批处理标准化，以避免内部协变量偏移，如论文批处理标准化：通过减少内部协变量偏移加速深度网络训练中所述。它使用小批数据重新缩放和更新特征，学习的参数可以在以下公式中描述。

$$y = \frac{x - mean}{\sqrt{variance + \epsilon}} * \gamma + \beta$$

其中 γ 是标度， β 是偏差， ε 是 ε ，平均值是 x 的平均值，方差是 x 的方差。

具体代码参考如下：

```
from mindspore import Tensor
import numpy as np
from mindspore import ops
from mindspore.common import dtype as mindspore

input_x = Tensor(np.ones([2, 2]), mindspore.float32)
scale = Tensor(np.ones([2]), mindspore.float32)
bias = Tensor(np.ones([2]), mindspore.float32)
mean = Tensor(np.ones([2]), mindspore.float32)
variance = Tensor(np.ones([2]), mindspore.float32)
batch_norm = ops.BatchNorm()
output = batch_norm(input_x, scale, bias, mean, variance)
print(output[0])
```

执行结果：

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

1.4.6 train

train 函数可以训练所有神经网络。例如回调函数是 MindSpore 自定义在 train 的过程中可实时进行验证，继承 callback 类自定义 evalcallback，可设置为每隔几个 epoch 进行验证，实时输出 model 指定的 metrics 评价指标。通过实例化对象，将这个回调过程放入 model.train()中的 callback()中。

下面是具体参考代码：

```
from mindspore import context
import numpy as np
from mindspore import dataset as ds
from mindspore import nn, Tensor, Model
import time
from mindspore.train.callback import Callback, LossMonitor
import mindspore as ms
ms.common.set_seed(0)

def get_data(num, a=2.0, b=3.0, c=5.0):
    for _ in range(num):
        x = np.random.uniform(-1.0, 1.0)
        y = np.random.uniform(-1.0, 1.0)
        noise = np.random.normal(0, 0.03)
        z = a * x ** 2 + b * y ** 3 + c + noise
        yield np.array([[x**2], [y**3]],dtype=np.float32).reshape(1,2), np.array([z]).astype(np.float32)

def create_dataset(num_data, batch_size=16, repeat_size=1):
    input_data = ds.GeneratorDataset(list(get_data(num_data)), column_names=['xy','z'])
    input_data = input_data.batch(batch_size)
    input_data = input_data.repeat(repeat_size)
    return input_data

data_number = 160
batch_number = 10
repeat_number = 10

ds_train = create_dataset(data_number, batch_size=batch_number, repeat_size=repeat_number)

class LinearNet(nn.Cell):
    def __init__(self):
        super(LinearNet, self).__init__()
        self.fc = nn.Dense(2, 1, 0.02, 0.02)

    def construct(self, x):
```

```
            x = self.fc(x)
            return x

start_time = time.time()
net = LinearNet()
model_params = net.trainable_params()
print ('Param Shape is: {}'.format(len(model_params)))
for net_param in net.trainable_params():
    print(net_param, net_param.asnumpy())
netLossBase = nn.loss.MSELoss()

optim = nn.Momentum(net.trainable_params(), learning_rate=0.01, momentum=0.6)
model = Model(net, netLossBase, optim)

epoch = 1
model.train(epoch, ds_train, callbacks=[LossMonitor(10)], dataset_sink_mode=True)

for net_param in net.trainable_params():
    print(net_param, net_param.asnumpy())

print ('The total time cost is: {}s'.format(time.time() - start_time))
```

执行结果：

```
Param Shape is: 2
Parameter (name=fc.weight, shape=(1, 2), dtype=Float32, requires_grad=True) [[0.02 0.02]]
Parameter (name=fc.bias, shape=(1,), dtype=Float32, requires_grad=True) [0.02]
epoch: 1 step: 10, loss is 9.748407
epoch: 1 step: 20, loss is 2.9714396
epoch: 1 step: 30, loss is 3.055818
epoch: 1 step: 40, loss is 1.9764047
epoch: 1 step: 50, loss is 1.9589096
epoch: 1 step: 60, loss is 0.82388747
epoch: 1 step: 70, loss is 0.93128484
epoch: 1 step: 80, loss is 2.1763358
epoch: 1 step: 90, loss is 1.7788371
epoch: 1 step: 100, loss is 1.9664948
epoch: 1 step: 110, loss is 2.601863
epoch: 1 step: 120, loss is 1.2736403
epoch: 1 step: 130, loss is 0.9266836
epoch: 1 step: 140, loss is 3.6779351
epoch: 1 step: 150, loss is 1.9010859
epoch: 1 step: 160, loss is 2.5267096
Parameter (name=fc.weight, shape=(1, 2), dtype=Float32, requires_grad=True) [[1.0694228
0.12706377]]
Parameter (name=fc.bias, shape=(1,), dtype=Float32, requires_grad=True) [5.1867013]
The total time cost is: 1.160539150238037s
```

1.5 自定义数据

MindSpore 可以加载常见的数据集或自定义的数据集，这部分功能在初级教程中进行了部分介绍。加载自定义数据集有两种途径：

- 通过 GeneratorDataset 对象加载。
- 将数据集转换为 MindRecord，即 MindSpore 数据格式，通过读取 MindRecord 文件进行加载数据。

如果用户想要获得更好的性能体验，可以将数据集转换为 MindRecord，从而方便地加载到 MindSpore 中进行训练。本节主要介绍 MindRecord 数据集转换。

1.5.1 自定义数据集转换为 MindRecord

首先，下载需要处理的图片数据 transform.jpg 作为待处理的原始数据。

```
!curl -O https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap2/transform.jpg
```

创建文件夹目录，用于存放所有的转换数据集。

```
!mkdir .\datasets\convert_dataset_to_mindrecord\data_to_mindrecord\
```

创建文件夹目录用于存放下载下来的图片数据。

```
!mkdir .\datasets\convert_dataset_to_mindrecord\images\
```

将下载的图片移至 images 文件夹中：

```
!move .\transform.jpg .\datasets\convert_dataset_to_mindrecord\images\
```

在 Jupyter Notebook 中执行以下命令：

```
#导入文件写入工具类 filewriter
from mindspore.mindrecord import FileWriter
#创建 FileWriter 对象，传入文件名及分片数量，需注意 data_record_path 路径中不能存在中文！
data_record_path = './datasets/convert_dataset_to_mindrecord/data_to_mindrecord/test.mindrecord'
writer = FileWriter(file_name=data_record_path,shard_num=4)
# 定义 schema
data_schema = {"file_name":{"type":"string"},"label":{"type":"int32"},"data":{"type":"bytes"}}
writer.add_schema(data_schema,"test_schema")

# 数据准备
file_name = "./datasets/convert_dataset_to_mindrecord/images/transform.jpg"
with open(file_name, "rb") as f:
    bytes_data = f.read()
data = [{"file_name":"transform.jpg", "label":1, "data":bytes_data}]

indexes = ["file_name","label"]
writer.add_index(indexes)

# 数据写入
writer.write_raw_data(data)
```

```
# 生成本地数据
writer.commit()
```

执行结果：

```
MSRStatus.SUCCESS
```

该示例会生成 8 个文件，成为 MindRecord 数据集。

1.5.2 读取 MindRecord

导入读取类 MindDataset。

```
import mindspore.dataset as ds
```

首先使用 MindDataset 读取 MindRecord 数据集，然后对数据创建字典迭代器，并通过迭代器读取一条数据记录。

```
file_name = './datasets/convert_dataset_to_mindrecord/data_to_mindrecord/test.mindrecord0'
# create MindDataset for reading data
define_data_set = ds.MindDataset(dataset_file=file_name)
# create a dictionary iterator and read a data record through the iterator
count = 0
for item in define_data_set.create_dict_iterator(output_numpy=True):
    print("sample: {}".format(item))
    count += 1
print("Got {} samples".format(count))
```

执行结果：

```
sample: {'data': array([255, 216, 255, ..., 159, 255, 217], dtype=uint8), 'file_name': array(b'transform.jpg', dtype='|S13'), 'label': array(1, dtype=int32)}
Got 1 samples
```

1.6 损失函数

损失函数用于衡量预测值与真实值差异的程度。深度学习中，模型训练就是通过不停地迭代来缩小损失函数值的过程。因此在模型训练过程中损失函数的选择非常重要，定义一个好的损失函数，可以有效提高模型的性能。

MindSpore 提供了许多通用损失函数供用户选择，但这些通用损失函数并不适用于所有场景，有时需要用户自定义所需的损失函数。因此，本文介绍损失函数的自定义构建方法。

1.6.1 自定义损失函数

Cell 是 MindSpore 的基本网络单元，可以用于构建网络，损失函数也需要通过 Cell 来定义。使用 Cell 定义损失函数的方法与定义一个普通的网络相同，差别在于，其执行逻辑用于计算前向网络输出与真实值之间的误差。

以 MindSpore 提供的损失函数 L1Loss 为例，损失函数的定义方法如下：

```
import mindspore.nn as nn
```

```
import mindspore.ops as ops

class L1Loss(nn.Cell):
    def __init__(self):
        super(L1Loss, self).__init__()
        self.abs = ops.Abs()
        self.reduce_mean = ops.ReduceMean()

    def construct(self, predict, target):
        x = self.abs(predict - target)
        return self.reduce_mean(x)
```

在__init__方法中实例化所需的算子，并在 construct 中调用这些算子。这样，一个用于计算 L1Loss 的损失函数就定义好了。

代码中使用 nn.Cell 作为 L1Loss 的基类，最后在 construct 中调用基类提供的 predict, target 方法。

给定一组预测值 predict 和真实值 target，调用损失函数，就可以得到这组预测值和真实值之间的差异，如下所示：

```
import numpy as np
from mindspore import Tensor

loss = L1Loss()
input_data = Tensor(np.array([0.1, 0.2, 0.3]).astype(np.float32))
target_data = Tensor(np.array([0.1, 0.2, 0.2]).astype(np.float32))

output = loss(input_data, target_data)
print(output)
```

以 Ascend 后端为例，输出结果如下：

```
0.033333335
```

在定义损失函数时还可以继承损失函数的基类 LossBase。LossBase 提供了 getLossBase 方法，用于对损失值求和或求均值，输出一个标量。L1Loss 使用 LossBase 作为基类的定义如下：

```
class L1Loss(nn.LossBase):
    def __init__(self, reduction="mean"):
        super(L1Loss, self).__init__(reduction)
        self.abs = ops.Abs()

    def construct(self, base, target):
        x = self.abs(base - target)
        return self.get_loss(x)
```

首先使用 LossBase 作为 L1Loss 的基类，然后给__init__增加一个参数 reduction，并通过 super 传给基类，最后在 construct 中调用基类提供的 get_loss 方法。reduction 的合法参数有 mean、sum 和 none，分别表示求均值、求和与输出原值。

1.6.2 损失函数与模型训练

1.6.2.1 定义数据集与网络

定义训练数据集生成函数，并增强为 MindSpore 可训练的数据类型。

- get_data：数据生成函数。
- create_dataset：将 numpy 数据转换为 MindSpore 可训练的函数，并构造数据集的 batch 增强方法。

```
import numpy as np
from mindspore import dataset as ds

# 生成随机数
def get_data(num, w=2.0, b=3.0):
    for _ in range(num):
        x = np.random.uniform(-10.0, 10.0)
        noise = np.random.normal(0, 1)
        y = x * w + b + noise
        yield np.array([x]).astype(np.float32), np.array([y]).astype(np.float32)

def create_dataset(num_data, batch_size=16):
    dataset = ds.GeneratorDataset(list(get_data(num_data)), column_names=['data', 'label'])
    dataset = dataset.batch(batch_size)
    return dataset
```

定义网络，nn.Dense 将数据集定义为所有的函数。

```
from mindspore.common.initializer import Normal

class LinearNet(nn.Cell):
    def __init__(self):
        super(LinearNet, self).__init__()
        self.fc = nn.Dense(1, 1, Normal(0.02), Normal(0.02))

    def construct(self, x):
        return self.fc(x)
```

1.6.2.2 使用 Model 进行模型训练

Model 是 MindSpore 提供的用于模型训练、评估和推理的高阶 API。创建数据集并定义一个 Model 就可以使用 train 接口进行模型训练。接下来我们使用 Model 进行模型训练，并采用之前定义好的 L1Loss 作为此次训练的损失函数。

使用之前定义的 LinearNet 和 L1Loss 作为前向网络和损失函数，并选择 MindSpore 提供的 Momemtum 作为优化器。

```
net = LinearNet()
loss = L1Loss()
opt = nn.Momentum(net.trainable_params(), learning_rate=0.005, momentum=0.9)
```

定义 Model 时需要指定前向网络、损失函数和优化器，Model 内部会将它们关联起来，组成一张训练网。

```
from mindspore import Model

model = Model(net, loss, opt)
```

创建数据集，并调用 train 接口进行模型训练。

参数解释：

- epoch：训练数据集的迭代次数。
- train_dataset：训练数据集。
- callbacks：model.train 的回调函数参数，也可不使用回调函数功能。
- LossMonitor：损失函数值监视器，用于打印训练过程中的模型损失值。
- dataset_sink_mode（bool）：数据下沉模式，可加快训练。Ascend 和 GPU 平台支持开启该功能（True）。

```
from mindspore.train.callback import LossMonitor

ds_train = create_dataset(num_data=160)
model.train(epoch=1, train_dataset=ds_train, callbacks=[LossMonitor()], dataset_sink_mode=False)
```

执行结果：

```
epoch: 1 step: 1, loss is 9.038219
epoch: 1 step: 2, loss is 8.728367
epoch: 1 step: 3, loss is 10.09524
epoch: 1 step: 4, loss is 8.946897
epoch: 1 step: 5, loss is 9.209949
epoch: 1 step: 6, loss is 8.801371
epoch: 1 step: 7, loss is 6.672551
epoch: 1 step: 8, loss is 6.9106197
epoch: 1 step: 9, loss is 5.992663
epoch: 1 step: 10, loss is 6.5900774
```

1.7 自定义评价指标

评价指标（Metrics）可以用来评估模型结果的可信度。

MindSpore 提供了多种 Metrics 评估指标，如：accuracy、loss、precision、recall、F1 等，完整的 Metrics 功能可参考 API。用户也可根据需求，自行开发并使用 Metrics。

1.7.1 Metrics 自定义方法

通过 Class 实现一个自定义的 Metric 功能，其中包含以下四部分：

- init：初始化，同时进行输入的校验。
- clear：变量初始化。

- update：进行中间过程的计算。
- eval：计算得到最后的输出值。

下面以相似度计算函数 Dice 为例，讲解 Metrics 的开发过程。

步骤 1 导入 Metric 模块

```
import numpy as np
from mindspore._checkparam import Validator as validator
from mindspore.nn.metrics.metric import Metric, rearrange_inputs
```

步骤 2 定义 Metrics

Dice 实际上计算了两个样本间的相似度，数学公式可以表达为：

$$\text{dice} = \frac{2 * (pred \cap true)}{pred + true}$$

Dice 的输入为两个尺度相同的 Tensor，list 或 numpy，一个为预测值，一个为实际值。最后输出两个 Tensor 间的相似度计算值。其中为防止计算过程中分母为零，引入参数 smooth，默认值为 1e-5。

代码实现为：

```
class Dice(Metric):
    def __init__(self, smooth=1e-5):
        """调用 super 进行初始化"""
        super(Dice, self).__init__()
        # 校验 smooth 的数据类型
        self.smooth = validator.check_positive_float(smooth, "smooth")
        # 调用 clear 清空变量
        self.clear()

    def clear(self):
        """清除内部计算结果，变量初始化"""
        self._dice_coeff_sum = 0
        self._samples_num = 0


    def update(self, *inputs):
        """更新内部计算结果"""

        # 校验输入的数量，y_pred 为预测值，y 为实际值
        if len(inputs) != 2:
            raise ValueError('Dice need 2 inputs (y_pred, y), but got {}'.format(len(inputs)))
        # 将输入的数据格式变为 numpy array
        y_pred = self._convert_data(inputs[0])
        y = self._convert_data(inputs[1])
        # 参数计算
        self._samples_num += y.shape[0]
```

```
        # 校验输入的 shape 是否一致
        if y_pred.shape != y.shape:
            raise RuntimeError('y_pred and y should have same the dimension, but the shape of y_pred
is{}, '
                               'the shape of y is {}.'.format(y_pred.shape, y.shape))

        # 根据公式实现 Dice 的过程计算
        intersection = np.dot(y_pred.flatten(), y.flatten())
        unionset = np.dot(y_pred.flatten(), y_pred.flatten()) + np.dot(y.flatten(), y.flatten())

        single_dice_coeff = 2 * float(intersection) / float(unionset + self.smooth)
        self._dice_coeff_sum += single_dice_coeff


    def eval(self):
        """进行 Dice 计算"""
        if self._samples_num == 0:
            raise RuntimeError('Total samples num must not be 0.')

        return self._dice_coeff_sum / float(self._samples_num)
```

步骤 3　在框架中导入 Metrics

在 MindSpore 的 Metris 模块的__init__.py 文件中已经添加定义好的 Dice，具体文件位置以 Anaconda 中虚拟环境 ms 安装的 MindSpore 为例，Metrics 模块位于 F:\Program\Anaconda（F:\Program\Anaconda 为 Anaconda 安装时的根目录）\envs\ms（环境名称）\Lib\site-packages\mindspore\nn\metrics 目录下。

```
__all__ = [
...
    "Dice",
...
]

__factory__ = {
...
    'dice': Dice,
...
}
```

步骤 4　举例验证

```
from mindspore import Tensor

x = Tensor(np.array([[0.2, 0.5], [0.3, 0.1], [0.9, 0.6]]))
y = Tensor(np.array([[0, 1], [1, 0], [0, 1]]))
metric = Dice(smooth=1e-5)
metric.clear()
```

```
metric.update(x, y)
dice = metric.eval()
print(dice)
```

执行结果：

```
0.20467791371802546
```

1.8 自定义训练

MindSpore 提供了 model.train 接口来进行模型训练。此外，还可以使用 TrainOneStepCell，该接口当前支持 GPU、Ascend 环境。

作为高阶接口，model.train 封装了 TrainOneStepCell，可以直接利用设定好的网络、损失函数与优化器进行训练。用户也可以选择使用 TrainOneStepCell 实现更加灵活的训练，例如控制训练数据集、实现多输入多输出网络、或自定义训练过程。

1.8.1 TrainOneStepCell 说明

TrainOneStepCell 中包含三种入参：

- network (Cell)：参与训练的网络，当前仅接受单输出网络。
- optimizer (Cell)：所使用的优化器。
- sens (Number)：反向传播的缩放比例。

下面使用 TrainOneStepCell 替换 model.train，实现简单的线性网络训练过程。

1.8.2 TrainOneStepCell 使用示例

步骤 1　定义网络 LinearNet

```
import numpy as np

from mindspore import Tensor
import mindspore.nn as nn
from mindspore.nn import Cell, Dense
import mindspore.ops as ops
import mindspore.dataset as ds
from mindspore import ParameterTuple

class LinearNet(Cell):
    def __init__(self):
        super().__init__()
        self.relu = nn.ReLU()
        self.dense1 = Dense(5, 32)
        self.dense2 = Dense(32, 1)

    def construct(self, x):
```

```
        x = self.dense1(x)
        x = self.relu(x)
        x = self.dense2(x)
        return x
```

步骤 2　产生输入数据

```
np .random.seed(4)

class DatasetGenerator:
    def __init__(self):
        self.data = np.random.randn(5, 5).astype(np.float32)
        self.label = np.random.randn(5, 1).astype(np.int32)

    def __getitem__(self, index):
        return self.data[index], self.label[index]

    def __len__(self):
        return len(self.data)
```

步骤 3　数据处理

```
#对输入数据进行处理
dataset_generator = DatasetGenerator()
dataset = ds.GeneratorDataset(dataset_generator, ["data", "label"], shuffle=True)
dataset = dataset.batch(32)

# 实例化网络
net = LinearNet()
```

1.8.3 定义 TrainOneStepCell

在 TrainOneStepCell 中，可以实现对训练过程的个性化设定。

```
class TrainOneStepCell(nn.Cell):
    def __init__(self, network, optimizer, sens=1.0):
        """参数初始化"""
        super(TrainOneStepCell, self).__init__(auto_prefix=False)
        self.network = network
        # 使用 tuple 包装 weight
        self.weights = ParameterTuple(network.trainable_params())
        self.optimizer = optimizer
        # 定义梯度函数
        self.grad = ops.GradOperation(get_by_list=True, sens_param=True)
        self.sens = sens

    def construct(self, data, label):
        """构建训练过程"""
        weights = self.weights
```

```
        loss = self.network(data, label)
        # 为反向传播设定系数
        sens = ops.Fill()(ops.DType()(loss), ops.Shape()(loss), self.sens)
        grads = self.grad(self.network, weights)(data, label, sens)
        return loss, self.optimizer(grads)
```

1.8.4 网络训练

在使用 TrainOneStepCell 时，需要利用 WithLossCell 接口引入损失函数，共同完成训练过程。下面利用之前设定好的参数训练 LeNet 网络，并获取 loss 值。

```
# 设定损失函数
crit = nn.MSELoss()
# 设定优化器
opt = nn.Adam(params=net.trainable_params())
# 引入损失函数
net_with_criterion = nn.WithLossCell(net, crit)
# 自定义网络训练
train_net = TrainOneStepCell(net_with_criterion, opt)

# 获取训练过程数据
for d in dataset.create_dict_iterator():
    for i in range(300):
        train_net(d["data"], d["label"])
        print(net_with_criterion(d["data"], d["label"]))
```

执行结果：

```
0.7998974
0.79927444
0.7986423
0.7979911
0.79732
... ...
0.042837422
0.041227795
0.039638687
... ...
9.276913e-06
8.4145695e-06
7.625091e-06
6.904066e-06
6.2513377e-06
```

1.9 实验小结

本实验通过学习自定义算子，掌握正向算子定义和反向算子定义的方法；将数据集转换为 MindRecord 格式，可在 MindSpore 中进行训练，获得更好的性能体验。基于 Loss 作为 L1Loss 的基类，定义损失函数实现简单线性函数拟合。通过学习 Dice 的评估模型掌握自定义评估方法，优化模型。

1.10 思考题

问题：MindSpore 数据格式的目标是什么？基本流程是什么？

答案：MindSpore 数据格式的目标是归一化用户的数据集，并进一步通过 MindDataset（详细使用方法参考 API）实现数据的读取，并用于训练过程。

1.11 缩略语表

API：Application Programming Interface，应用编程接口。

ME：MindExpress，前端表达层。

NN：Neural Network，神经网络。

TBE：Tensor Boost Engine，张量加速引擎。

华为认证 AI 系列教程

HCIP-AI-MindSpore

MindSpore特性与应用

实验指导手册

版本：V1.0

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

华为认证体系介绍

华为认证是华为公司基于“平台+生态”战略，围绕“云-管-端”协同的新ICT技术架构，打造的覆盖**ICT**（Information and Communications Technology 信息通信技术）全技术领域的认证体系，包含**ICT**技术架构与应用认证、云服务与平台认证两类认证。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

华为认证覆盖ICT全领域，符合ICT融合的技术趋势，致力于提供领先的人才培养体系和认证标准，培养数字化时代新型ICT人才，构建良性ICT人才生态。

HCIP-AI-MindSpore Developer V1.0 定位于培养和认证具备使用华为深度学习框架MindSpore进行人工智能开发能力的算法工程师。

通过HCIP-AI-MindSpore Developer V1.0认证，您将掌握MindSpore和MindSpore Lite的架构与特性相关知识，理解MindSpore网络迁移、分布式训练、端云侧推理与部署等相关特性；具备使用MindSpore完成图像处理任务和自然语言处理任务的能力；能够胜任人工智能算法工程师、深度学习工程师、图像算法工程师、自然语言处理算法工程师等岗位。


Huawei Certification
Huawei Certified ICT Expert
Huawei Certified ICT Professional
Huawei Certified ICT Associate
HCIE
HCIP
HCIA
Cloud Service & Platform
云服务与平台
ICT Infrastructure
ICT技术架构与应用
HarmonyOS(To Be)
IoT
openGauss (To Be)
GaussDB
openEuler
Cloud(Cloud Computing + Cloud Service)
Intelligent Driving (To Be)
Big Data
Kunpeng Application Developer
AI
Storage
Datacom
Transmission
Intelligent Computing
WLAN
Access
Enterprise Communication
Security
LTE
Intelligent Vision
SDN
5G
Data Center

前 言

简介

本书为 HCIP-AI-MindSpore Developer V1.0 中文版认证培训教程，适用于准备参加 HCIP-AI-MindSpore Developer 考试的学员或者希望了解动静态图、以及端云侧推理与部署技术的读者。

内容描述

本实验指导书共包含 3 个实验：

- 实验一为动态图实验，通过 MindSpore 在动态图模式下执行单算子、执行普通函数以及调试网络模型帮助学员理解什么是动态图以及动态图模式的优势；
- 实验二为云侧推理部署实验，通过使用 checkpoint 格式文件进行单卡推理和创建 LeNet 模型并进行推理操作帮助学员了解云侧如何进行推理以及其优势；
- 实验三为基于 MindSpore Lite 的推理部署实验，通过 MindSpore 来构建一个猫狗分类模型，然后利用 MindSpore Lite 框架将模型部署到手机上。

读者知识背景

本课程为华为认证高级课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有人工智能知识背景，了解人工智能开发流程；
- 熟悉 python 语言；
- 掌握机器学习算法以及深度学习算法的基础原理；
- 熟悉深度学习框架；

实验环境说明

- 华为云（https://www.huaweicloud.com/）- ModelArts;
- MindSpore1.3.0;

目录

1 MindSpore 特性与应用实验

1.1 动态图实验

1.1.1 实验介绍

1.1.1.1 关于本实验

本实验介绍了 MindSpore 在 PyNative 模式下如何执行单算子、执行普通函数以及调试网络训练模型。

- 任务一：在动态图模式下执行单算子；
- 任务二：在动态图模式下执行普通函数；
- 任务三：在动态图模式下调试网络训练模型。

1.1.1.2 实验目的

- 列举动态图的优势。
- 执行 MindSpore 在动态图模式下的单算子。
- 执行 MindSpore 在动态图模式下的普通函数。
- 训练 MindSpore 在动态图模式下的网络模型。

1.1.1.3 实验介绍

MindSpore 支持两种运行模式，在调试或者运行方面做了不同的优化。

- PyNative 模式：也称动态图模式，将神经网络中的各个算子逐一下发执行，方便用户编写和调试神经网络模型。
- Graph 模式：也称静态图模式或者图模式，将神经网络模型编译成一整张图，然后下发执行。该模式利用图优化等技术提高运行性能，同时有助于规模部署和跨平台运行。

默认情况下，MindSpore 处于 PyNative 模式，可以通过 context.set_context(mode=context.GRAPH_MODE)切换为 Graph 模式；同样地，MindSpore 处于 Graph 模式时，可以通过 context.set_context(mode=context.PYNATIVE_MODE)切换为 PyNative 模式。

PyNative 模式下，支持执行单算子、普通函数和网络，以及单独求梯度的操作。以下是执行过程中的注意事项：

- PyNative 模式下为了提升性能，算子在 device 上使用了异步执行方式，因此在算子执行错误的时候，错误信息可能会在程序执行到最后才显示。

- PyNative 不支持 summary 功能，图模式 summary 相关算子不能使用。

1.1.2 实验任务

1.1.2.1 执行单算子

执行单个算子，并打印相关结果，如下例所示。

输入：

```
import numpy as np
import mindspore.nn as nn
from mindspore import context, Tensor
from mindspore import ParameterTuple
from mindspore.common.initializer import TruncatedNormal
from mindspore.nn import Dense, WithLossCell, SoftmaxCrossEntropyWithLogits, Momentum
from mindspore.common.initializer import Normal
import mindspore.ops as ops
from mindspore import ms_function
from mindspore import dtype as mstype

# 切换为 PyNative 模式
context.set_context(mode=context.PYNATIVE_MODE, device_target="Ascend")

# 打印 Cond2d 算子的输出
conv = nn.Conv2d(3, 4, 3, bias_init='zeros')
input_data = Tensor(np.ones([1, 3, 5, 5]).astype(np.float32))
output = conv(input_data)
print(output.asnumpy())
```

输出：

```
[[[[-0.00423431   0.00647354   0.00647354   0.00647354 -0.01216125]
   [-0.00676727   0.02015686   0.02015686   0.02015686   0.01252747]
   [-0.00676727   0.02015686   0.02015686   0.02015686   0.01252747]
   [-0.00676727   0.02015686   0.02015686   0.02015686   0.01252747]
   [ 0.00930023   0.02043152   0.02043152   0.02043152   0.02119446]]

  [[ 0.00868225 -0.03726196 -0.03726196 -0.03726196 -0.03817749]
   [-0.02784729 -0.06561279 -0.06561279 -0.06561279 -0.03991699]
   [-0.02784729 -0.06561279 -0.06561279 -0.06561279 -0.03991699]
   [-0.02784729 -0.06561279 -0.06561279 -0.06561279 -0.03991699]
   [-0.03338623 -0.05471802 -0.05471802 -0.05471802 -0.0171814 ]]
......
```

1.1.2.2 执行普通函数

将若干算子组合成一个函数，然后直接通过函数调用的方式执行这些算子，并打印相关结果，如下例所示，本例中 add 以普通 PyNative 方法执行。

步骤 1　执行 add 函数

输入：

```
# 定义 add 函数
def add_func(x, y):
    z = ops.add(x, y)
    z = ops.add(z, x)
    return z

x = Tensor(np.ones([3, 3], dtype=np.float32))
y = Tensor(np.ones([3, 3], dtype=np.float32))
output = add_func(x, y)
print(output.asnumpy())
```

输出：

```
[[3. 3. 3.]
 [3. 3. 3.]
 [3. 3. 3.]]
```

步骤 2　提升 PyNative 性能

为了提高 PyNative 模式下的前向计算任务执行速度，MindSpore 提供了 Staging 功能，该功能可以在 PyNative 模式下将 Python 函数或者 Python 类的方法编译成计算图，通过图优化等技术提高运行速度，是一种混合运行机制。Staging 功能的使用通过 ms_function 装饰器达成，该装饰器会将将模块编译成计算图，在给定输入之后，以图的形式下发执行。本例中 add 以 Staging 模式执行：

输入：

```
# 导入 ms_function
from mindspore import ms_function

# 仍设定为 PyNative 模式
context.set_context(mode=context.PYNATIVE_MODE, device_target="Ascend")

add = ops.Add()

# 使用装饰器编译计算图
@ms_function
def add_fn(x, y):
    res = add(x, y)
    return res

x = Tensor(np.ones([4, 4]).astype(np.float32))
y = Tensor(np.ones([4, 4]).astype(np.float32))
z = add_fn(x, y)
print(z.asnumpy())
```

输出：

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

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

在加了 ms_function 装饰器的函数中，如果包含不需要进行参数训练的算子（如 pooling、add 等算子），则这些算子可以在被装饰的函数中直接调用，如上例所示。如果被装饰的函数中包含了需要进行参数训练的算子（如 Convolution、BatchNorm 等算子），则这些算子必须在被装饰等函数之外完成实例化操作。

输入：

```
# Conv2d 实例化操作
conv_obj = nn.Conv2d(in_channels=3, out_channels=4, kernel_size=3, stride=2, padding=0)
conv_obj.init_parameters_data()

@ms_function
def conv_fn(x):
    res = conv_obj(x)
    return res

input_data = np.random.randn(2, 3, 6, 6).astype(np.float32)
z = conv_fn(Tensor(input_data))
print(z.asnumpy())
```

输出：

```
[[[[ 0.05426025  0.06481934 -0.0802002 ]
   [-0.01829529  0.00033069  0.02818298]
   [ 0.00945282 -0.02426147 -0.01374054]]

  [[-0.10272217 -0.00058126  0.05145264]
   [-0.08319092 -0.04943848 -0.01768494]
   [-0.00632858 -0.02482605  0.02029419]]

  [[ 0.04156494 -0.01858521  0.03753662]
   [ 0.00609207  0.01160431  0.05355835]
   [-0.02302551 -0.01756287 -0.00577545]]

  [[-0.07904053 -0.05474854  0.01872253]
   [-0.07104492  0.05877686  0.01779175]
   [-0.00523376 -0.04986572 -0.00030184]]]
```

1.1.2.3 调试网络训练模型

PyNative 模式下，还可以支持单独求梯度的操作。如下例所示，可通过 GradOperation 求该函数或者网络所有的输入梯度。需要注意，输入类型仅支持 Tensor。

输入：

```
def mul(x, y):
```

```
    return x * y

def mainf(x, y):
    return ops.GradOperation(get_all=True)(mul)(x, y)

print(mainf(Tensor(1, mstype.int32), Tensor(2, mstype.int32)))
```

输出：

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

在进行网络训练时，求得梯度然后调用优化器对参数进行优化（暂不支持在反向计算梯度的过程中设置断点），然后再利用前向计算 loss，从而实现在 PyNative 模式下进行网络训练。

下面在 PyNative 模式下进行 LeNet 训练：

步骤 1　构建网络。

输入：

```
context.set_context(mode=context.PYNATIVE_MODE, device_target="Ascend")

class LeNet5(nn.Cell):
    """
    Lenet 网络结构
    """
    def __init__(self, num_class=10, num_channel=1):
        super(LeNet5, self).__init__()
        # 定义所需要的运算
        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 * 5 * 5, 120, weight_init=Normal(0.02))
        self.fc2 = nn.Dense(120, 84, weight_init=Normal(0.02))
        self.fc3 = nn.Dense(84, num_class, weight_init=Normal(0.02))
        self.relu = nn.ReLU()
        self.max_pool2d = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten = nn.Flatten()

    def construct(self, x):
        # 使用定义好的运算构建前向网络
        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

# 实例化网络
net = LeNet5()
```

步骤 2　利用 GradOperation 求函数的输入梯度

输入：

```
class GradWrap(nn.Cell):
    """求函数输入梯度"""
    def __init__(self, network):
        super(GradWrap, self).__init__(auto_prefix=False)
        self.network = network
        # 用 Tuple 的形式包装 weight
        self.weights = ParameterTuple(filter(lambda x: x.requires_grad, network.get_parameters()))

    def construct(self, x, label):
        weights = self.weights
        # 返回值为梯度
        return ops.GradOperation(get_by_list=True)(self.network, weights)(x, label)
```

步骤 3　在 PyNative 模式中进行网络训练。

输入：

```
# 设定优化器、损失函数
optimizer = Momentum(filter(lambda x: x.requires_grad, net.get_parameters()), 0.1, 0.9)
criterion = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
# 通过 WithLossCell 获取 Loss 值
net_with_criterion = WithLossCell(net, criterion)
# 调用 GradWrap
train_network = GradWrap(net_with_criterion)
train_network.set_train()

# 产生输入数据
input_data = Tensor(np.ones([32, 1, 32, 32]).astype(np.float32) * 0.01)
label = Tensor(np.ones([32]).astype(np.int32))
output = net(Tensor(input_data))

# 利用前向网络计算 loss
loss_output = criterion(output, label)
# 求得梯度
grads = train_network(input_data, label)
# 优化参数
success = optimizer(grads)
```

```
loss = loss_output.asnumpy()
print(loss)
```

输出：

```
2.3025854
```

1.1.3 实验小结

本实验主要使用 MindSpore 在动态图（PyNative）模式下进行单算子的执行、普通函数的执行以及调试网络训练模型。

1.1.4 思考题

MindSpore 有怎样的功能可以将 python 函数或者 python 类的方法编译成计算图?

答案：为了提高 PyNative 模式下的前向计算任务执行速度，MindSpore 提供了 Staging 功能，该功能可以在 PyNative 模式下将 Python 函数或者 Python 类的方法编译成计算图，通过图优化等技术提高运行速度，是一种混合运行机制。Staging 功能的使用通过 ms_function 装饰器达成，该装饰器会将模块编译成计算图，在给定输入之后，以图的形式下发执行。

1.2 云侧推理-推理部署

1.2.1 实验介绍

1.2.1.1 关于本实验

MindSpore 可以基于训练好的模型，在不同的硬件平台上执行推理任务。本文将介绍如何在 Ascend910 硬件环境中，利用 Checkpoint 执行推理，Checkpoint 是训练参数，采用了

Protocol Buffers 格式，一般用于训练任务中断后恢复训练，或训练后的微调（Fine Tune）任务。

1.2.1.2 实验目的

- 理解 MindSpore 如何在 Ascend910 硬件环境中执行推理任务。
- 理解如何使用 checkpoint 格式文件单卡推理。

1.2.1.3 实验介绍

MindSpore 支持保存两种类型的数据：训练参数和网络模型（模型中包含参数信息）。

- 训练参数指的是 Checkpoint 格式文件。
- 网络模型包括 MindIR、AIR 和 ONNX 三种格式文件。

Checkpoint 采用了 Protocol Buffers 格式，存储了网络中所有的参数值。一般用于训练任务中断后恢复训练，或训练后的微调（Fine Tune）任务，其他三种格式包括 MindIR、AIR 和 ONNX 三种格式文件一般用于跨硬件平台执行推理任务，本实验主要在华为云上通过加载网络训练产生的 Checkpoint 文件，调用 model.predict 接口进行推理验证。

1.2.2 实验任务

1.2.2.1 使用 checkpoint 格式文件单卡推理

使用本地模型推理，用户可以通过 load_checkpoint 和 load_param_into_net 接口从本地加载模型与参数，传入验证数据集后使用 model.eval 即可进行模型验证，使用 model.predict 可进行模型推理。在这里我们下载 MindSpore Hub 中已经预训练好的 LeNet 和 MINIST 数据集进行推理演示：

步骤 1　准备数据

```
!mkdir -p ./datasets/MNIST_Data/test
!wget -NP ./datasets/MNIST_Data/test https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap3/t10k-labels-idx1-ubyte
!wget -NP ./datasets/MNIST_Data/test https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap3/t10k-images-idx3-ubyte

!mkdir -p ./checkpoint
!wget -NP ./checkpoint https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap3/lenet_ascend_v111_offical_cv_mnist_bs32_acc98.ckpt --no-check-certificate
```

步骤 2　配置运行所需信息，进行推理的数据处理。

输入：

```
import os
import argparse
from mindspore import context
```

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

parser = argparse.ArgumentParser(description='MindSpore Inference Example')
parser.add_argument('--device_target', type=str, default="GPU", choices=['Ascend', 'GPU'])

args = parser.parse_known_args()[0]
context.set_context(mode=context.GRAPH_MODE, device_target=args.device_target)

def create_dataset(data_path, batch_size=32, repeat_size=1,
                           num_parallel_workers=1):
    # 定义数据集
    mnist_ds = ds.MnistDataset(data_path)
    resize_height, resize_width = 32, 32
    rescale = 1.0 / 255.0
    shift = 0.0
    rescale_nml = 1 / 0.3081
    shift_nml = -1 * 0.1307 / 0.3081

    # 定义所需要操作的 map 映射
    resize_op = CV.Resize((resize_height, resize_width), interpolation=Inter.LINEAR)
    rescale_nml_op = CV.Rescale(rescale_nml, shift_nml)
    rescale_op = CV.Rescale(rescale, shift)
    hwc2chw_op = CV.HWC2CHW()
    type_cast_op = C.TypeCast(mstype.int32)

    # 使用 map 映射函数，将数据操作应用到数据集
    mnist_ds = mnist_ds.map(operations=type_cast_op, input_columns="label",
num_parallel_workers=num_parallel_workers)
    mnist_ds = mnist_ds.map(operations=resize_op, input_columns="image",
num_parallel_workers=num_parallel_workers)
    mnist_ds = mnist_ds.map(operations=rescale_op, input_columns="image",
num_parallel_workers=num_parallel_workers)
    mnist_ds = mnist_ds.map(operations=rescale_nml_op, input_columns="image",
num_parallel_workers=num_parallel_workers)
    mnist_ds = mnist_ds.map(operations=hwc2chw_op, input_columns="image",
num_parallel_workers=num_parallel_workers)

    # 进行 shuffle、batch 操作
    buffer_size = 10000
    mnist_ds = mnist_ds.shuffle(buffer_size=buffer_size)
    mnist_ds = mnist_ds.batch(batch_size, drop_remainder=True)

    return mnist_ds
```

1.2.2.2 创建 LeNet 模型

输入：

```
import mindspore.nn as nn
from mindspore.common.initializer import Normal

class LeNet5(nn.Cell):
    """
    Lenet 网络结构
    """
    def __init__(self, num_class=10, num_channel=1):
        super(LeNet5, self).__init__()
        # 定义所需要的运算
        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 * 5 * 5, 120, weight_init=Normal(0.02))
        self.fc2 = nn.Dense(120, 84, weight_init=Normal(0.02))
        self.fc3 = nn.Dense(84, num_class, weight_init=Normal(0.02))
        self.relu = nn.ReLU()
        self.max_pool2d = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten = nn.Flatten()

    def construct(self, x):
        # 使用定义好的运算构建前向网络
        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

# 实例化网络
net = LeNet5()
```

步骤 1　从本地加载模型与参数。

在推理进行前，需要使用 load_checkpoint 和 load_param_into_net 接口从本地加载模型与参数，这样一来就可以使用本地模型完成后面的推理过程。

输入：

```
from mindspore import load_checkpoint, load_param_into_net
```

```
ckpt_file_name = "./checkpoint/lenet_ascend_v111_offical_cv_mnist_bs32_acc98.ckpt"
param_dict = load_checkpoint(ckpt_file_name)
load_param_into_net(net, param_dict)
```

步骤 2　设置损失函数与优化器。

设置损失函数与优化器，并调用 model 接口创建对象：

```
import numpy as np
from mindspore.nn import Accuracy
from mindspore import Model, Tensor

net_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")
net_opt = nn.Momentum(net.trainable_params(), learning_rate=0.01, momentum=0.9)
model = Model(net, net_loss, net_opt, metrics={"Accuracy": Accuracy()})
```

下面调用 model.eval 接口执行验证过程：

输入：

```
mnist_path = "./datasets/MNIST_Data/test"
ds_eval = create_dataset(os.path.join(mnist_path))
acc = model.eval(ds_eval, dataset_sink_mode=False)
print("============== {} ==============".format(acc))
```

输出：

```
============== {'Accuracy': 0.9847756410256411} ==============
```

调用 model.predict 接口执行验证过程，这里选取数据集中的一张图片进行预测：

输入：

```
ds= ds_eval.create_dict_iterator()
data = next(ds)

# images 为测试图片，labels 为测试图片的实际分类
images = data["image"].asnumpy()
labels =data["label"].asnumpy()

#  使用函数 model.predict 预测 image 对应分类
output = model.predict(Tensor(data['image']))
predicted = np.argmax(output.asnumpy(), axis=1)

#  输出预测分类与实际分类
print(f'Predicted: "{predicted[0]}", Actual: "{labels[0]}"')
```

输出：

```
Predicted: "3", Actual: "3"
```

1.2.3 实验小结

本实验主要使用 MindSpore 在 Ascend 910 AI 处理器上推理，通过 checkpoint 格式文件进行单卡推理。

1.2.4 思考题

MindSpore 可以基于训练好的模型，在不同的硬件平台上执行推理任务，MindSpore 支持哪两种类型的数据?

答案：训练参数和网络模型（模型中包含参数信息）。

1.3 基于 MindSpore Lite 的猫狗分类实验

1.3.1 实验介绍

在深度学习算法飞速发展的今天，越来越多的研究者希望复杂的神经网络模型可以应用于更多的场景，在更多的终端应用以实现智慧生活，本实验通过 MindSpore 来构建一个猫狗分类模型，然后利用 mindspore lite 框架将模型部署到手机上。

【实验环境要求】:

- Windows
- MindSpore 1.3.0
- MindSpore Lite converter 1.3.0
- 参考《HCIP-AI-MindSpore V1.0 实验环境搭建指南》完成 MindSpore 的安装。

1.3.2 实验目的

- 了解如何使用 MindSpore 对 MobileNet V2 模型进行微调。
- 掌握 MindSpore Lite 模型推理的流程。

1.3.3 实验设计

1.3.3.1 实验总体设计

本实验主要分为两部分，其中第一部分是使用 MindSpore（CPU）通过 Fine-Tuning 训练一个猫狗分类模型，并部署到手机端；第二部分是解析 APP 构建代码，学习模型部署的执行逻辑。

1.3.3.2 MobileNetV2 介绍

MobileNetV2 是专门为移动和嵌入式设备设计的网络架构，该架构能在保持类似精度的条件下显著减少模型参数和计算量，因为本实验最后需要把模型部署到手机上，所以选择了这个网络，更多关于 MobileNetV2 的介绍可以参考原始论文：https://arxiv.org/abs/1801.04381。

1.3.3.3 MindSpore Lite 介绍

2020 年 3 月，MindSpore 全场景 AI 计算框架对外开源，作为端、边、云全场景的一部分，端侧领域 MindSpore Lite 在 2020 年 9 月正式发布并对外开源。

MindSpore Lite 支持 CPU/GPU/NPU，并且支持 IOS、Android 以及 LiteOS 等嵌入式操作系统，在模型方面支持 MindSpore/TensorFlow Lite/Caffe/Onnx 模型，极致性能、轻量化、全场景支持且高效部署。

1.3.4 实验详细设计与实现

本实验是一个完整的从数据获取、数据集清洗到端侧部署的全流程实验。整体实验流程为下图所示：

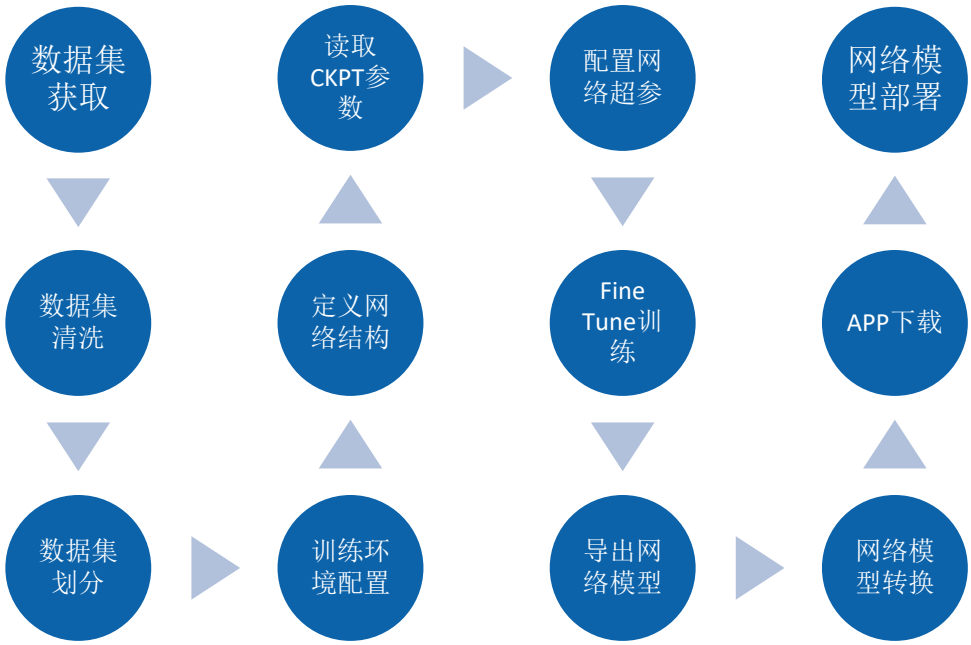

图1-1 实验流程图

1.3.5 环境准备及资源下载

深度学习计算中，从头开始训练一个视觉任务络耗时巨大，需要大量计算能力。预训练模型选择的常见的 OpenImage、ImageNet、VOC、COCO 等公开大型数据集，规模达到几十万甚至超过上百万张。大部分任务数据规模较大，训练网络模型时，如果不使用预训练模型，从头开始训练网络，需要消耗大量的时间与计算能力，模型容易陷入局部极小值和过拟合。因此大部分任务都会选择预训练模型，在其上做微调（也称为 Fine Tune）。

1.3.5.1.1 环境准备

实验开始前，请务必在本机安装好以下环境：

- python3.7.5
- MindSpore 1.3.0
- Python IDE（Pycharm/Spyder/Jupyter Notebook）

安装操作请参考《HCIP-AI-MindSpore V1.0 实验环境搭建指南》完成实验环境搭建。

1.3.5.1.2 资源下载

本实验中需要下载的内容为：1.数据集；2. MindSpore Lite convert 转换工具。数据集主要用于 FineTune 训练，转换工具负责将 mindspore 框架训练的模型转换为 ms 格式从而部署在手机端。

数据集和转换工具下载链接：https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap3/MindSporePetClassification.zip

在 MindSporePetClassification 中，确保路径中无中文字符。数据集解压后的结构如下：

```
PetImages
    ├──Cat
    ├──Dog
```

工程文件夹在 7.4.1.2 下载的实验资料中已经提供，文件目录如下图：

```
MindSporePetClassification
        ├──ADB                  //支持手机与电脑传递文件工具
        |
        ├──code                 //Fine tune训练代码
        |   |
        |   ├──mobilenetV2.ckpt //模型文件
        |   |
        |   ├──preprocessing_dataset.py //预先处理数据集脚本
        |   |
        |   ├──requirements.txt //需要的依赖软件包
        |   |
        |   ├──src              //训练脚本调用的配置、数据处理等脚本
        |   |
        |   └──train.py         //主训练脚本
        |
        └──converter            //转换工具MindSpore Lite Converter
```

图1-2 工程文件结构

确保数据集压缩文件 kagglecatsanddogs_3367a.zip 放在 MindSporePetClassification 文件夹的下一级目录中。

1.3.5.2 数据清洗及数据集划分

切换至 MindSporePetClassification 文件夹下的 code 目录中，在地址栏输入：cmd 激活命令行窗口。

图1-3 命令行窗口

输入以下命令，运行数据清洗脚本。

```
python preprocessing_dataset.py D:\MindSporePetClassification\kagglecatsanddogs_3367a.zip
```

其中 D:\MindSporePetClassification\kagglecatsanddogs_3367a.zip 是数据集文件所在位置。

1.3.5.2.2 代码解析

初始数据集往往存在很多的“脏数据”，这部分样本的存在会对模型产生不利的影响，如：影响准确率。所以，在实际工程中，往往第一步会进行数据清洗，本实验中，preprocessing_dataset.py 文件主要实现数据集清洗并划分数据集的功能。代码如下：

```
#导入实验所需模块
import os
import sys
import zipfile
from PIL import Image
import shutil
#设定数据集文件夹名称及需要的图片格式
_DATASET_NAME = "PetImages"
IMG_EXTENSIONS = ['.jpg', '.jpeg', '.png', '.ppm', '.bmp', '.tif', '.tiff']
#判断该文件是否为图片
def is_image_file(filename):
    """Judge whether it is a picture."""
    return any(filename.lower().endswith(extension) for extension in IMG_EXTENSIONS)
#删除图片或文件夹
def delete_file(img_path):
    if os.path.isdir(img_path):
        os.rmdir(img_path)
    else:
        os.remove(img_path)
        print(f"Invalid image file, delete {img_path}")
#判断是否为 JPG 格式图片
def is_jpg(img_path):
    try:
        i=Image.open(img_path)
        return i.format =='JPEG'
    except IOError:
        return False
#过滤数据集文件夹下的所有图片，如果是不需要的格式，则删除
def filter_dataset(dataset_path):
    for sub_dir in os.listdir(dataset_path):
        for file_name in os.listdir(os.path.join(dataset_path, sub_dir)):
            img_path = os.path.join(os.path.join(dataset_path, sub_dir, file_name))
            if not (is_image_file(file_name) and is_jpg(img_path)):
```

```
                    delete_file(img_path)
                    continue
#分割数据集，将上一步已经清洗干净的数据集进行分割，分割比例为训练集：测试集 9:1
def split_dataset(dataset_path, eval_split=0.1):
    os.makedirs(os.path.join(dataset_path, "train"))
    os.makedirs(os.path.join(dataset_path, "eval"))
    for sub_dir in os.listdir(dataset_path):
        if sub_dir in ["train", "eval"]:
            continue
        cls_list = os.listdir(os.path.join(dataset_path, sub_dir))
        train_size = int(len(cls_list) * (1 - eval_split))
        os.makedirs(os.path.join(dataset_path, "train", sub_dir))
        os.makedirs(os.path.join(dataset_path, "eval", sub_dir))
        for i, file_name in enumerate(os.listdir(os.path.join(dataset_path, sub_dir))):
            source_file = os.path.join(dataset_path, sub_dir, file_name)
            if i <= train_size:
                target_file = os.path.join(dataset_path, "train", sub_dir, file_name)
            else:
                target_file = os.path.join(dataset_path, "eval", sub_dir, file_name)
            shutil.move(source_file, target_file)
        delete_file(os.path.join(dataset_path, sub_dir))
#解压数据集，将 zip 文件解压
def extract_dataset(zip_file, save_dir):
    if not os.path.exists(zip_file):
        ValueError(f"{zip_file} is not a valid zip file!")
    try:
        print("extract dataset")
        zip_file = zipfile.ZipFile(zip_file)
        for names in zip_file.namelist():
            zip_file.extract(names, save_dir)
        zip_file.close()
        print(f"extract dataset at {os.path.join(save_dir, _DATASET_NAME)}")
    except:
        ValueError(f"{save_path} is not a valid zip file!")
#main 函数执行时，首先解压数据集，然后遍历解压后文件夹获取所有样本，进行数据清洗，然后分割为训练集和测试集，分别保存在对应的文件夹中
if __name__ == '__main__':
    save_dir = os.path.abspath("./dataset")
    if not os.path.isdir(save_dir):
        os.makedirs(save_dir)
    zip_file = sys.argv[1]
    extract_dataset(zip_file, save_dir)
    print("filter invaild images!")
    dataset_path = os.path.join(save_dir, _DATASET_NAME)
    filter_dataset(dataset_path)
    print("filter invaild images done, then split dataset to train and eval")
    split_dataset(dataset_path, eval_split=0.1)
```

```
    print(f"final dataset at {dataset_path}")
```

其中，filter_dataset 函数实现了数据集清洗的作用，split_dataset 函数将数据集进行了切分，分为训练集和测试集，本实验中以 9:1 的比例进行了划分，这个比例作为超参数可以更改。

1.3.5.3 网络模型搭建及训练

获取到干净的数据集后，下一步就是进行网络模型的搭建并训练，在 code 文件夹下执行以下命令。

```
python train.py
```

如果本地有多个 python 环境，请切换到安装有 MindSpore 的环境中。

训练过程如下：

脚本会自动安装所需的依赖软件包：'opencv-python' 和 'matplotlib'，用于图形处理，我们后面会使用到，安装完成如下图所示，整个过程视网络速度，一般在 1~2 分钟。

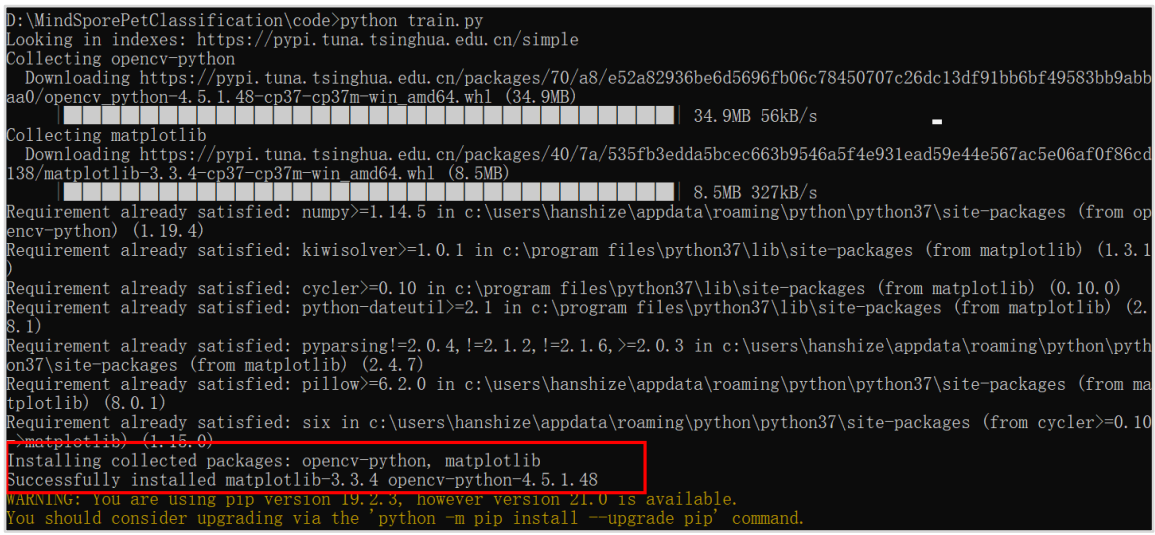

图1-4 命令行窗口

脚本会在正式训练前从数据集中抽取 6 张图片载入当前模型文件，对猫狗进行判别，如下图所示，部分猫狗分类明显不符合预期，点击图片右上角关闭图片，使脚本正式进入 Fine tune 训练。

图1-5 预训练模型效果

训练时会对模型的精度进行测试，当训练的模型精度超过当前模型精度时，就会将高精度模型保存下来，如下图所示。

```
training feature cached [22272/22272] 100%
train feature cache finished!
evaluation feature cached [2476/2476] 100%
eval feature cache finished!                                                    开始进入迭代训练
saved feature features folder
epoch[1/30], iter[696] cost: 2047.780, per step time: 2.942, avg loss: 0.364, acc: 0.986, train acc: 0.988
update best acc: 0.9866997360484405
epoch[2/30], iter[696] cost: 1710.441, 不断替换保存高精度模型 loss: 0.354, acc: 0.988, train acc: 0.988
update best acc: 0.9880012390984098
epoch[3/30], iter[696] cost: 1729.818, per step time: 2.485, avg loss: 0.352, acc: 0.991, train acc: 0.990
update best acc: 0.9906043902691896
epoch[4/30], iter[696] cost: 1743.477, per step time: 2.505, avg loss: 0.351, acc: 0.989, train acc: 0.991
epoch[5/30], iter[696] cost: 1748.227, per step time: 2.512, avg loss: 0.351, acc: 0.989, train acc: 0.991
epoch[6/30], iter[696] cost: 1762.436, per step time: 2.532, avg loss: 0.350, acc: 0.987, train acc: 0.990
```

图1-6 命令行窗口

训练完成，脚本会再次调用之前的图片对 Fine tune 后的模型进行验证，如下图所示，所有图片分类符合预期，说明模型精度满足要求，点击图片右上角关闭图片，使脚本继续运行完成文件转换。

图1-7 Fine Tune 后效果

训练后将 Fine tune 后的模型文件转换为 mindir 文件，并在 code 文件夹下生成 mobilenetv2.mindir 文件。

```
epoch[4/30], iter[696] cost: 1743.477, per step time: 2.505, avg loss: 0.351, acc: 0.989, train acc: 0.991
epoch[5/30], iter[696] cost: 1748.227, per step time: 2.512, avg loss: 0.351, acc: 0.989, train acc: 0.991
epoch[6/30], iter[696] cost: 1762.436, per step time: 2.532, avg loss: 0.350, acc: 0.987, train acc: 0.990
epoch[7/30], iter[696] cost: 1773.321, per step time: 2.548, avg loss: 0.350, acc: 0.989, train acc: 0.992
epoch[8/30], iter[696] cost: 1746.750, per step time: 2.510, avg loss: 0.350, acc: 0.展示训练信息与训练完成
epoch[9/30], iter[696] cost: 1805.162, per step time: 2.594, avg loss: 0.350, acc: 0.988, train acc: 0.992
early stop! the best epoch is 2
train total cost 1093.3910 s
export mobilenetv2 MINDIR file at d:\MindSporePetClassification\code\mobilenetv2.mindir
```

图1-8 命令行窗口

1.3.5.3.2 代码解析

步骤 1 训练环境配置

利用 mindspore 框架进行模型训练的时候需要先导入所需的依赖模块，并配置训练环境。

```
import time

from mindspore import Tensor, nn
from mindspore.nn.optim.momentum import Momentum
from mindspore.nn.loss import SoftmaxCrossEntropyWithLogits
from mindspore.common import set_seed

from src.dataset import extract_features
from src.lr_generator import get_lr
from src.config import set_config
from src.args import train_parse_args
from src.utils import context_device_init, export_mindir, predict_from_net,
get_samples_from_eval_dataset
from src.models import CrossEntropyWithLabelSmooth, define_net, load_ckpt, get_networks, train
set_seed(1)
```

```
if __name__ == '__main__':
    #设定训练的相关参数，主要确定训练所在的硬件平台以及是否训练
    args_opt = train_parse_args()
    #确定在 CPU 进行训练后，设定训练网络的相关超参数，如：图像归一化尺寸，batchsize 大小等
    config = set_config(args_opt)
    start = time.time()

    #初始化环境设置
    context_device_init(config)
```

其中，context_device_init 函数初始化了训练环境，具体可参考 src 文件夹中的源码。

步骤 2 定义网络结构

一般一个分类网络包含两个部分：backbone 和 head，其中 backbone 部分通常是一系列卷积层，负责提取图片特征；head 部分通常是一组全连接层，用于分类，一般最后一个全连接层的输出对应使用数据集的分类个数。 同数据集和任务中特征提取层（卷积层）分布趋于一致，但是特征向量的组合（全连接层）不相同，分类数量（全连接层 output_size）通常也不一致。这里我们选择冻结 backbone 部分的参数， 只训练 head 的参数进行微调。

```
# 定义网络结构
backbone_net, head_net, net = define_net(config, activation="Softmax")
```

具体可参考 src 文件夹中的源码。

步骤 3 读取 CKPT 参数

Backbone 部分的网络参数往往存储于 ckpt 文件中，这里我们首先要读取相关参数，再进行训练。

```
# 从 ckpt 文件中获取 mobilenet 的 backbone 参数
load_ckpt(backbone_net, args_opt.pretrain_ckpt, trainable=False)
```

具体可参考 src 文件夹中的源码。

步骤 4 配置网络超参

网络的超参数是训练网络时非常关键的信息，优秀的超参数可以提高网络的训练效率和性能。

```
# 读取图片并展示只使用 backbone 预测的图片结果
test_list = get_samples_from_eval_dataset(args_opt.dataset_path)
predict_from_net(net, test_list, config, show_title="pre training")

# catch backbone features
data, step_size = extract_features(backbone_net, args_opt.dataset_path, config)

    # 定义损失函数
if config.label_smooth > 0:
    loss = CrossEntropyWithLabelSmooth(
        smooth_factor=config.label_smooth, num_classes=config.num_classes)
else:
```

```
    loss = SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')

# 获取模型的学习率，在不同的 epoch 有不同的学习率
lr = Tensor(get_lr(global_step=0,
                   lr_init=config.lr_init,
                   lr_end=config.lr_end,
                   lr_max=config.lr_max,
                   warmup_epochs=config.warmup_epochs,
                   total_epochs=config.epoch_size,
                   steps_per_epoch=step_size))

# 定义优化器
opt = Momentum(filter(lambda x: x.requires_grad, head_net.get_parameters()), lr, config.momentum,
config.weight_decay)
```

代码中，首先在读取了 backbone 的参数之后就直接进行了预测，并展示了结果，从结果中可以发现预测精度不足，然后又定义了优化器、学习率、损失函数。具体可参考 src 文件夹中的源码。

步骤 5 训练

利用训练集进行网络训练，并展示训练后的效果。

```
# 定义训练网络并开始训练
train_net, eval_net = get_networks(head_net, loss, opt)
train(train_net, eval_net, net, data, config)
print("train total cost {:5.4f} s".format(time.time() - start))

# 展示 Fine Tune 之后的预测结果。
predict_from_net(net, test_list, config, show_title="after training")
```

在训练过后，原数据集文件中会产生几个 npy 文件，该文件主要保存训练过程中的相关变量与标签。

步骤 6 导出网络模型

导出 MindIR 模型。

```
# 导出 mindir 模型
export_mindir(net, "mobilenetv2")
```

至此，网络完成了本地训练并成功获取了 MindIR 模型，该模型将用于下一步的模型转换操作。

1.3.5.4 模型转换

本地训练的网络模型在应用到端侧或者边缘侧设备时，往往因为依赖库，硬件环境以及软件环境的不同需要进行模型转换以适配部署端的环境。在本地训练完，我们获得的模型文件是 MindIR 格式的文件，在端侧部署时，我们需要转换为 ms 格式的文件。MindSpore Lite 提供了完整的转换工具来帮助开发者完成模型转换，在 7.4.1.2 节中下载的课程源码包中已经包括

了转换工具，放在 converter 文件夹下，如果需要最新的转换工具，请参考 7.4.1.2 节的链接下载。

准备好 converter 转换工具之后，进入 converter 文件夹中，在地址栏输入 cmd 启动命令行窗口，输入以下代码进行模型转换。

```
call converter_lite --fmk=MINDIR --modelFile=d:\MindSporePetClassification\code\mobilenetv2.mindir --outputFile=pet
```

其中，fmk 参数代表待转换文件类型，modelFile 代表模型绝对路径（如果模型文件也位于 converter 文件夹中，则可以直接写文件名），outputFile 代表模型输出名称。

转换成功，如下图所示会输出“CONVERT RESULT SUCCESS:0”。

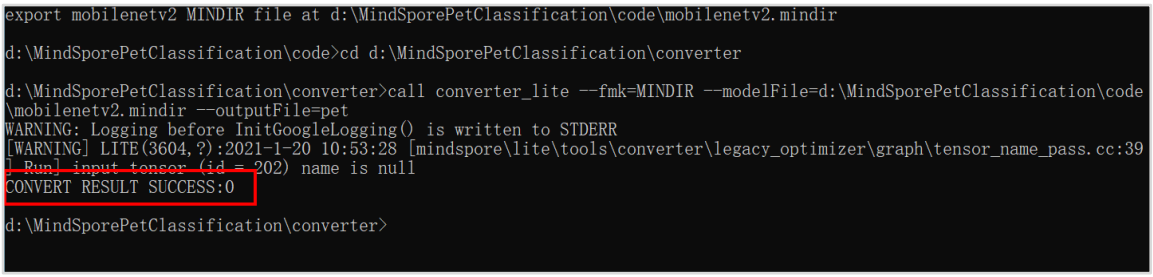

图1-9 命令行窗口

其中 WRANING 信息不影响使用，大家不用关注，成功后，会在 converter 文件夹中生成 pet.ms 文件。

1.3.5.5 模型部署

经过模型转换后，我们获得了 ms 文件，该文件可以在端侧被调用，接下来我们将在端侧构建一个 APP 来调用 ms 模型，实现猫狗分类。

步骤 1 APP 下载

本实验已准备好成熟的 APP 安装文件，通过下图的二维码直接下载安装。

图1-10 APP 下载码

步骤 2 APP 功能预览

打开 APP 选择 Choose a Demo 中任何一个 Demo 图片，如下图所示，由于目前应用还未导入训练好的模型，识别结果是不准确的，需导入训练好的模型才能准确识别猫狗。

图1-11 APP 效果图

注意：启动 APP 后会在手机中自动生成“PetClassification”文件夹，可在文件管理类应用中查看。

步骤 3 将模型传输至手机

将本机上的 ms 模型传送到手机上，这一步操作的方式比较灵活，这里不提供传输形式。

建议使用：USB 线连接传输、邮件传输、第三方软件传输等。

步骤 4 移动 ms 文件至 PetClassification 文件下

将 pet.ms 文件移动至 PetClassification 文件夹下，这一步需要使用手机中的文件管理工具，因手机型号各异，这里仅提供华为手机操作的流程图，仅供参考。

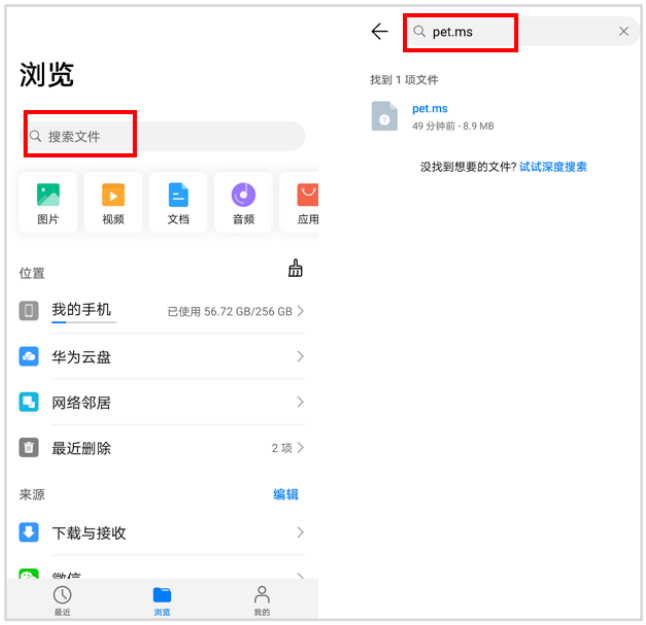

图1-12 手机搜索 ms 文件

长按 pet.ms 文件，选择“移动”=>“我的手机”=>“PetClassification”，点击右上角确定，即完成导入，如下图所示。

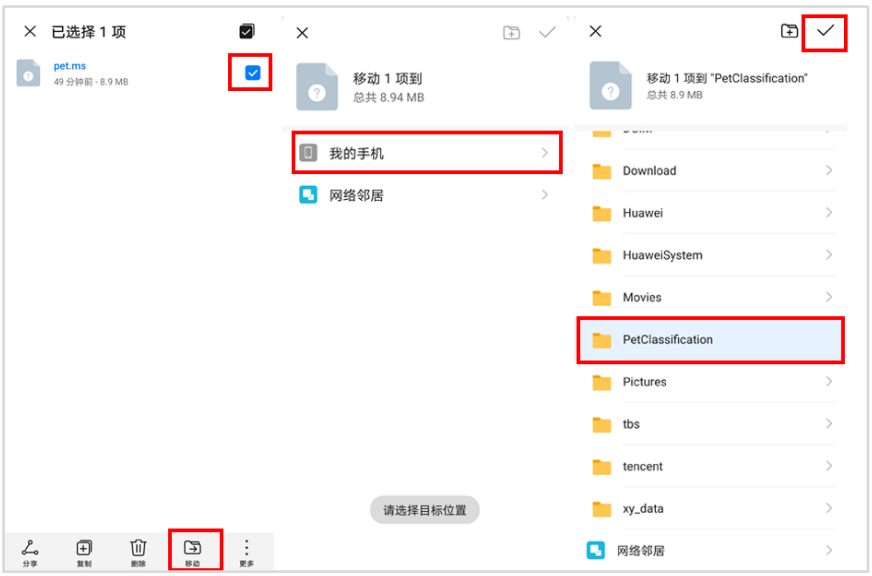

图1-13 移动 ms 模型文件

步骤 5 识别效果验证

将 ms 文件移动好之后，在手机端打开 APP-PetClassification，就可以体验猫狗识别了，如下图所示。

图1-14 APP 界面

APP 支持如下功能：

1. 图片：选择相册中的猫狗照片识别；
2. 相机：打开相机拍照识别；
3. 预览：打开摄像头扫描识别：
4. 选择一个案例：选择以下任意一张猫狗图片识别。

1.3.5.6 推理侧代码解析

在 7.4.7 节中我们通过二维码下载了 APP 以便使用。该 APP 中包含了推理侧的逻辑代码，这一节，我们就详细的解释推理侧的代码，以便老师自己进行开发。

MindSpore Lite 是用于端侧设备的框架，在推理时需要准备模型、前处理代码和后处理代码以及用于推理的 MindSpore Lite 运行库。

步骤 1　工程文件说明

工程文件在 app 文件夹下面，外层是一些通用依赖项，可以不管，主要看 app 文件夹下的内容，目录结构如下：

```
app
├── src/main
│   ├── assets # 资源文件
│   │   └── model # 模型文件
│   │       └── mobilenetv2.ms # 存放的模型文件
│   │
│   ├── cpp # 模型加载和预测主要逻辑封装类
│   │   ├── ..
│   │   ├── mindspore-lite-1.1.0-inference-android # MindSpore Lite版本
│   │   ├── MindSporeNetnative.cpp # MindSpore调用相关的JNI方法
│   │   └── MindSporeNetnative.h # 头文件
│   │   └── MsNetWork.cpp # MindSpore接口封装
│   │
│   ├── java # java层应用代码
│   │   └── com.mindspore.classification
│   │       ├── gallery.classify # 图像处理及MindSpore JNI调用相关实现
│   │       │   └── ...
│   │       └── widget # 开启摄像头及绘制相关实现
│   │           └── ...
│   │
│   ├── res # 存放Android相关的资源文件
│   └── AndroidManifest.xml # Android配置文件
│
├── CMakeList.txt # cmake编译入口文件
│
├── build.gradle # 其他Android配置文件
├── download.gradle # 工程依赖文件下载
└── ...
```

图1-15 示例程序结构

在本示例中，build 过程由 app/download.gradle 文件自动下载 MindSpore Lite 版本文件和工程所需模型，并放置在 app/src/main/cpp/目录下，下面部分会挑一些重要代码说明，完整代码见工程文件。

在 app/CMakeLists.txt 文件中建立.so 库文件链接，如下所示。

```
cmake_minimum_required(VERSION 3.14.0)
project(MindSpore)

if(CMAKE_CXX_COMPILER_ID STREQUAL "GNU" AND CMAKE_CXX_COMPILER_VERSION VERSION_LESS 7.3.0)
    message(FATAL_ERROR "GCC version ${CMAKE_CXX_COMPILER_VERSION} must not be less than 7.3.0")
endif()

include(${CMAKE_SOURCE_DIR}/cmake/options.cmake)
include(${CMAKE_SOURCE_DIR}/cmake/check_requirements.cmake)
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/cmake/modules/")
```

```
if(NOT CMAKE_SYSTEM_NAME MATCHES "Windows")
    if(NOT ENABLE_GLIBCXX)
        add_compile_definitions(_GLIBCXX_USE_CXX11_ABI=0)
    endif()
endif()

if(${CMAKE_SYSTEM_NAME} MATCHES "Darwin")
    set(CMAKE_OSX_SYSROOT "")
    set(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O2 -Winconsistent-missing-override -Wuser-defined-warnings \
        -Wno-return-std-move -Wno-unused-private-field -Wno-unused-lambda-capture -Wno-sign-compare \
        -Wno-overloaded-virtual -Wno-unneeded-internal-declaration -Wno-unused-variable -Wno-pessimizing-move \
        -Wno-inconsistent-missing-override -DHALF_ENABLE_CPP11_USER_LITERALS=0 -D_FORTIFY_SOURCE=2")
else()
    set(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O2 -Wl,--allow-shlib-undefined \
        -DHALF_ENABLE_CPP11_USER_LITERALS=0 -D_FORTIFY_SOURCE=2")
endif()

if(ENABLE_PYTHON)
    add_compile_definitions(ENABLE_PYTHON)
endif()

set(CMAKE_CXX_FLAGS_DEBUG "$ENV{CXXFLAGS} -O0 -g2 -ggdb -fno-inline-functions -fno-omit-frame-pointer \
    -Wl,--allow-shlib-undefined -D_LIBCPP_INLINE_VISIBILITY='' -D_LIBCPP_DISABLE_EXTERN_TEMPLATE=1 \
    -DHALF_ENABLE_CPP11_USER_LITERALS=0 -D_FORTIFY_SOURCE=2 -Wno-cpp")

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -I/usr/local/include -std=c++17 \
    -Werror -Wall -Wno-deprecated-declarations -fPIC")
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

set(PYBIND11_CPP_STANDARD -std=c++17)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OPTION_CXX_FLAGS}")

if(ENABLE_AKG AND (ENABLE_D OR ENABLE_GPU))
    add_subdirectory("${CMAKE_SOURCE_DIR}/akg")
endif()

include(${CMAKE_SOURCE_DIR}/cmake/mind_expression.cmake)
include_directories(${CMAKE_CURRENT_SOURCE_DIR})
include_directories(${CMAKE_CURRENT_SOURCE_DIR}/third_party/securec/include)
include_directories(${CMAKE_CURRENT_SOURCE_DIR}/third_party/flatbuffers/include)
include_directories(${CMAKE_CURRENT_SOURCE_DIR}/third_party/flatbuffers/include/flatbuffers)
```

```
include(${CMAKE_SOURCE_DIR}/cmake/dependency_utils.cmake)
find_package(Python3 COMPONENTS Interpreter Development)
if(Python3_FOUND)
    set(PYTHON_INCLUDE_DIRS "${Python3_INCLUDE_DIRS}")
    set(PYTHON_LIBRARIES "${Python3_LIBRARIES}")
    if(WIN32)
        if(Python3_DIR)
            message("Python3_DIR set already: " ${Python3_DIR})
        else()
            string(LENGTH ${PYTHON_LIBRARIES} PYTHON_LIBRARIES_LEN)
            string(LENGTH "libpythonxx.a" Python3_NAME_LEN)
            math(EXPR Python3_DIR_LEN  ${PYTHON_LIBRARIES_LEN}-${Python3_NAME_LEN})
            string(SUBSTRING ${Python3_LIBRARIES} 0 ${Python3_DIR_LEN} Python3_DIR)
            message("Python3_DIR: " ${Python3_DIR})
        endif()
        link_directories(${Python3_DIR})
    endif()
else()
    find_python_package(py_inc py_lib)
    set(PYTHON_INCLUDE_DIRS "${py_inc}")
    set(PYTHON_LIBRARIES "${py_lib}")
endif()
message("PYTHON_INCLUDE_DIRS = ${PYTHON_INCLUDE_DIRS}")
message("PYTHON_LIBRARIES = ${PYTHON_LIBRARIES}")
include_directories(${PYTHON_INCLUDE_DIRS})

set(MS_CCSRC_PATH ${CMAKE_SOURCE_DIR}/mindspore/ccsrc)
set(MS_CCSRC_BUILD_PATH ${BUILD_PATH}/mindspore/mindspore/ccsrc)

if(ENABLE_D OR ENABLE_ACL OR ENABLE_TESTCASES)
    include(${CMAKE_SOURCE_DIR}/cmake/dependency_graphengine.cmake)
endif()

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fvisibility=hidden")
add_subdirectory(mindspore/ccsrc)
add_subdirectory(mindspore/core)
if(ENABLE_TESTCASES OR ENABLE_CPP_ST)
    add_subdirectory(tests)
endif()

include(cmake/package.cmake)
```

MindSpore Lite 支持 JAVA 和 C++，由于调用手机摄像头需要使用 JAVA，因此推理代码可以 C++和 JAVA 一起用或纯 JAVA 编写，本示例采用 C++和 JAVA 一起用，JAVA 用于调用手机摄像头和加载模型，C++用于推理，过程如下，完整代码见：MindSporeNetnative.cpp 和 MSNetWork.cpp。

步骤 1 加载模型文件：创建并配置用于模型推理的上下文，这里是使用 JAVA 接口加载了模型，C++代码读取的模型 buffer

```
// Buffer is the model data passed in by the Java layer
jlong bufferLen = env->GetDirectBufferCapacity(buffer);
char *modelBuffer = CreateLocalModelBuffer(env, buffer);
```

步骤 2 创建会话

```
void **labelEnv = new void *;
MSNetWork *labelNet = new MSNetWork;
*labelEnv = labelNet;

// Create context.
mindspore::lite::Context *context = new mindspore::lite::Context;
context->thread_num_ = num_thread;

// Create the mindspore session.
labelNet->CreateSessionMS(modelBuffer, bufferLen, context);
delete (context);
context->device_list_[0].device_info_.cpu_device_info_.cpu_bind_mode_ = mindspore::lite::NO_BIND;
context->device_list_[0].device_info_.cpu_device_info_.enable_float16_ = true;
context->device_list_[0].device_type_ = mindspore::lite::DT_CPU;
```

步骤 3 加载模型文件并构建用于推理的计算图

```
void MSNetWork::CreateSessionMS(char* modelBuffer, size_t bufferLen, std::string name,
mindspore::lite::Context* ctx)
{
    CreateSession(modelBuffer, bufferLen, ctx);
    session = mindspore::session::LiteSession::CreateSession(ctx);
    auto model = mindspore::lite::Model::Import(modelBuffer, bufferLen);
    int ret = session->CompileGraph(model);
}
```

步骤 4 将待检测图片数据转换为输入 MindSpore 模型的 Tensor

```
// Convert the Bitmap image passed in from the JAVA layer to Mat for OpenCV processing
 BitmapToMat(env, srcBitmap, matImageSrc);
// Processing such as zooming the picture size.
matImgPreprocessed = PreProcessImageData(matImageSrc);

ImgDims inputDims;
inputDims.channel = matImgPreprocessed.channels();
inputDims.width = matImgPreprocessed.cols;
inputDims.height = matImgPreprocessed.rows;
float *dataHWC = new float[inputDims.channel * inputDims.width * inputDims.height]

// Copy the image data to be detected to the dataHWC array.
```

```
// The dataHWC[image_size] array here is the intermediate variable of the input MindSpore model tensor.
float *ptrTmp = reinterpret_cast<float *>(matImgPreprocessed.data);
for(int i = 0; i < inputDims.channel * inputDims.width * inputDims.height; i++){
    dataHWC[i] = ptrTmp[i];
}

// Assign dataHWC[image_size] to the input tensor variable.
auto msInputs = mSession->GetInputs();
auto inTensor = msInputs.front();
memcpy(inTensor->MutableData(), dataHWC,
    inputDims.channel * inputDims.width * inputDims.height * sizeof(float));
delete[] (dataHWC);
```

步骤 5 对输入数据进行处理

```
bool PreProcessImageData(const LiteMat &lite_mat_bgr, LiteMat *lite_norm_mat_ptr) {
    bool ret = false;
    LiteMat lite_mat_resize;
    LiteMat &lite_norm_mat_cut = *lite_norm_mat_ptr;
    ret = ResizeBilinear(lite_mat_bgr, lite_mat_resize, 256, 256);
    if (!ret) {
        MS_PRINT("ResizeBilinear error");
        return false;
    }
    LiteMat lite_mat_convert_float;
    ret = ConvertTo(lite_mat_resize, lite_mat_convert_float, 1.0 / 255.0);
    if (!ret) {
        MS_PRINT("ConvertTo error");
        return false;
    }
    LiteMat lite_mat_cut;
    ret = Crop(lite_mat_convert_float, lite_mat_cut, 16, 16, 224, 224);
    if (!ret) {
        MS_PRINT("Crop error");
        return false;
    }
    float means[3] = {0.485, 0.456, 0.406};
    float vars[3] = {1.0 / 0.229, 1.0 / 0.224, 1.0 / 0.225};
    SubStractMeanNormalize(lite_mat_cut, lite_norm_mat_cut, means, vars);
    return true;
}
```

步骤 6 图执行，端测推理

```
// After the model and image tensor data is loaded, run inference.
auto status = mSession->RunGraph();
```

步骤 7 获取输出数据

```
auto names = mSession->GetOutputTensorNames();
std::unordered_map<std::string, mindspore::tensor::MSTensor *> msOutputs;
for (const auto &name : names) {
    auto temp_dat =mSession->GetOutputByTensorName(name);
    msOutputs.insert(std::pair<std::string, mindspore::tensor::MSTensor *> {name, temp_dat});
  }
std::string retStr = ProcessRunnetResult(msOutputs, ret);
```

步骤 8 输出数据的后续处理

```
std::string ProcessRunnetResult(const int RET_CATEGORY_SUM, const char *const labels_name_map[],
                                std::unordered_map<std::string, mindspore::tensor::MSTensor *>
msOutputs) {
    // Get the branch of the model output.
    // Use iterators to get map elements.
    std::unordered_map<std::string, mindspore::tensor::MSTensor *>::iterator iter;
    iter = msOutputs.begin();
    // The mobilenetv2.ms model output just one branch.
    auto outputTensor = iter->second;
    int tensorNum = outputTensor->ElementsNum();
    MS_PRINT("Number of tensor elements:%d", tensorNum);
    // Get a pointer to the first score.
    float *temp_scores = static_cast<float *>(outputTensor->MutableData());
    float scores[RET_CATEGORY_SUM];
    for (int i = 0; i < RET_CATEGORY_SUM; ++i) {
        scores[i] = temp_scores[i];
    }

    float unifiedThre = 0.5;
    float probMax = 1.0;
    for (size_t i = 0; i < RET_CATEGORY_SUM; ++i) {
        float threshold = g_thres_map[i];
        float tmpProb = scores[i];
        if (tmpProb < threshold) {
            tmpProb = tmpProb / threshold * unifiedThre;
        } else {
            tmpProb = (tmpProb - threshold) / (probMax - threshold) * unifiedThre + unifiedThre;
        }
        scores[i] = tmpProb;
    }
    for (int i = 0; i < RET_CATEGORY_SUM; ++i) {
        if (scores[i] > 0.5) {
            MS_PRINT("MindSpore scores[%d] : [%f]", i, scores[i]);
        }
    }

    // Score for each category.
    // Converted to text information that needs to be displayed in the APP.
```

```
    std::string categoryScore = "";
    for (int i = 0; i < RET_CATEGORY_SUM; ++i) {
        categoryScore += labels_name_map[i];
        categoryScore += ":";
        std::string score_str = std::to_string(scores[i]);
        categoryScore += score_str;
        categoryScore += ";";
    }
    return categoryScore;
}
```

更多完整 APP 构建案例，请参考以下链接：https://www.mindspore.cn/tutorial/lite/zh-CN/r1.2/quick_start/quick_start.html

1.3.6 实验小结

本实验通过 Fine Tune 训练网络模型，再将模型转换为可以部署到手机端的 ms 模型，进而在手机端成功部署。

1.4 思考题

一般一个分类网络包含两个部分：backbone 和 head，其中 backbone 部分通常是一系列卷积层，负责提取图片特征；head 部分通常是一组全连接层，用于分类，一般最后一个全连接层的输出对应使用数据集的分类个数，在进行迁移学习微调时，一般冻结哪个部分，微调哪个部分?

答案：一般情况下冻结 backbone 部分的参数， 只训练 head 的参数进行微调。

1.5 缩略语

ONNX：Open Neural Network Exchange，一种针对机器学习模型的通用表达。

MindIR：MindSpore IR，MindSpore 的一种基于图表示的函数式 IR。

AIR：Ascend Intermediate Representation，华为定义的针对机器学习所设计的开放式文件格式。

华为认证 AI 系列教程

HCIP-AI-MindSpore Developer

图像处理

实验指导手册

版本：1.0

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

华为认证体系介绍

华为认证是华为公司基于“平台+生态”战略，围绕“云-管-端”协同的新ICT技术架构，打造的覆盖ICT（Information and Communications Technology 信息通信技术）全技术领域的认证体系，包含ICT技术架构与应用认证、云服务与平台认证两类认证。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

华为认证覆盖ICT全领域，符合ICT融合的技术趋势，致力于提供领先的人才培养体系和认证标准，培养数字化时代新型ICT人才，构建良性ICT人才生态。

HCIP-AI-MindSpore Developer V1.0 定位于培养和认证具备使用华为深度学习框架MindSpore进行人工智能开发能力的算法工程师。

通过HCIP-AI-MindSpore Developer V1.0认证，您将掌握MindSpore和MindSpore Lite的架构与特性相关知识，理解MindSpore网络迁移、分布式训练、端云侧推理与部署等相关特性；具备使用MindSpore完成图像处理任务和自然语言处理任务的能力；能够胜任人工智能算法工程师、深度学习工程师、图像算法工程师、自然语言处理算法工程师等岗位。


Huawei Certification
Huawei Certified ICT Expert
HCIE
Huawei Certified ICT Professional
HCIP
Huawei Certified ICT Associate
HCIA
Cloud Service & Platform
云服务与平台
HarmonyOS(To Be)
IoT
GaussDB
openGauss (To Be)
openEuler
Cloud(Cloud Computing + Cloud Service)
Intelligent Driving (To Be)
Big Data
AI
Kunpeng Application Developer
ICT Infrastructure
ICT技术架构与应用
Storage
Datacom
Transmission
Intelligent Computing
WLAN
Access
Enterprise Communication
Security
LTE
Intelligent Vision
SDN
5G
Data Center

前 言

简介

本书为 HCIP-AI-MindSpore Developer V1.0 中文版认证培训教程，适用于准备参加 HCIP-AI-MindSpore Developer 考试的学员或者希望了解人工智能图像处理中如何进行迁移学习、对抗示例生成以及深度卷积对抗生成网络的读者。

内容描述

本实验指导书共包含 3 个实验：

- 实验一图像分类迁移学习，为了减少从头训练所带来的时间成本，大多数情况下会基于已有模型来进行迁移学习，本实验讲解如何在 MindSpore 中加载预训练模型，并通过固定权重来实现迁移学习；
- 实验二中将以梯度符号攻击 FGSM（Fast Gradient Sign Method）为例，演示此类攻击是如何误导模型的；
- 实验三将使用真实名人的照片来训练一个生成对抗网络（GAN），接着产生虚假名人图片。

读者知识背景

本课程为华为认证高级课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有人工智能知识背景，了解人工智能开发流程；
- 熟悉 Python 语言；
- 掌握机器学习算法以及深度学习算法的基础原理；
- 熟悉深度学习框架；

实验环境说明

- 华为云（https://www.huaweicloud.com/）-ModelArts
- MindSpore1.3.0

目录

1 图像分类迁移学习

1.1 图像分类迁移学习

1.1.1 实验介绍

1.1.1.1 关于本实验

在实际场景中，为了减少从头开始训练模型所带来的时间成本，大多数情况下会基于已有的模型来进行迁移学习。本章将会以狗和狼的图像分类为例，讲解如何基于深度学习框架MindSpore 加载预训练模型，并通过固定权重来实现迁移学习的目的。

1.1.1.2 实验目的

- 描述迁移学习的目的。
- 实现基于 MindSpore 进行迁移学习。

1.1.1.3 预备知识

迁移学习(Transfer Learning) 顾名思义就是把已训练好的模型（预训练模型）参数迁移到新的模型来帮助新模型训练。考虑到大部分数据或任务都是存在相关性的，所以通过迁移学习我们可以将已经学到的模型参数（也可理解为模型学到的知识）通过某种方式来分享给新模型从而加快并优化模型的学习效率，不用像大多数网络那样从零学习。其中，实现迁移学习有以下三种手段：

- **Transfer Learning**：冻结预训练模型的全部卷积层，只训练自己定制的全连接层。
- **Extract Feature Vector**：先计算出预训练模型的卷积层对所有训练和测试数据的特征向量，然后抛开预训练模型，只训练自己定制的简配版全连接网络。
- **Fine-tuning**：冻结预训练模型的部分卷积层（通常是靠近输入的多数卷积层，因为这些层保留了大量底层信息）甚至不冻结任何网络层，训练剩下的卷积层（通常是靠近输出的部分卷积层）和全连接层。

1.1.2 实验任务

1.1.2.1 准备环节

步骤 1　数据集和预训练模型准备

数据集中的图像来自于 ImageNet，每个分类有大约 120 张训练图像与 30 张验证图像。将下载后的数据集解压到当前目录下。

数据目录结构如下：

```
./
└─resnet50.ckpt
└─data
    └─Canidae
        └─train
        │    └─dogs
        │    └─wolves
        └─val
             └─dogs
             └─wolves
```

输入：

```
!wget https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap4/Canidae_data.zip
!wget https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap4/resnet50.ckpt
!wget https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap4/resnet.py
!unzip Canidae_data.zip
```

步骤 2　导入模块

输入：

```
import os
import math
import stat
import numpy as np
import matplotlib.pyplot as plt

import mindspore.ops as ops
import mindspore.nn as nn
import mindspore.dataset as ds
import mindspore.dataset.vision.c_transforms as CV
import mindspore.dataset.transforms.c_transforms as C
from mindspore.dataset.vision import Inter
from mindspore.common.initializer import Normal
from mindspore import dtype as mstype
from mindspore.train.callback import TimeMonitor, Callback
from mindspore import Model, Tensor, context, save_checkpoint, load_checkpoint, load_param_into_net
from resnet import resnet50
```

步骤 3　环境配置

我们使用 GRAPH 模式运行实验，使用 Ascend 环境。

```
context.set_context(mode=context.GRAPH_MODE, device_target="Ascend")
```

1.1.2.2 加载数据和数据可视化

步骤 1 加载数据

定义训练和验证数据集的路径。

```
train_data_path = 'data/Canidae/train'
val_data_path = 'data/Canidae/val'
```

定义 create_dataset 函数对数据进行处理。

输入：

```
def create_dataset(data_path, batch_size=24, repeat_num=1, training=True):
    """定义数据集"""
    data_set = ds.ImageFolderDataset(data_path, num_parallel_workers=8, shuffle=True)

    # 对数据进行增强操作
    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()
        ]
    type_cast_op = C.TypeCast(mstype.int32)

    # 实现数据的 map 映射、批量处理和数据重复的操作
    data_set = data_set.map(operations=trans, input_columns="image", num_parallel_workers=8)
    data_set = data_set.map(operations=type_cast_op, input_columns="label", num_parallel_workers=8)
    data_set = data_set.batch(batch_size, drop_remainder=True)
    data_set = data_set.repeat(repeat_num)

    return data_set
```

实例化数据集

```
train_ds =  create_dataset(train_data_path)
```

步骤 2 可视化图像

通过 matplotlib 可视化部分增强后的训练数据。

输入：

```
data = next(train_ds.create_dict_iterator())
images = data["image"]
labels = data["label"]
print("Tensor of image", images.shape)
print("Labels:",labels)
class_name = {0:"dogs",1:"wolves"}
count = 1

# 输出测试图
plt.figure(figsize=(12,5))
for i in images:
    plt.subplot(3,8,count)
    plt.imshow(i.asnumpy().transpose(1,2,0))
    plt.title(class_name[int(labels[count-1].asnumpy())])
    plt.xticks([])
    count += 1
    plt.axis("off")
plt.show()
```

输出：

```
Tensor of image (24, 3, 224, 224)
Labels: [0 0 1 0 0 1 0 1 1 1 0 0 1 0 1 0 0 0 1 0 0 1 0 1]
```

图1-1 原始图片示例

1.1.2.3 定义网络和训练模型

直接用 Model Zoo 中的 ResNet50。

步骤 1　建立模型：Transfer Learning

从训练好的 ckpt 文件里删除需要重置的参数。

```
def filter_checkpoint_parameter_by_list(origin_dict, param_filter):
    for key in list(origin_dict.keys()):
```

```
            for name in param_filter:
                if name in key:
                    print("Delete parameter from checkpoint: ", key)
                    del origin_dict[key]
                    break
```

加载预训练的模型并重置最终的全连接层。

输入：

```
net = resnet50(2)
num_epochs=20

# 加载预训练模型
param_dict = load_checkpoint('resnet50.ckpt')

# 获取全连接层的名字
filter_list = [x.name for x in net.end_point.get_parameters()]

# 删除预训练模型的全连接层
filter_checkpoint_parameter_by_list(param_dict, filter_list)

# 给网络加载参数
load_param_into_net(net,param_dict)

# 定义优化器和损失函数
opt = nn.Momentum(params=net.trainable_params(), learning_rate=0.001, momentum=0.9)
loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True,reduction='mean')

# 实例化模型
model = Model(net, loss,opt,metrics={"Accuracy":nn.Accuracy()})
```

输出：

```
Delete parameter from checkpoint:   end_point.weight
Delete parameter from checkpoint:   end_point.bias
Delete parameter from checkpoint:   moments.end_point.weight
Delete parameter from checkpoint:   moments.end_point.bias
```

步骤 2　训练和评估模型

定义 apply_eval 函数，用来验证模型的精度。

```
# 模型验证
def apply_eval(eval_param):
    eval_model = eval_param['model']
    eval_ds = eval_param['dataset']
    metrics_name = eval_param['metrics_name']
    res = eval_model.eval(eval_ds)
    return res[metrics_name]
```

我们自定义一个数据收集的回调类 EvalCallBack，用于实现下面两种信息：

- 训练过程中，每一个 epoch 结束之后，训练集的损失值和验证集的模型精度。
- 保存精度最高的模型。

```
class EvalCallBack(Callback):
    """
    回调类，获取训练过程中模型的信息
    """

    def __init__(self, eval_function, eval_param_dict, interval=1, eval_start_epoch=1,
save_best_ckpt=True,
                 ckpt_directory="./", besk_ckpt_name="best.ckpt", metrics_name="acc"):
        super(EvalCallBack, self).__init__()
        self.eval_param_dict = eval_param_dict
        self.eval_function = eval_function
        self.eval_start_epoch = eval_start_epoch
        if interval < 1:
            raise ValueError("interval should >= 1.")
        self.interval = interval
        self.save_best_ckpt = save_best_ckpt
        self.best_res = 0
        self.best_epoch = 0
        if not os.path.isdir(ckpt_directory):
            os.makedirs(ckpt_directory)
        self.best_ckpt_path = os.path.join(ckpt_directory, besk_ckpt_name)
        self.metrics_name = metrics_name

    # 删除 ckpt 文件
    def remove_ckpoint_file(self, file_name):
        os.chmod(file_name, stat.S_IWRITE)
        os.remove(file_name)

    # 每一个 epoch 后，打印训练集的损失值和验证集的模型精度，并保存精度最好的 ckpt 文件
    def epoch_end(self, run_context):
        cb_params = run_context.original_args()
        cur_epoch = cb_params.cur_epoch_num
        loss_epoch = cb_params.net_outputs
        if cur_epoch >= self.eval_start_epoch and (cur_epoch - self.eval_start_epoch) % self.interval ==
0:
            res = self.eval_function(self.eval_param_dict)
            print('Epoch {}/{}'.format(cur_epoch, num_epochs))
            print('-' * 10)
            print('train Loss: {}'.format(loss_epoch))
            print('val Acc: {}'.format(res))
            if res >= self.best_res:
                self.best_res = res
                self.best_epoch = cur_epoch
                if self.save_best_ckpt:
                    if os.path.exists(self.best_ckpt_path):
```

```
                                self.remove_ckpoint_file(self.best_ckpt_path)
                            save_checkpoint(cb_params.train_network, self.best_ckpt_path)

    # 训练结束后，打印最好的精度和对应的 epoch
    def end(self, run_context):
        print("End training, the best {0} is: {1}, the best {0} epoch is {2}".format(self.metrics_name,

self.best_res,

self.best_epoch), flush=True)
```

运行下面代码，开始模型训练。

输入：

```
train_ds =   create_dataset(train_data_path)
val_ds = create_dataset(val_data_path)
eval_param_dict = {"model":model,"dataset":val_ds,"metrics_name":"Accuracy"}
eval_cb = EvalCallBack(apply_eval, eval_param_dict,)

# 训练模型
model.train(num_epochs,train_ds, callbacks=[eval_cb, TimeMonitor()], dataset_sink_mode=True)
```

输出：

```
Epoch 1/20
----------
train Loss: 0.47486544
val Acc: 0.8333333333333334
epoch time: 8439.054 ms, per step time: 140.651 ms
Epoch 2/20
----------
train Loss: 0.20464368
val Acc: 0.8333333333333334
epoch time: 3805.755 ms, per step time: 63.429 ms
Epoch 3/20
----------
train Loss: 0.3345307
val Acc: 0.9166666666666666
epoch time: 3721.042 ms, per step time: 62.017 ms
Epoch 4/20
----------
train Loss: 0.7761406
val Acc: 0.8333333333333334
epoch time: 3302.892 ms, per step time: 55.048 ms
Epoch 5/20
----------
```

步骤 3　可视化模型预测

定义 visualize_mode 函数，可视化模型预测。

```
def visualize_model(best_ckpt_path,val_ds):
    # 定义网络并加载参数，对验证集进行预测
    net = resnet50(2)
    param_dict = load_checkpoint(best_ckpt_path)
    load_param_into_net(net,param_dict)
    loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True,reduction='mean')
    model = Model(net, loss,metrics={"Accuracy":nn.Accuracy()})
    data = next(val_ds.create_dict_iterator())
    images = data["image"].asnumpy()
    labels = data["label"].asnumpy()
    class_name = {0:"dogs",1:"wolves"}
    output = model.predict(Tensor(data['image']))
    pred = np.argmax(output.asnumpy(),axis=1)

    # 可视化模型预测
    plt.figure(figsize=(12,5))
    for i in range(len(labels)):
        plt.subplot(3,8,i+1)


        color = 'blue' if pred[i] == labels[i] else 'red'
        plt.title('pre:{}'.format(class_name[pred[i]]), color=color)
        plt.imshow(images[i].transpose(1,2,0))
        plt.axis('off')
    plt.show()
```

使用精度最好的模型对验证集进行可视化预测。

输入：

```
visualize_model('best.ckpt', val_ds)
```

输出：

图1-2 可视化预测结果

步骤 4　建立模型：Extract Feature Vector

我们需要冻结除最后一层之外的所有网络。通过设置 requires_grad = False 冻结参数，以便不在反向传播中计算梯度。

```
# 定义网络
net = resnet50(2)
num_epochs=20

# 加载预训练模型
param_dict = load_checkpoint('resnet50.ckpt')

# 获取最后一层参数的名字
filter_list = [x.name for x in net.end_point.get_parameters()]

# 删除预训练模型最后一层的参数
filter_checkpoint_parameter_by_list(param_dict, filter_list)

# 给网络加载参数
load_param_into_net(net,param_dict)

# 冻结除最后一层外的所有参数
for param in net.get_parameters():
    if param.name not in ["end_point.weight","end_point.bias"]:
        param.requires_grad = False

# 定义优化器和损失函数
opt = nn.Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9)
loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True,reduction='mean')

# 实例化模型
model = Model(net, loss,opt,metrics={"Accuracy":nn.Accuracy()})
```

输出：

```
Delete parameter from checkpoint:  end_point.weight
Delete parameter from checkpoint:  end_point.bias
Delete parameter from checkpoint:  moments.end_point.weight
Delete parameter from checkpoint:  moments.end_point.bias
```

步骤 5　训练与评估模型

开始训练模型，与没有预训练模型相比，将节约一大半时间，因为此时可以不用计算部分梯度。

说明：由于机器性能不同，用户实际的运行时间可能会比案例中的长。

输入：

```
# 加载训练和验证数据集
train_ds =  create_dataset(train_data_path)
val_ds = create_dataset(val_data_path)
```

```
# 实例化回调类
eval_param_dict = {"model":model,"dataset":val_ds,"metrics_name":"Accuracy"}
eval_cb = EvalCallBack(apply_eval, eval_param_dict,)

# 模型训练
model.train(num_epochs,train_ds, callbacks=[eval_cb, TimeMonitor()], dataset_sink_mode=True)
```

输出：

```
Epoch 1/20
----------
train Loss: 16.637333
val Acc: 0.5166666666666667
epoch time: 5061.097 ms, per step time: 84.352 ms
Epoch 2/20
----------
train Loss: 0.00037534387
val Acc: 0.7333333333333333
epoch time: 2339.947 ms, per step time: 38.999 ms
Epoch 3/20
----------
train Loss: 0.0
val Acc: 0.7166666666666667
epoch time: 1988.850 ms, per step time: 33.147 ms
Epoch 4/20
----------
train Loss: 3.231116
val Acc: 0.8333333333333334
epoch time: 2145.357 ms, per step time: 35.756 ms
Epoch 5/20
----------
```

步骤 6　可视化模型预测

使用精度最好的模型对验证集进行可视化预测。

输入：

```
visualize_model('best.ckpt', val_ds)
```

输出：

图1-3 验证集的可视化预测

1.2 对抗示例生成

1.2.1 实验介绍

1.2.1.1 关于本实验

近年来随着数据、计算能力、理论的不断发展演进，深度学习在图像、文本、语音、自动驾驶等众多领域都得到了广泛应用。与此同时，人们也越来越关注各类模型在使用过程中的安全问题，因为 AI 模型容易受到外界攻击而产生错误的结果。在本实验中，我们将以梯度符号攻击（Fast Gradient Sign Method，FGSM）为例，演示此类攻击是如何误导模型的。

1.2.1.2 实验目的

- 描述梯度符号攻击 FGSM 的基本原理；
- 基于 MindSpore 实现梯度符号攻击。

1.2.1.3 预备知识

1.2.1.3.1 对抗样本定义

Szegedy 在 2013 年最早提出对抗样本的概念：在原始样本处加入人类无法察觉的微小扰动，使得深度模型性能下降，这种样本即对抗样本。如下图所示，本来预测为“panda”的图像在添加噪声之后，模型就将其预测为“gibbon”，右边的样本就是一个对抗样本。

1.2.1.3.2 攻击方法

对模型的攻击方法可以按照以下方法分类：

1、攻击者掌握的信息多少。

- 白盒攻击：攻击者具有对模型的全部知识和访问权限，包括模型结构、权重、输入、输出。攻击者在产生对抗性攻击数据的过程中能够与模型系统有所交互。攻击者可以针对被攻击模型的特性设计特定的攻击算法。

- 黑盒攻击：与白盒攻击相反，攻击者仅具有关于模型的有限知识。攻击者对模型的结构权重一无所知，仅了解部分输入输出。

2、攻击者的目的。

- 有目标的攻击：攻击者将模型结果误导为特定分类。
- 无目标的攻击：攻击者只想产生错误结果，而不在乎新结果是什么。

本实验中用到的 FGSM 是一种白盒攻击方法，既可以是有目标也可以是无目标攻击。

更多的模型安全功能可参考 MindArmour，现支持 FGSM、LLC、Substitute Attack 等多种对抗样本生成方法，并提供对抗样本鲁棒性模块、Fuzz Testing 模块、隐私保护与评估模块，帮助用户增强模型安全性。

1.2.1.3.3 快速梯度符号攻击（FGSM）

正常分类网络的训练会定义一个损失函数，用于衡量模型输出值与样本真实标签的距离，通过反向传播计算模型梯度，梯度下降更新网络参数，减小损失值，提升模型精度。

FGSM（Fast Gradient Sign Method）是一种简单高效的对抗样本生成方法。不同于正常分类网络的训练过程，FGSM 通过计算 loss 对于输入的梯度∇xJ(θ,x,y)，这个梯度表征了 loss 对于输入变化的敏感性。然后基于原始输入加上上述梯度，使得 loss 增大，模型对于改造后的输入样本分类效果变差，达到攻击效果。对抗样本的另一要求是生成样本与原始样本的差异要尽可能的小，使用 sign 函数可以使得修改图片时尽可能的均匀。

产生的对抗扰动用公式可以表示为：

$$\eta = \varepsilon sign(\nabla x J(\theta, x, y))$$

对抗样本可公式化为：

$$x' = x + \varepsilon sign(\nabla x J(\theta, x, y))$$

其中，x 为正确分类为“panda”的原始输入图像，y 是 x 的输出，θ 为模型参数，ε 为攻击系数，J(θ,x,y)为训练网络的损失，∇xJ(θ, x, y)为反向传播梯度。

1.2.2 实验任务

1.2.2.1 实验准备

步骤 1　导入模型训练需要的库

```
import os
import numpy as np

from mindspore import Tensor, context, Model, load_checkpoint, load_param_into_net
import mindspore.nn as nn
import mindspore.ops as ops
from mindspore.common.initializer import Normal
from mindspore.train.callback import LossMonitor, ModelCheckpoint, CheckpointConfig
import mindspore.dataset as ds
import mindspore.dataset.transforms.c_transforms as C
import mindspore.dataset.vision.c_transforms as CV
from mindspore.dataset.vision import Inter
```

```
from mindspore import dtype as mstype
```

步骤 2 下载数据集

输入：

```
!mkdir -p ./datasets/MNIST_Data/train ./datasets/MNIST_Data/test
!wget -NP ./datasets/MNIST_Data/train https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap4/train-labels-idx1-ubyte
!wget -NP ./datasets/MNIST_Data/train https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap4/train-images-idx3-ubyte
!wget -NP ./datasets/MNIST_Data/test https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap4/t10k-labels-idx1-ubyte
!wget -NP ./datasets/MNIST_Data/test https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap4/t10k-images-idx3-ubyte
!tree ./datasets/MNIST_Data
```

步骤 3 环境配置

在完成准备工作之后，开始训练精度达标的 LeNet 网络。

采用 GRAPH_MODE 在 CPU/GPU/Ascend 中运行本实验，下面将硬件设定为 Ascend：

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

1.2.2.2 创建 LeNet 模型

实验中使用 LeNet 作为演示模型完成图像分类，这里先定义网络并使用 MNIST 数据集进行训练。

步骤 1 定义 LeNet 网络

输入：

```
class LeNet5(nn.Cell):

    def __init__(self, num_class=10, num_channel=1):
        super(LeNet5, self).__init__()
        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 * 5 * 5, 120, weight_init=Normal(0.02))
        self.fc2 = nn.Dense(120, 84, weight_init=Normal(0.02))
        self.fc3 = nn.Dense(84, num_class, weight_init=Normal(0.02))
        self.relu = nn.ReLU()
        self.max_pool2d = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten = nn.Flatten()

    def construct(self, x):
        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

net = LeNet5()
```

步骤 2　数据处理

输入：

```
def create_dataset(data_path, batch_size=1, repeat_size=1, num_parallel_workers=1):

    # 定义数据集
    mnist_ds = ds.MnistDataset(data_path)
    resize_height, resize_width = 32, 32
    rescale = 1.0 / 255.0
    shift = 0.0
    rescale_nml = 1 / 0.3081
    shift_nml = -1 * 0.1307 / 0.3081

    # 定义所需要操作的 map 映射
    resize_op = CV.Resize((resize_height, resize_width), interpolation=Inter.LINEAR)
    rescale_nml_op = CV.Rescale(rescale_nml, shift_nml)
    rescale_op = CV.Rescale(rescale, shift)
    hwc2chw_op = CV.HWC2CHW()
    type_cast_op = C.TypeCast(mstype.int32)

    # 使用 map 映射函数，将数据操作应用到数据集
    mnist_ds = mnist_ds.map(operations=type_cast_op, input_columns="label",
num_parallel_workers=num_parallel_workers)
    mnist_ds = mnist_ds.map(operations=resize_op, input_columns="image",
num_parallel_workers=num_parallel_workers)
    mnist_ds = mnist_ds.map(operations=rescale_op, input_columns="image",
num_parallel_workers=num_parallel_workers)
    mnist_ds = mnist_ds.map(operations=rescale_nml_op, input_columns="image",
num_parallel_workers=num_parallel_workers)
    mnist_ds = mnist_ds.map(operations=hwc2chw_op, input_columns="image",
num_parallel_workers=num_parallel_workers)

    # 进行 shuffle、batch 操作
    buffer_size = 10000
    mnist_ds = mnist_ds.shuffle(buffer_size=buffer_size)
    mnist_ds = mnist_ds.batch(batch_size, drop_remainder=True)
```

```
    return mnist_ds
```

步骤 3　设置损失函数与优化器

设置损失函数与优化器，并调用 model 接口创建对象：

```
net_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
net_opt = nn.Momentum(net.trainable_params(), learning_rate=0.01, momentum=0.9)
```

步骤 4　定义网络参数

```
config_ck = CheckpointConfig(save_checkpoint_steps=1875, keep_checkpoint_max=10)
ckpoint = ModelCheckpoint(prefix="checkpoint_lenet", config=config_ck)
```

步骤 5　定义 LeNet 网络的训练函数和测试函数

```
def test_net(model, data_path):
    ds_eval = create_dataset(os.path.join(data_path, "test"))
    acc = model.eval(ds_eval, dataset_sink_mode=False)
    print("{}".format(acc))

def train_net(model, epoch_size, data_path, repeat_size, ckpoint_cb, sink_mode):
    ds_train = create_dataset(os.path.join(data_path, "train"), 32, repeat_size)
    model.train(epoch_size, ds_train, callbacks=[ckpoint_cb, LossMonitor(125)],
dataset_sink_mode=sink_mode)
train_epoch = 1
mnist_path = "./datasets/MNIST_Data/"
repeat_size = 1
model = Model(net, net_loss, net_opt, metrics={"Accuracy": nn.Accuracy()})
```

步骤 6　训练 LeNet 网络

输入：

```
train_net(model, train_epoch, mnist_path, repeat_size, ckpoint, False)
```

输出：

```
epoch: 1 step: 125, loss is 2.3018382
epoch: 1 step: 250, loss is 2.2910337
epoch: 1 step: 375, loss is 2.2876282
epoch: 1 step: 500, loss is 2.293197
epoch: 1 step: 625, loss is 2.2983356
epoch: 1 step: 750, loss is 0.73134214
epoch: 1 step: 875, loss is 0.39000687
epoch: 1 step: 1000, loss is 0.12004304
epoch: 1 step: 1125, loss is 0.10009943
epoch: 1 step: 1250, loss is 0.31425583
epoch: 1 step: 1375, loss is 0.14330618
epoch: 1 step: 1500, loss is 0.05759584
```

```
epoch: 1 step: 1625, loss is 0.18315211
epoch: 1 step: 1750, loss is 0.19758298
epoch: 1 step: 1875, loss is 0.0815863
```

测试此时的网络，可以观察到 LeNet 已经达到比较高的精度：

输入：

```
test_net(model, mnist_path)
```

输出：

```
{'Accuracy': 0.9691}
```

加载已经训练好的 LeNet 模型：

```
param_dict = load_checkpoint("checkpoint_lenet-1_1875.ckpt")
load_param_into_net(net, param_dict)
```

1.2.2.3 实现 FGSM

在得到精准的 LeNet 网络之后，下面将会采用 FSGM 攻击方法，在图像中加载噪声后重新进行测试。先通过损失函数求取反向梯度：

```
class WithLossCell(nn.Cell):
    """
    包装网络与损失函数
    """
    def __init__(self, network, loss_fn):
        super(WithLossCell, self).__init__()
        self._network = network
        self._loss_fn = loss_fn

    def construct(self, data, label):
        out = self._network(data)
        return self._loss_fn(out, label)


class GradWrapWithLoss(nn.Cell):
    """
    通过 loss 求反向梯度
    """
    def __init__(self, network):
        super(GradWrapWithLoss, self).__init__()
        self._grad_all = ops.composite.GradOperation(get_all=True, sens_param=False)
        self._network = network

    def construct(self, inputs, labels):
        gout = self._grad_all(self._network)(inputs, labels)
        return gout[0]
```

然后根据公式实现 FGSM 攻击：

```
class FastGradientSignMethod:
    """
    实现 FGSM 攻击
    """
    def __init__(self, network, eps=0.07, loss_fn=None):
        # 变量初始化
        self._network = network
        self._eps =   eps
        with_loss_cell = WithLossCell(self._network, loss_fn)
        self._grad_all = GradWrapWithLoss(with_loss_cell)
        self._grad_all.set_train()


    def _gradient(self, inputs, labels):
        # 求取梯度
        out_grad = self._grad_all(inputs, labels)
        gradient = out_grad.asnumpy()
        gradient = np.sign(gradient)
        return gradient

    def generate(self, inputs, labels):
        # 实现 FGSM
        inputs_tensor = Tensor(inputs)
        labels_tensor = Tensor(labels)
        gradient = self._gradient(inputs_tensor, labels_tensor)
        # 产生扰动
        perturbation = self._eps*gradient
        # 生成受到扰动的图片
        adv_x = inputs + perturbation
        return adv_x

    def batch_generate(self, inputs, labels, batch_size=32):
        # 对数据集进行处理
        arr_x = inputs
        arr_y = labels
        len_x = len(inputs)
        batches = int(len_x / batch_size)
        rest = len_x - batches*batch_size
        res = []
        for i in range(batches):
            x_batch = arr_x[i*batch_size: (i + 1)*batch_size]
            y_batch = arr_y[i*batch_size: (i + 1)*batch_size]
            adv_x = self.generate(x_batch, y_batch)
            res.append(adv_x)
        adv_x = np.concatenate(res, axis=0)
        return adv_x
```

再次处理 MINIST 数据集中测试集的图片：

```
images = []
labels = []
test_images = []
test_labels = []
predict_labels = []

ds_test = create_dataset(os.path.join(mnist_path, "test"),
batch_size=32).create_dict_iterator(output_numpy=True)

for data in ds_test:
    images = data['image'].astype(np.float32)
    labels = data['label']
    test_images.append(images)
    test_labels.append(labels)
    pred_labels = np.argmax(model.predict(Tensor(images)).asnumpy(), axis=1)
    predict_labels.append(pred_labels)

test_images = np.concatenate(test_images)
predict_labels = np.concatenate(predict_labels)
true_labels = np.concatenate(test_labels)
```

1.2.2.4 运行攻击

由 FGSM 攻击公式中可以看出，攻击系数 ε 越大，对梯度的改变就越大。当 ε 为零时则攻击效果不体现。

$$\eta = \varepsilon sign(\nabla x J(\theta, x, y))$$

现在先观察当 ε 为零时的攻击效果：

输入：

```
fgsm = FastGradientSignMethod(net, eps=0.0, loss_fn=net_loss)
advs = fgsm.batch_generate(test_images, true_labels, batch_size=32)

adv_predicts = model.predict(Tensor(advs)).asnumpy()
adv_predicts = np.argmax(adv_predicts, axis=1)
accuracy = np.mean(np.equal(adv_predicts, true_labels))
print(accuracy)
```

输出：

```
0.9602363782051282
```

再将ε设定为 0.5，尝试运行攻击：

输入：

```
fgsm = FastGradientSignMethod(net, eps=0.5, loss_fn=net_loss)
advs = fgsm.batch_generate(test_images, true_labels, batch_size=32)

adv_predicts = model.predict(Tensor(advs)).asnumpy()
```

```
adv_predicts = np.argmax(adv_predicts, axis=1)
accuracy = np.mean(np.equal(adv_predicts, true_labels))
print(accuracy)
```

输出：

```
0.3212139423076923
```

此时 LeNet 模型的精度大幅降低。

1.2.3 实验小结

本实验使用 MNIST 训练一个精度达标的 LeNet 网络，然后运行上文中所提到的 FGSM 攻击方法，实现错误分类的效果。

1.2.4 思考题

问题：我们做梯度符号攻击（Fast Gradient Sign Method，FGSM）的目的是什么?

答案：目的是希望在原始图片上做肉眼难以识别的修改，但是却可以让图像识别产生误判。

1.3 深度卷积对抗生成网络

1.3.1 实验介绍

1.3.1.1 关于本实验

本实验通过真实名人的照片来训练一个生成对抗网络（GAN），接着产生虚假名人图片。

1.3.1.2 实验目的

- 列举 GAN 的基本原理与功能。
- 构建 GAN 网络。
- 构建 GAN 判别器和生成器。

1.3.1.3 实验介绍

生成对抗网络（GAN, Generative Adversarial Networks ）是一种深度学习模型，是近年来复杂分布上无监督学习最具前景的方法之一。最初，GAN 由 Ian Goodfellow 于 2014 年发明，并在论文 Generative Adversarial Nets 中首次进行了描述。GAN 由两个不同的模型组成：生成器和判别器。生成器的任务是生成看起来像训练图像的“假”图像。判别器需要判断从生成器输出的图像是真实的训练图像还是生成的假图像。在训练过程中，生成器会不断尝试通过生成更好的假图像来骗过判别器，而判别器在这过程中也会逐步提升判别能力。这种博弈的平衡点是，当生成器生成的假图像看起来像训练数据时，判别器拥有 50%的真假判断置信度。

DCGAN（深度卷积对抗生成网络），DCGAN 是上述 GAN 的直接扩展。不同之处在于，DCGAN 会分别在判别器和生成器中使用卷积和卷积转置层。它最早由 Radford 等人在论文 Unsupervised Representation Learning With Deep Convolutional Generative Adversarial

Networks 中进行描述。判别器由分层的卷积层、BatchNorm 层和 LeakyReLU 激活层组成。输入是 3x64x64 的图像，输出是该图像为真图像的概率。生成器则是由转置卷积层、BatchNorm 层和 ReLU 激活层组成。输入是标准正态分布中提取出的隐向量 z，输出是 3x64x64 的 RGB 图像。在下面的教程中，提供了有关如何设置优化器、如何计算损失函数以及如何初始化模型权重的说明。

1.3.2 实验任务

1.3.2.1 准备环节

步骤 1 导入模块

输入：

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

import mindspore.dataset as ds
import mindspore.dataset.vision.c_transforms as vision
from mindspore.common.initializer import Initializer
from mindspore import nn, ops, Tensor, context
from mindspore import dtype as mstype
```

步骤 2 环境配置

我们使用 GRAPH 模式运行实验，使用 Ascend 环境。

```
context.set_context(mode=context.GRAPH_MODE, device_target="Ascend")
```

步骤 3 数据集和预训练模型准备

在本实验中，我们将使用 Celeb-A Faces 数据集，该数据集为人脸属性数据集，其包含 10,177 个名人身份的 202,599 张人脸图片。官网提供了多个下载链接，我们选择 Align&Cropped Images 下的 img_align_celeba.zip，是 202,599 张经过人脸对齐和裁剪了的图像。因数据集较大，本教程为了节省下载和训练时间，所以采用了部分的数据集。为了完整流畅地运行程序，需要在当前路径下创建一个 data 目录，并在 data 目录下创建一个名为 celeba 的目录，并将压缩文件解压缩到该目录中。最后，将此实验的 dataroot 输入设置为刚创建的 celeba 目录。数据目录结构如下：

```
./data/celeba
    -> img_align_celeba
        -> 188242.jpg
        -> 173822.jpg
        -> 284702.jpg
        -> 537394.jpg
            ...
```

输入：

```
!mkdir -p ./data/celeba
```

```
!wget -NP ./data/celeba https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap4/img_align_celeba.zip
!unzip  ./data/celeba/img_align_celeba.zip -d ./data/celeba/
```

1.3.2.2 加载数据和数据可视化

步骤 1 数据处理

首先为执行过程定义一些输入：

- dataroot：数据集文件夹根目录；
- workers：加载数据的线程数；
- batch_size：训练中使用的批量大小，DCGAN 论文使用的批量大小为 128；
- image_size：训练图像的大小，此实现默认为 64x64，如果需要其他尺寸，则必须同时更改 D 和 G 的结构；
- nc：输入图像中的彩色通道数，因为此次是彩色图像所以设为 3；
- nz：隐向量的长度；
- ngf：设置通过生成器的特征图的深度；
- ndf：设置通过判别器传播的特征图的深度；
- num_epochs：要运行的训练周期数，训练更长的时间可能会导致更好的结果，但也会花费更长的时间；
- lr：训练的学习率，如 DCGAN 论文中所述，此数字应为 0.0001；
- beta1：Adam 优化器的 beta1 超参数。如 DCGAN 论文所述，该数字应为 0.5。

```
# 数据集根目录
dataroot = "./data/celeba"
# 载入数据线程数
workers = 4
# 批量大小
batch_size = 128
# 训练图像空间大小，所有图像都将调整为该大小
image_size = 64
# 图像彩色通道数，对于彩色图像为 3
nc = 3
# 隐向量的长度
nz = 100
# 特征图在生成器中的大小
ngf = 64
# 特征图在判别器中的大小
ndf = 64
# 训练周期数
num_epochs = 10
# 学习率
lr = 0.0001
```

```
# Beta1 超参数
beta1 = 0.5
```

定义 create_dataset_imagenet 函数对数据进行处理和增强操作。

输入：

```
def create_dataset_imagenet(dataset_path, num_parallel_workers=None):
    # 数据加载
    data_set = ds.ImageFolderDataset(dataset_path, num_parallel_workers=num_parallel_workers,
shuffle=True)

    # 数据增强操作
    transform_img = [
        vision.Decode(),
        vision.Resize(image_size),
        vision.CenterCrop(image_size),
        vision.HWC2CHW()
    ]

    # 数据映射操作
    data_set = data_set.map(input_columns="image", num_parallel_workers=num_parallel_workers,
operations=transform_img,
                            output_columns="image")
    data_set = data_set.map(input_columns="image", num_parallel_workers=num_parallel_workers,
                            operations=lambda x: ((x - 255) / 255).astype("float32"))
    data_set = data_set.map(
        input_columns="image",
        operations=lambda x: (
            x,
            np.random.normal(size=(nz, 1, 1)).astype("float32")
        ),
        output_columns=["image", "latent_code"],
        column_order=["image", "latent_code"],
        num_parallel_workers=num_parallel_workers
    )

    # 批量操作
    data_set = data_set.batch(batch_size)

    return data_set

# 获取处理后的数据集
data = create_dataset_imagenet(dataroot, num_parallel_workers=workers)

# 获取数据集大小
size = data.get_dataset_size()
```

步骤 2　可视化图像

通过 create_dict_iterator 函数将数据转换成字典迭代器，然后使用 matplotlib 模块可视化部分训练数据。

输入：

```
data_iter = next(data.create_dict_iterator(output_numpy=True, num_epochs=num_epochs))
images = data_iter['image']
count = 1

# 可视化 36 张图片
for i in images[:36]:
    plt.subplot(6, 6, count)
    plt.imshow(i.transpose(1, 2, 0))
    plt.axis("off")
    plt.xticks([])
    count += 1

plt.show()
```

输出：

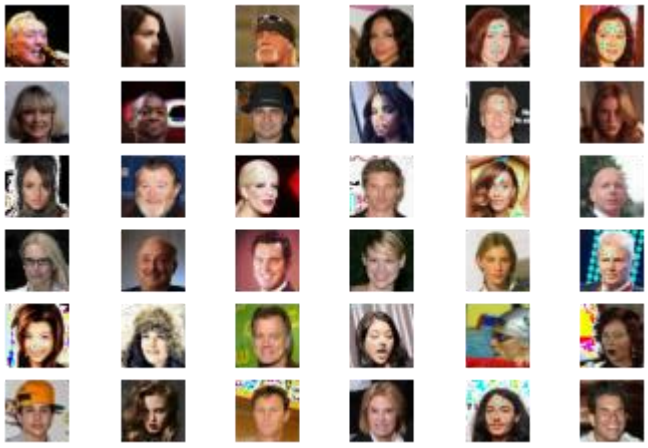

图1-4 原始图片示例

1.3.2.3 创建网络

当处理完数据后，就可以来进行网络的搭建了。网络搭建将以权重初始化策略为起点，逐一详细讨论生成器、判别器和损失函数。

步骤 1　权重初始化

实验遵循 DCGAN 论文中的内容，所有模型权重均应从 mean 为 0，sigma 为 0.02 的正态分布中随机初始化。

```
def _assignment(arr, num):
    if arr.shape == ():
        arr = arr.reshape((1))
        arr[:] = num
        arr = arr.reshape(())
```

```
    else:
        if isinstance(num, np.ndarray):
            arr[:] = num[:]
        else:
            arr[:] = num
    return arr

class Normal(Initializer):
    """将模型权重从均值为 0，标准差为 0.02 的正态分布中随机初始化"""
    def __init__(self, mean=0.0, sigma=0.02):
        super(Normal, self).__init__()
        self.sigma = sigma
        self.mean = mean

    def _initialize(self, arr):
        np.random.seed(999)
        arr_normal = np.random.normal(self.mean, self.sigma, arr.shape)
        _assignment(arr, arr_normal)
```

步骤 2　创建生成器

生成器 G 的功能是将隐向量 z 映射到数据空间。由于数据是图像，这一过程也会创建与真实图像大小相同的 RGB 图像。在实践场景中，该功能是通过一系列 Conv2dTranspose 转置卷积层来完成的，每个层都与 BatchNorm2d 层和 ReLu 激活层配对，输出数据会经过 tanh 函数，使其返回[-1,1]的数据范围内。

我们通过输入部分中设置的 nz、ngf 和 nc 来影响代码中的生成器结构。nz 是隐向量 z 的长度，ngf 与通过生成器传播的特征图的大小有关，nc 是输出图像中的通道数。

以下是生成器的代码实现：

```
def convt(in_channels, out_channels, kernel_size, stride=1, padding=0, pad_mode="pad"):
    """定义转置卷积层"""
    weight_init = Normal(mean=0, sigma=0.02)
    return nn.Conv2dTranspose(in_channels, out_channels,
                              kernel_size=kernel_size, stride=stride, padding=padding,
                              weight_init=weight_init, has_bias=False, pad_mode=pad_mode)


def bn(num_features):
    """定义 BatchNorm2d 层"""
    gamma_init = Normal(mean=1, sigma=0.02)
    return nn.BatchNorm2d(num_features=num_features, gamma_init=gamma_init)


class Generator(nn.Cell):
    """DCGAN 网络生成器"""
    def __init__(self):
        super(Generator, self).__init__()
```

```
        self.generator = nn.SequentialCell()
        self.generator.append(convt(nz, ngf * 8, 4, 1, 0))
        self.generator.append(bn(ngf * 8))
        self.generator.append(nn.ReLU())
        self.generator.append(convt(ngf * 8, ngf * 4, 4, 2, 1))
        self.generator.append(bn(ngf * 4))
        self.generator.append(nn.ReLU())
        self.generator.append(convt(ngf * 4, ngf * 2, 4, 2, 1))
        self.generator.append(bn(ngf * 2))
        self.generator.append(nn.ReLU())
        self.generator.append(convt(ngf * 2, ngf, 4, 2, 1))
        self.generator.append(bn(ngf))
        self.generator.append(nn.ReLU())
        self.generator.append(convt(ngf, nc, 4, 2, 1))
        self.generator.append(nn.Tanh())

    def construct(self, x):
        return self.generator(x)
```

实例化生成器，并打印出生成器的结构。

输入：

```
netG = Generator()
print(netG)
```

输出：

```
Generator<
  (generator): SequentialCell<
    (0): Conv2dTranspose<input_channels=100, output_channels=512, kernel_size=(4, 4),stride=(1, 1), pad_mode=pad, padding=0, dilation=(1, 1), group=1, has_bias=False,weight_init=<__main__.Normal object at 0x7fdaf3c14b50>, bias_init=zeros>
    (1): BatchNorm2d<num_features=512, eps=1e-05, momentum=0.09999999999999998, gamma=Parameter (name=1.gamma, shape=(512,), dtype=Float32, requires_grad=True), beta=Parameter (name=1.beta, shape=(512,), dtype=Float32, requires_grad=True), moving_mean=Parameter (name=1.moving_mean, shape=(512,), dtype=Float32, requires_grad=False), moving_variance=Parameter (name=1.moving_variance, shape=(512,), dtype=Float32, requires_grad=False)>
    (2): ReLU<>
    (3): Conv2dTranspose<input_channels=512, output_channels=256, kernel_size=(4, 4),stride=(2, 2), pad_mode=pad, padding=1, dilation=(1, 1), group=1, has_bias=False,weight_init=<__main__.Normal object at 0x7fdaf3bf8d90>, bias_init=zeros>
    (4): BatchNorm2d<num_features=256, eps=1e-05, momentum=0.09999999999999998, gamma=Parameter (name=4.gamma, shape=(256,), dtype=Float32, requires_grad=True), beta=Parameter (name=4.beta, shape=(256,), dtype=Float32, requires_grad=True), moving_mean=Parameter (name=4.moving_mean, shape=(256,), dtype=Float32, requires_grad=False), moving_variance=Parameter (name=4.moving_variance, shape=(256,), dtype=Float32, requires_grad=False)>
    (5): ReLU<>
    (6): Conv2dTranspose<input_channels=256, output_channels=128, kernel_size=(4, 4),stride=(2, 2), pad_mode=pad, padding=1, dilation=(1, 1), group=1, has_bias=False,weight_init=<__main__.Normal object at 0x7fdaf18340d0>, bias_init=zeros>
```

```
    (7): BatchNorm2d<num_features=128, eps=1e-05, momentum=0.09999999999999998,
gamma=Parameter (name=7.gamma, shape=(128,), dtype=Float32, requires_grad=True),
beta=Parameter (name=7.beta, shape=(128,), dtype=Float32, requires_grad=True),
moving_mean=Parameter (name=7.moving_mean, shape=(128,), dtype=Float32, requires_grad=False),
moving_variance=Parameter (name=7.moving_variance, shape=(128,), dtype=Float32,
requires_grad=False)>
```

步骤 3　生成判别器

如前所述，判别器 D 是一个二分类网络模型，输出判定该图像为真实图的概率。通过一系列的 Conv2d、BatchNorm2d 和 Leaky ReLU 层对其进行处理，最后通过 Sigmoid 激活函数得到最终概率。

DCGAN 论文提到，使用卷积而不是通过池化来进行下采样是一个好习惯，因为它可以让网络学习自己的池化特征。

判别器的代码实现如下：

输入：

```
def conv(in_channels, out_channels, kernel_size, stride=1, padding=0, pad_mode="pad"):
    '''定义卷积层'''
    weight_init = Normal(mean=0, sigma=0.02)
    return nn.Conv2d(in_channels, out_channels,
                     kernel_size=kernel_size, stride=stride, padding=padding,
                     weight_init=weight_init, has_bias=False, pad_mode=pad_mode)

class Discriminator(nn.Cell):
    """
    DCGAN 网络判别器
    """

    def __init__(self):
        super(Discriminator, self).__init__()
        self.discriminator = nn.SequentialCell()
        self.discriminator.append(conv(nc, ndf, 4, 2, 1))
        self.discriminator.append(nn.LeakyReLU(0.2))
        self.discriminator.append(conv(ndf, ndf * 2, 4, 2, 1))
        self.discriminator.append(bn(ndf * 2))
        self.discriminator.append(nn.LeakyReLU(0.2))
        self.discriminator.append(conv(ndf * 2, ndf * 4, 4, 2, 1))
        self.discriminator.append(bn(ndf * 4))
        self.discriminator.append(nn.LeakyReLU(0.2))
        self.discriminator.append(conv(ndf * 4, ndf * 8, 4, 2, 1))
        self.discriminator.append(bn(ndf * 8))
        self.discriminator.append(nn.LeakyReLU(0.2))
        self.discriminator.append(conv(ndf * 8, 1, 4, 1))
        self.discriminator.append(nn.Sigmoid())

    def construct(self, x):
        return self.discriminator(x)
```

输入：

```
netD = Discriminator()
print(netD)
```

输出：

```
Discriminator<
  (discriminator): SequentialCell<
    (0): Conv2d<input_channels=3, output_channels=64, kernel_size=(4, 4),stride=(2, 2),
pad_mode=pad, padding=1, dilation=(1, 1), group=1, has_bias=Falseweight_init=<__main__.Normal
object at 0x7fdaf17989d0>, bias_init=zeros, format=NCHW>
    (1): LeakyReLU<>
    (2): Conv2d<input_channels=64, output_channels=128, kernel_size=(4, 4),stride=(2, 2),
pad_mode=pad, padding=1, dilation=(1, 1), group=1, has_bias=Falseweight_init=<__main__.Normal
object at 0x7fdaf1798b90>, bias_init=zeros, format=NCHW>
    (3): BatchNorm2d<num_features=128, eps=1e-05, momentum=0.09999999999999998,
gamma=Parameter (name=3.gamma, shape=(128,), dtype=Float32, requires_grad=True),
beta=Parameter (name=3.beta, shape=(128,), dtype=Float32, requires_grad=True),
moving_mean=Parameter (name=3.moving_mean, shape=(128,), dtype=Float32, requires_grad=False),
moving_variance=Parameter (name=3.moving_variance, shape=(128,), dtype=Float32,
requires_grad=False)>
    (4): LeakyReLU<>
    (5): Conv2d<input_channels=128, output_channels=256, kernel_size=(4, 4),stride=(2, 2),
pad_mode=pad, padding=1, dilation=(1, 1), group=1, has_bias=Falseweight_init=<__main__.Normal
object at 0x7fdaf18ba350>, bias_init=zeros, format=NCHW>
    (6): BatchNorm2d<num_features=256, eps=1e-05, momentum=0.09999999999999998,
gamma=Parameter (name=6.gamma, shape=(256,), dtype=Float32, requires_grad=True),
beta=Parameter (name=6.beta, shape=(256,), dtype=Float32, requires_grad=True),
moving_mean=Parameter (name=6.moving_mean, shape=(256,), dtype=Float32, requires_grad=False),
moving_variance=Parameter (name=6.moving_variance, shape=(256,), dtype=Float32,
requires_grad=False)>
    (7): LeakyReLU<>
```

步骤 4　连接网络和损失函数

MindSpore 将损失函数、优化器等操作都封装到了 Cell 中，因为 GAN 结构上的特殊性，其损失是判别器和生成器的多输出形式，这就导致它和一般的分类网络不同。所以我们需要自定义 WithLossCell 类，将网络和 Loss 连接起来。

输入：

```
class WithLossCellG(nn.Cell):
    """连接生成器和损失"""
    def __init__(self, netD, netG, loss_fn):
        super(WithLossCellG, self).__init__(auto_prefix=True)
        self.netD = netD
        self.netG = netG
        self.loss_fn = loss_fn

    def construct(self, latent_code):
        """构建生成器损失计算结构"""
        ones = ops.Ones()
```

```
            fake_data = self.netG(latent_code)
            out = self.netD(fake_data)
            label = ones(out.shape, mstype.float32)
            loss = self.loss_fn(out, label)
            return loss

class WithLossCellD(nn.Cell):
    """连接判别器和损失"""
    def __init__(self, netD, netG, loss_fn):
            super(WithLossCellD, self).__init__(auto_prefix=True)
            self.netD = netD
            self.netG = netG
            self.loss_fn = loss_fn

    def construct(self, real_data, latent_code):
            """构建判别器损失计算结构"""
            ones = ops.Ones()
            zeros = ops.Zeros()

            out1 = self.netD(real_data)
            label1 = ones(out1.shape, mstype.float32)
            loss1 = self.loss_fn(out1, label1)

            fake_data = self.netG(latent_code)
            fake_data = ops.stop_gradient(fake_data)
            out2 = self.netD(fake_data)
            label2 = zeros(out2.shape, mstype.float32)
            loss2 = self.loss_fn(out2, label2)
            return loss1 + loss2
```

步骤 5　损失函数和优化器

当定义了 D 和 G 后，接下来将使用 MindSpore 中定义的二进制交叉熵损失函数 BCELoss ，为 D 和 G 加上损失函数和优化器。这里设置了两个单独的优化器，一个用于 D，另一个用于 G。这两个都是 lr = 0.0002 和 beta1 = 0.5 的 Adam 优化器。为此将真实标签定义为 1，将虚假标签定义为 0，该标签在分别计算 D 和 G 的损失时使用。为了跟踪生成器的学习进度，将生成一批固定的遵循高斯分布的隐向量 fixed_noise。在训练的过程中，定期将 fixed_noise 输入到 G 中，可以看到隐向量生成的图像。

输入：

```
# 定义损失函数
criterion = nn.BCELoss(reduction='mean')

# 创建一批隐向量用来观察 G
np.random.seed(1)
fixed_noise = Tensor(np.random.randn(64, nz, 1, 1),dtype=mstype.float32)
```

```
# 为生成器和判别器设置优化器
optimizerD = nn.Adam(netD.trainable_params(), learning_rate=lr, beta1=beta1)
optimizerG = nn.Adam(netG.trainable_params(), learning_rate=lr, beta1=beta1)
```

1.3.2.4 训练模型

训练分为两个主要部分：训练判别器和训练生成器。

3、训练判别器的目的是最大程度地提高判别图像真伪的概率。按照 Goodfellow 的方法，是希望通过提高其随机梯度来更新判别器，所以我们要最大化损失函数的值。

4、训练生成器，如 DCGAN 论文所述，我们希望通过最小化损失函数来训练生成器，以产生更好的虚假图像。

在这两个部分中，分别获取训练过程中的损失，并在每个周期结束时进行统计，将 fixed_noise 批量推送到生成器中，以直观地跟踪 G 的训练进度。

步骤 1 定义 DCGAN 网络。

```
class DCGAN(nn.Cell):
    def __init__(self, myTrainOneStepCellForD, myTrainOneStepCellForG):
        super(DCGAN, self).__init__(auto_prefix=True)
        self.myTrainOneStepCellForD = myTrainOneStepCellForD
        self.myTrainOneStepCellForG = myTrainOneStepCellForG

    def construct(self, real_data, latent_code):
        output_D = self.myTrainOneStepCellForD(real_data, latent_code).view(-1)
        netD_loss = output_D.mean()
        output_G = self.myTrainOneStepCellForG(latent_code).view(-1)
        netG_loss = output_G.mean()
        return netD_loss, netG_loss
```

步骤 2 实例化生成器和判别器的 WithLossCell 和 TrainOneStepCell

```
# 实例化 WithLossCell
netD_with_criterion = WithLossCellD(netD, netG, criterion)
netG_with_criterion = WithLossCellG(netD, netG, criterion)

# 实例化 TrainOneStepCell
myTrainOneStepCellForD = nn.TrainOneStepCell(netD_with_criterion, optimizerD)
myTrainOneStepCellForG = nn.TrainOneStepCell(netG_with_criterion, optimizerG)
```

步骤 3 循环训练网络，每经过 50 次迭代，就收集生成器和判别器的损失，以便于后面绘制训练过程中损失函数的图像。

输入：

```
# 实例化 DCGAN 网络
dcgan = DCGAN(myTrainOneStepCellForD, myTrainOneStepCellForG)
dcgan.set_train()
```

```
#创建迭代器
data_loader = data.create_dict_iterator(output_numpy=True, num_epochs=num_epochs)
G_losses = []
D_losses = []
iters = 0
image_list = []

# 开始循环训练
print("Starting Training Loop...")

for epoch in range(num_epochs):
    # 为每轮训练读入数据
    for i, d in enumerate(data_loader):
        real_data = Tensor(d['image'])
        latent_code = Tensor(d["latent_code"])
        netD_loss, netG_loss = dcgan(real_data, latent_code)
        if i % 50 == 0:
            # 输出训练记录
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f'% (epoch + 1, num_epochs, i, size,
netD_loss.asnumpy(), netG_loss.asnumpy()))
        D_losses.append(netD_loss.asnumpy())
        G_losses.append(netG_loss.asnumpy())
        if (iters % 100) == 0 or ((epoch == num_epochs) and (i == size-1)):
            img = netG(fixed_noise)
            image_list.append(img)
        iters += 1
```

输出：

```
[1/10][0/54]    Loss_D: 2.4556  Loss_G: 3.0041
[1/10][50/54]   Loss_D: 0.1433  Loss_G: 7.6950
[2/10][0/54]    Loss_D: 0.1365  Loss_G: 8.8596
[2/10][50/54]   Loss_D: 0.1310  Loss_G: 8.7747
[3/10][0/54]    Loss_D: 0.0504  Loss_G: 10.5596
[3/10][50/54]   Loss_D: 0.8163  Loss_G: 8.0759
[4/10][0/54]    Loss_D: 0.3023  Loss_G: 5.7986
[4/10][50/54]   Loss_D: 0.5194  Loss_G: 4.3674
[5/10][0/54]    Loss_D: 0.3352  Loss_G: 3.4594
[5/10][50/54]   Loss_D: 0.4974  Loss_G: 4.4283
[6/10][0/54]    Loss_D: 0.3801  Loss_G: 4.0097
```

1.3.2.5 验证结果可视化

运行下面代码，描绘 D 和 G 损失与训练迭代的关系图：

输入：

```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G",color='blue')
plt.plot(D_losses,label="D",color='orange')
```

```
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

输出：

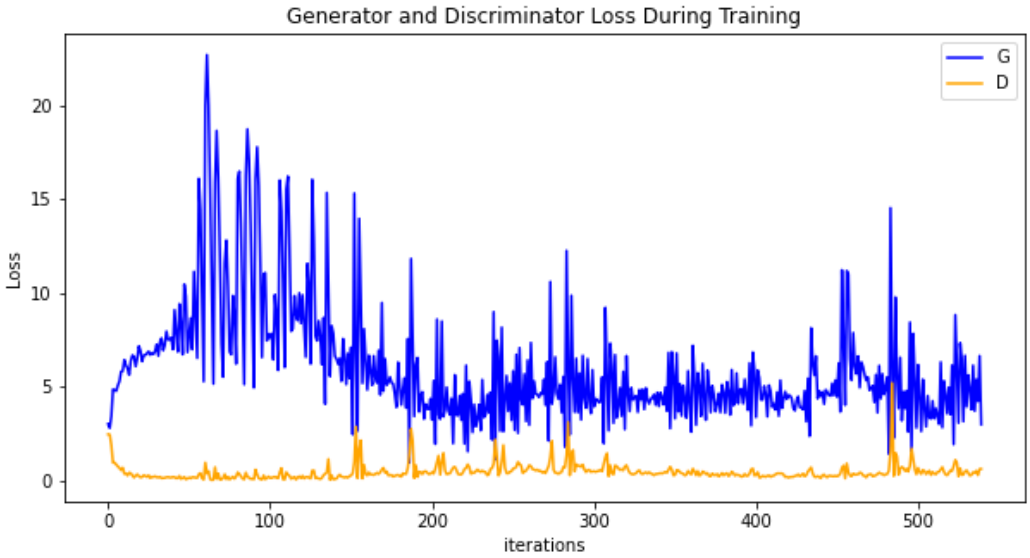

图1-5 Loss 变化图

1.3.3 实验小结

本实验搭建了 DCGAN 网络完成图片生成任务。

华为认证 AI 系列教程

HCIP-AI-MindSpore Developer

自然语言处理原理与应用

实验指导手册

版本：1.0

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

华为认证体系介绍

华为认证是华为公司基于“平台+生态”战略，围绕“云-管-端”协同的新ICT技术架构，打造的覆盖ICT（Information and Communications Technology 信息通信技术）全技术领域的认证体系，包含ICT技术架构与应用认证、云服务与平台认证两类认证。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

华为认证覆盖ICT全领域，符合ICT融合的技术趋势，致力于提供领先的人才培养体系和认证标准，培养数字化时代新型ICT人才，构建良性ICT人才生态。

HCIP-AI-MindSpore Developer V1.0 定位于培养和认证具备使用华为深度学习框架MindSpore进行人工智能开发能力的算法工程师。

通过HCIP-AI-MindSpore Developer V1.0认证，您将掌握MindSpore和MindSpore Lite的架构与特性相关知识，理解MindSpore网络迁移、分布式训练、端云侧推理与部署等相关特性；具备使用MindSpore完成图像处理任务和自然语言处理任务的能力；能够胜任人工智能算法工程师、深度学习工程师、图像算法工程师、自然语言处理算法工程师等岗位。


Huawei Certification
Huawei Certified ICT Expert
Huawei Certified ICT Professional
Huawei Certified ICT Associate
HCIE
HCIP
HCIA
Cloud Service & Platform
云服务与平台
ICT Infrastructure
ICT技术架构与应用
HarmonyOS(To Be)
IoT
openGauss (To Be)
GaussDB
openEuler
Cloud(Cloud Computing + Cloud Service)
Intelligent Driving (To Be)
Big Data
Kunpeng Application Developer
AI
Storage
Datacom
Transmission
Intelligent Computing
WLAN
Access
Enterprise Communication
Security
LTE
Intelligent Vision
SDN
5G
Data Center

前 言

简介

本书为 HCIP-AI-MindSpore Developer 认证培训教程，适用于准备参加 HCIP-AI-MindSpore Developer 考试的学员或者希望了解 AI 基础知识、图像处理与自然语言处理方法与实践、MindSpore 开发框架架构与应用的读者。

内容描述

本实验指导书共包含 3 个实验，目标是基于 MindSpore 进行模型开发完成自然语言处理相关任务。

- 实验一为使用字符级 RNN（Recurrent Neural Network）对单词进行名称分类。
- 实验二为使用字符级 RNN 生成不同语言的名称。
- 实验三为基于 Tramsformer 模型进行文本翻译。

读者知识背景

本课程为华为认证基础课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有基本的 AI 知识背景，同时熟悉华为 MindSpore 框架，了解基本 AI 开发流程。

实验环境说明

为了满足 MindSpore 框架下的模型训练算力需求，我们推荐使用华为云 ModelArts 平台进行实验。

准备实验环境

本实验建议使用 MindSpore 1.3 及以上版本，推荐实验一、二于 Windows 本机上使用 CPU 运行，实验三于 ModelArts Notebook 中使用 Ascend 运行。

目录

1 自然语言处理实验

1.1 实验介绍

1.1.1 关于本实验

本实验为课程中《自然语言处理原理与应用》对应的实践部分，旨在通过代码实操加深对自然语言处理 AI 模型的的理解，并掌握基于 MindSpore 的开发 NLP（Natural Language Processing）应用的能力。

1.1.2 实验目的

- 使用 MindSpore 开展自然语言处理典型任务，包括分类、生成与翻译任务。

1.1.3 环境准备

本实验在华为云 ModelArts 上进行，使用 Notebook 开发环境，基于 MindSpore-Ascend 环境进行。具体的购买和打开操作请参考《HCIP-AI-MindSpore Developer V1.0 实验环境搭建指南》，此处不重复赘述。

首先在启动界面新建 MindSpore 环境的 Notebook。

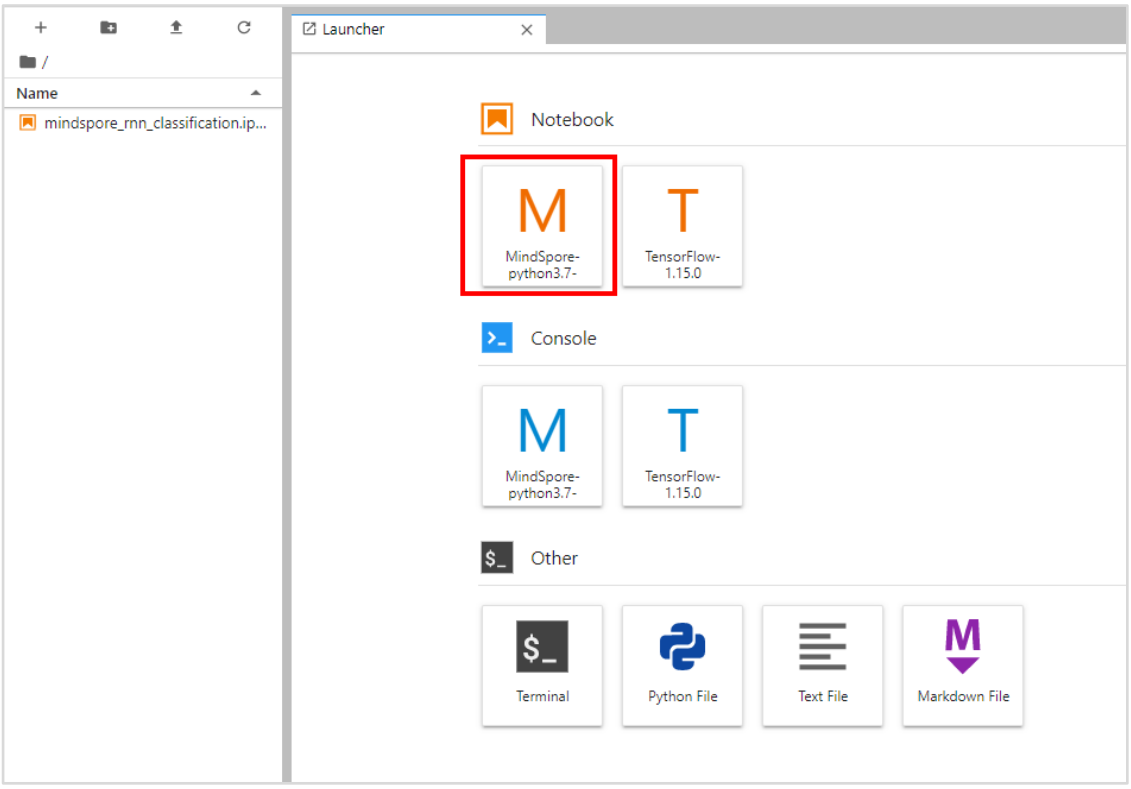

图1-1 新建 Notebook

以下的实验步骤均在 Notebook 中进行，Notebook 的使用方法主要是添加代码块并运行，如下图所示。

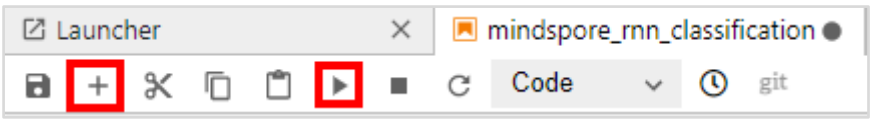

图1-2 Notebook 使用

1.2 实验一：使用字符级 RNN 分类名称

1.2.1 概述

循环神经网络是一类以序列（sequence）数据为输入，在序列的演进方向进行递归（recursion）且所有节点（循环单元）按链式连接的递归神经网络（recursive neural network），常用于 NLP 领域当中来解决序列化数据的建模问题。

本教程我们将建立和训练基本的字符级 RNN 模型对单词进行分类，以帮助理解循环神经网络原理。实验中，我们将训练来自 18 种语言的数千种姓氏，并根据拼写内容预测名称的来源。

1.2.2 设定运行模式

输入：

```
from mindspore import context
context.set_context(mode=context.PYNATIVE_MODE, device_target="Ascend")
```

- 说明：实验采用 PyNative 模式，基于 Ascend 环境，通过 context 模块中的方法进行设定。

1.2.3 数据处理

1.2.3.1 准备数据

步骤 1 数据下载

如前所述，本实验所用的数据集为 18 种不同语言的姓氏，我们基于这些数据开发训练模型实现对于姓氏名称的语言类别的分类判断。在 Notebook 中执行如下命令进行数据下载：

```
!wget -N https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap5/data.zip
!unzip -n data.zip
```

部分输出：

```
Archive:  data.zip
   creating: data/
  inflating: data/eng-fra.txt
   creating: data/names/
  inflating: data/names/Arabic.txt
  inflating: data/names/Chinese.txt
  inflating: data/names/Czech.txt
  inflating: data/names/Dutch.txt
  inflating: data/names/English.txt
  inflating: data/names/French.txt
  inflating: data/names/German.txt
  inflating: data/names/Greek.txt
  inflating: data/names/Irish.txt
  inflating: data/names/Italian.txt
  inflating: data/names/Japanese.txt
  inflating: data/names/Korean.txt
  inflating: data/names/Polish.txt
  inflating: data/names/Portuguese.txt
  inflating: data/names/Russian.txt
  inflating: data/names/Scottish.txt
  inflating: data/names/Spanish.txt
  inflating: data/names/Vietnamese.txt
```

图1-3 数据文件内容

从结果中可以看到，data/names 文件夹为我们用到的数据，包括了阿拉伯文，汉语在内的不同语言的姓氏。

- 思考：如何查看这些文件中的内容呢？（答案在下一小节）

1.2.3.2 数据处理

步骤 1 数据查看

方法一，在左侧的文件预览窗口我们可以直接打开 txt 文件：

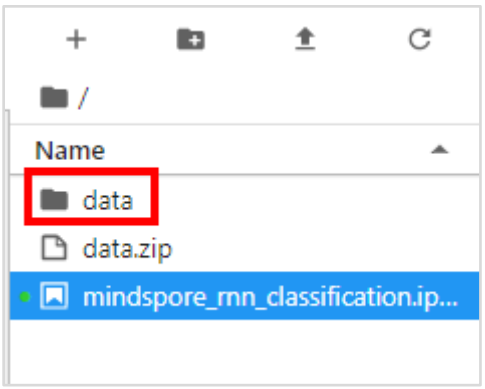

图1-4 文件预览窗口

方法二，可以用代码打开 txt 文件并打印部分内容：

```
f = open("./data/names/Chinese.txt")
for i in range(10):
    print(f.readline().strip())
f.close()
```

输出：

```
Ang
Au-Yong
Bai
Ban
Bao
Bei
Bian
Bui
Cai
Cao
```

图1-5 输出结果

步骤 2 导入数据处理工具模块

```
from io import open
import glob
import os
import unicodedata
import string
```

步骤 3 数据导入与处理

定义 find_files 函数，查找符合通配符要求的文件。

```
def find_files(path):
    return glob.glob(path)

print(find_files('data/names/*.txt'))
```

通过该函数，我们可以遍历所有的 txt 数据文件。这里 glob 提供了文件检索的作用。输出结果：

图1-6 输出结果

定义 unicode_to_ascii 函数，将 Unicode 转换为 ASCII 码。

```
all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)

def unicode_to_ascii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )

print(unicode_to_ascii('Bélanger'))
```

- 原理：通过 unicodedata.normalize 方法实现字符的标准化。排除“Mn”类别的字符，即消去 Bélanger 中 e 的上标。并与 all_letters 字母表对应。
- 思考 1：n_letters 值为多少？可以先查看 string.ascii_letters 的内容。
- 思考 2：为什么要转化为 ASCII 码？（两个思考题答案如下）

我们来看一个小测试：

```
print("\u00f1")
print("n\u0303")
```

输出：

图1-7 输出结果

其中\u 代表 Unicode 编码，可以看到不同的 Unicode 编码可能输出同样的字符结果，因此我们需要转化为统一的字符集合内进行分类任务的比较。

步骤 4 读取文件并转换

定义 read_lines 函数，读取文件，并将文件每一行内容的编码转换为 ASCII。

```
def read_lines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split('\n')
```

```
    return [unicode_to_ascii(line) for line in lines]
```

定义 category_lines 字典和 all_categories 列表。其中 category_lines 的 key 为语言的类别，value 为名称的列表。all_categories 列表包含所有语言的种类。

```
category_lines = {}
all_categories = []

for filename in find_files('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    all_categories.append(category)
    lines = read_lines(filename)
    category_lines[category] = lines

n_categories = len(all_categories)
```

将语言为 Italian，内容为前 5 行的数据进行打印显示。

```
print(category_lines['French'][:5])
```

输出：

```
['Abel', 'Abraham', 'Adam', 'Albert', 'Allard']
```

图1-8 输出结果

步骤 5 进行向量化

因为字符无法进行数学运算，所以需要将名称转变为向量。

为了表示单个字母，我们使用大小为<1 x n_letters>的 one-hot 向量，因为将离散型特征使用 one-hot 编码，会让特征之间的距离计算更加合理。

one-hot 向量用 0 填充，但当前字母的索引处的数字为 1，例如 "b" = <0 1 0 0 0 ...>。

为了组成单词，我们将其中的一些向量连接成 2D 矩阵<line_length x 1 x n_letters>。

导入模块：

```
import numpy as np

from mindspore import Tensor
from mindspore import dtype as mstype
```

定义 letter_to_index 函数，从 all_letters 列表中查找字母索引。

```
def letter_to_index(letter):
    return all_letters.find(letter)
```

定义 letter_to_tensor 函数，将字母转换成维度是<1 x n_letters>的 one-hot 向量。

```
def letter_to_tensor(letter):
    tensor = Tensor(np.zeros((1, n_letters)),mstype.float32)
    tensor[0,letter_to_index(letter)] = 1.0
    return tensor
```

定义 line_to_tensor 函数，将一行转化为<line_length x 1 x n_letters>的 one-hot 向量。

```
def line_to_tensor(line):
    tensor = Tensor(np.zeros((len(line), 1, n_letters)),mstype.float32)
    for li, letter in enumerate(line):
        tensor[li,0,letter_to_index(letter)] = 1.0
    return tensor
```

分别将字母 A 和单词 Alex 转换为 one-hot 向量，并打印显示。

```
print(letter_to_tensor('A'))
print(line_to_tensor('Alex').shape)
```

输出：

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0.]]
(4, 1, 57)
```

图1-9 输出结果

1.2.4 创建网络

完成数据的处理之后，我们实现了将名称字符的向量化。接着我们进行模型的搭建，本实验采用 RNN 模型实现名称分类。

RNN 网络结构如下图所示：

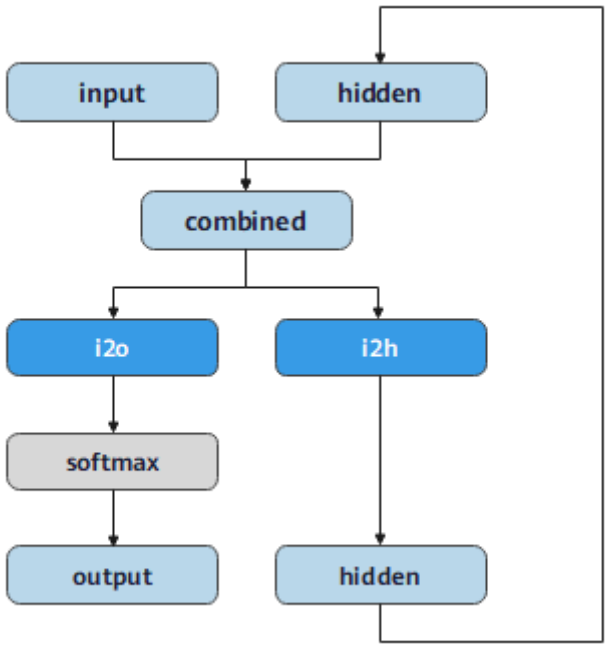

图1-10 网络结构

创建的 RNN 网络有 i2o 和 i2h 两个线性层，它们在输入 input 和隐藏状态 hidden 下运行，在线性层 i2o 的输出之后是 LogSoftmax 层。

1.2.4.1 模型搭建代码

```
from mindspore import nn, ops

class RNN(nn.Cell):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Dense(input_size + hidden_size, hidden_size)
        self.i2o = nn.Dense(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(axis=1)

    def construct(self, input, hidden):
        op = ops.Concat(axis=1)
        combined = op((input, hidden))
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return Tensor(np.zeros((1, self.hidden_size)),mstype.float32)

n_hidden = 128
rnn = RNN(n_letters, n_hidden, n_categories)
```

要运行此网络，我们需要输入代表当前字母的 one-hot 向量，以及上一个字母输出的隐藏状态（将隐藏状态初始化为 0）。此网络将输出属于每种语言的概率和下一个字母需要输入的隐藏状态。

思考：使用 Logsoftmax 的意义是?

Softmax 计算中由于指数运算可能出现超出数值范围的情况，Logsoftmax 能解决该问题。

1.2.4.2 模型计算测试

```
input = letter_to_tensor('A')
hidden = Tensor(np.zeros((1, n_hidden)), mstype.float32)
output, next_hidden = rnn(input, hidden)
```

为了提高效率，避免在每一步中都创建一个新向量，因此将使用 line_to_tensor 而不是 letter_to_tensor，同时采取切片操作。

```
input = line_to_tensor('Albert')
hidden = Tensor(np.zeros((1, n_hidden)), mstype.float32)
output, next_hidden = rnn(input[0], hidden)
print(output)
```

结果为：

```
[[-2.893743  -2.8924723 -2.8802812 -2.884685  -2.8995385 -2.8837736
  -2.9037814 -2.8999913 -2.8988907 -2.894345  -2.901554  -2.8825603
  -2.8956528 -2.8768175 -2.8908525 -2.8856812 -2.8936315 -2.8692   ]]
```

图1-11 输出结果

可以看到，输出为<1 x n_categories>形式的向量，其中每个数字都代表了分类的可能性。

1.2.5 模型训练

1.2.5.1 训练准备

定义 category_from_output 函数，获得网络模型输出的最大值，也就是分类类别概率为最大的类别。

```
def category_from_output(output):
    topk = ops.TopK(sorted=True)
    top_n, top_i = topk(output, 1)
    print(top_n,top_i)
    category_i = top_i.asnumpy().item(0)
    return all_categories[category_i], category_i

print(category_from_output(output))
```

通过 random_training 函数随机选择一种语言和其中一个名称作为训练数据。

```
import random

# 随机选择
def random_choice(l):
    return l[random.randint(0, len(l) - 1)]

# 随机选择一种语言和一个名称
def random_training():
    category = random_choice(all_categories)
    line = random_choice(category_lines[category])
    category_tensor = Tensor([all_categories.index(category)], mstype.int32)
    line_tensor = line_to_tensor(line)
    return category, line, category_tensor, line_tensor

# 随机选 10 组
for i in range(10):
    category, line, category_tensor, line_tensor = random_training()
    print('category =', category, '/ line =', line)
```

输出：

```
category = Czech / line = Kanak
category = Greek / line = Letsos
category = Irish / line = Mulryan
category = Korean / line = Ahn
category = French / line = Tolbert
category = Greek / line = Kalogeria
category = Chinese / line = Chieu
category = Vietnamese / line = Lac
category = Portuguese / line = Paredes
category = Vietnamese / line = Tieu
```

图1-12 输出结果

1.2.5.2 训练网络

定义 NLLLoss 损失函数：

```
from mindspore.ops import functional as F

class NLLLoss(nn.loss.loss._Loss):
    def __init__(self, reduction='mean'):
        super(NLLLoss, self).__init__(reduction)
        self.one_hot = ops.OneHot()
        self.reduce_sum = ops.ReduceSum()

    def construct(self, logits, label):
        label_one_hot = self.one_hot(label, F.shape(logits)[-1], F.scalar_to_array(1.0),
F.scalar_to_array(0.0))
        loss = self.reduce_sum(-1.0 * logits * label_one_hot, (1,))
        return self.get_loss(loss)
```

每个循环训练将会执行下面几个步骤：

- 创建输入和目标向量。
- 初始化隐藏状态。
- 学习每个字母并保存下一个字母的隐藏状态。
- 比较最终输出与目标值。
- 反向传播梯度变化。
- 返回输出和损失值。

MindSpore 将损失函数，优化器等操作都封装到了 Cell 中，但是本教程的 rnn 网络需要循环一个序列长度之后再求损失，所以我们需要自定义 WithLossCellRnn 类，将网络和 Loss 连接起来。

```
class WithLossCellRnn(nn.Cell):
    def __init__(self, backbone, loss_fn):
        super(WithLossCellRnn, self).__init__(auto_prefix=True)
        self._backbone = backbone
```

```
        self._loss_fn = loss_fn

    def construct(self, line_tensor, hidden, category_tensor):
        for i in range(line_tensor.shape[0]):
            output, hidden = self._backbone(line_tensor[i], hidden)
        return self._loss_fn(output, category_tensor)
```

创建优化器、WithLossCellRnn 实例和 TrainOneStepCell 训练网络。

```
rnn_cf = RNN(n_letters, n_hidden, n_categories)
optimizer = nn.Momentum(filter(lambda x: x.requires_grad, rnn_cf.get_parameters()), 0.001, 0.9)
criterion=NLLLoss()
net_with_criterion = WithLossCellRnn(rnn_cf, criterion)
net = nn.TrainOneStepCell(net_with_criterion, optimizer)
net.set_train()

# 训练网络
def train(category_tensor, line_tensor):
    hidden = rnn_cf.initHidden()
    loss = net(line_tensor, hidden, category_tensor)
    for i in range(line_tensor.shape[0]):
        output, hidden = rnn_cf(line_tensor[i], hidden)
    return output, loss
```

为了跟踪网络模型训练过程中的耗时，定义 time_since 函数，用来计算训练运行的时间，方便我们持续看到训练的整个过程。

```
import time
import math

n_iters = 10000
print_every = 500
plot_every = 100
current_loss = 0
all_losses = []

def time_since(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)
```

通过 print_every(500)次迭代就打印一次，分别打印迭代次数、迭代进度、迭代所用时间、损失值、语言名称、预测语言类型、是否正确，其中通过√、×来表示模型判断的正误。同时，根据 plot_every 的值计算平均损失，将其添加进 all_losses 列表，以便于后面绘制训练过程中损失函数的图像。

```
start = time.time()

for iter in range(1, n_iters + 1):
```

```
    category, line, category_tensor, line_tensor = random_training()
    output, loss = train(category_tensor, line_tensor)
    current_loss += loss

    # 分别打印迭代次数、迭代进度、迭代所用时间、损失值、语言名称、预测语言类型、是否正确
    if iter % print_every == 0:
        guess, guess_i = category_from_output(output)
        correct = '✓' if guess == category else '✗ (%s)' % category
        print('%d %d%% (%s) %s %s / %s %s' % (iter, iter / n_iters * 100, time_since(start),
loss.asnumpy(), line, guess, correct))

    # 将 loss 的平均值添加至 all_losses
    if iter % plot_every == 0:
        all_losses.append((current_loss / plot_every).asnumpy())
        current_loss = 0
```

输出：

```
500 5% (0m 56s) 2.8811955 Cheng / Korean ✗ (Chinese)
1000 10% (1m 47s) 2.8138192 Glennon / Russian ✗ (English)
1500 15% (2m 37s) 2.7881339 Marugo / Italian ✗ (Japanese)
2000 20% (3m 29s) 2.9860206 O'Meara / Japanese ✗ (Irish)
2500 25% (4m 18s) 2.857955 Wen / Irish ✗ (Chinese)
3000 30% (5m 12s) 2.411959 O'Hannigain / Irish ✓
3500 35% (6m 28s) 2.1828568 Hishikawa / Japanese ✓
4000 40% (7m 17s) 2.5861049 Kennedy / Irish ✓
4500 45% (9m 45s) 3.1115925 La / Japanese ✗ (Vietnamese)
5000 50% (12m 36s) 2.811106 Cavey / Russian ✗ (French)
5500 55% (13m 35s) 2.8926034 Christodoulou / Vietnamese ✗ (Greek)
6000 60% (14m 22s) 2.5833995 Nanami / Italian ✗ (Japanese)
6500 65% (15m 6s) 2.9273236 Sissons / Greek ✗ (English)
7000 70% (15m 54s) 2.8183262 Houttum / Vietnamese ✗ (Dutch)
7500 75% (16m 41s) 3.0385728 Winograd / Arabic ✗ (Polish)
8000 80% (17m 31s) 3.0026562 Morales / Greek ✗ (Spanish)
8500 85% (18m 21s) 2.670665 Roach / Vietnamese ✗ (Irish)
9000 90% (19m 8s) 3.0125608 Kendrick / Polish ✗ (English)
9500 95% (19m 58s) 2.8149955 Kazmier / German ✗ (Czech)
10000 100% (20m 46s) 2.3972077 Chin / Irish ✗ (Korean)
```

图1-13 训练输出

1.2.6 结果处理查看

步骤 1 损失值绘制

从 all_losses 绘制网络模型学习过程中每个 step 得到的损失值，可显示网络学习情况。

```
%matplotlib inline
import matplotlib.pyplot as plt
from matplotlib import ticker
```

```
plt.figure()
plt.plot(all_losses)
```

输出：

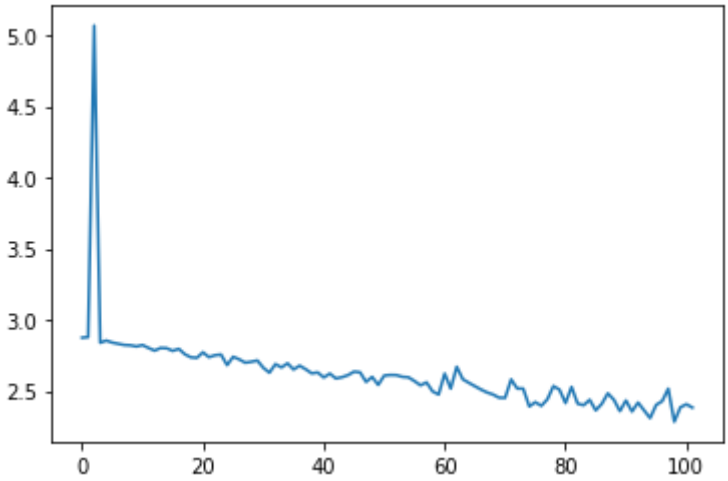

图1-14 输出结果

步骤 2 评估结果

为了查看网络在不同分类上的表现，我们将创建一个混淆矩阵，行坐标为实际语言，列坐标为预测的语言。为了计算混淆矩阵，使用 evaluate()函数进行模型推理。

```
# 在混淆矩阵中记录正确预测
confusion = Tensor(np.zeros((n_categories, n_categories)), mstype.float32)
n_confusion = 1000

# 模型推理
def evaluate(line_tensor):
    hidden = rnn_cf.initHidden()
    for i in range(line_tensor.shape[0]):
        output, hidden = rnn_cf(line_tensor[i], hidden)
    return output

# 运行样本，并记录正确的预测
for i in range(n_confusion):
    category, line, category_tensor, line_tensor = random_training()
    output = evaluate(line_tensor)
    guess, guess_i = category_from_output(output)
    category_i = all_categories.index(category)
    confusion[category_i, guess_i] += 1

for i in range(n_categories):
    confusion[i] / Tensor(np.sum(confusion[i].asnumpy()), mstype.float32)
```

使用 matplotlib 绘制混淆矩阵的图像。

```
# 绘制图表
```

```
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(confusion.asnumpy())
fig.colorbar(cax)

# 设定轴
ax.set_xticklabels([''] + all_categories, rotation=90)
ax.set_yticklabels([''] + all_categories)

# 在坐标处添加标签
ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

plt.show()
```

输出：

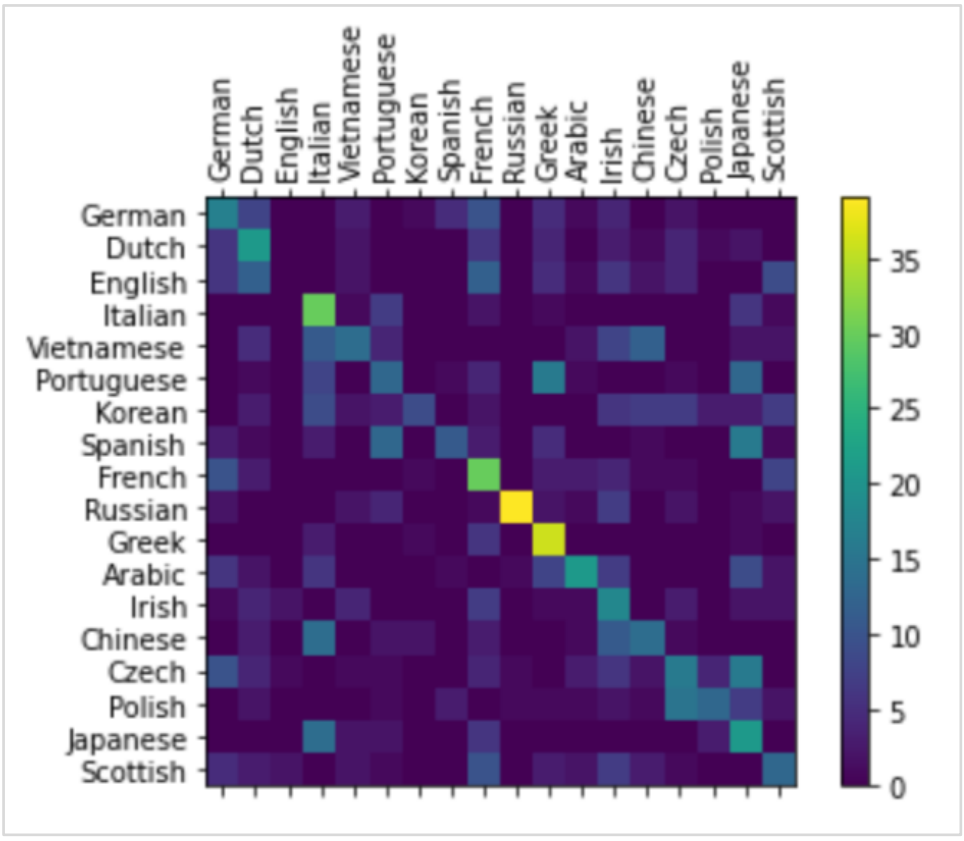

图1-15 输出结果

1.3 实验二：使用字符级 RNN 生成名称

1.3.1 概述

本章节中，我们将通过反向操作来生成不同语言的名称。这里仍通过编写由线性层结构构建出的小型 RNN 网络模型来实现目标。此次与上一章节最大的区别在于，不是通过输入名称中的所有字母来预测分类，而是输入一个分类类别，然后一次输出一个字母，这种用于预测字符来形成一个单词的方法通常称为“语言模型”。

1.3.2 关闭之前运行任务

本章前面的步骤与实验一中类似，如果在实验一中的 Notebook 环境的基础上进行实验，需要关闭上一章节的运行任务。点击左侧窗口的按键，如下图所示。

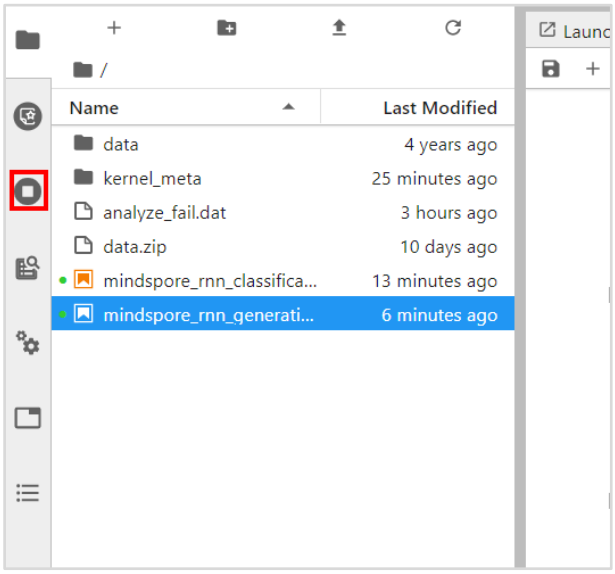

图1-16 运行任务查看

再点击“SHUT DOWN”。

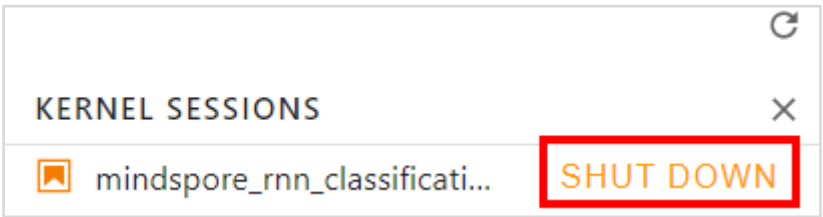

图1-17 关闭运行任务

1.3.3 设定运行模式

本教程我们在 Ascend 环境下，使用 PyNative 模式运行实验。

```
from mindspore import context

context.set_context(mode=context.PYNATIVE_MODE, device_target="Ascend")
```

1.3.4 数据处理

1.3.4.1 准备数据

输入：

```
!wget -NP ./ https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap5/data.zip
!unzip ./data.zip
```

如果在实验一中的 Notebook 环境的基础上进行实验，可以省略这一环节。

1.3.4.2 处理数据

步骤 1 导入模块

```
import os
import glob
import string
import unicodedata
from io import open
```

步骤 2 数据导入与处理

定义 find_files 函数，查找符合通配符要求的文件。

```
def find_files(path):
    return glob.glob(path)

print(find_files('data/names/*.txt'))
```

输出：

```
['data/names/German.txt', 'data/names/Dutch.txt', 'data/names/English.txt', 'data/names/Italian.txt', 'data/names/Vietnamese.txt', 'data/names/Portuguese.txt', 'data/names/Korean.txt', 'data/names/Spanish.txt', 'data/names/French.txt', 'data/names/Russian.txt', 'data/names/Greek.txt', 'data/names/Arabic.txt', 'data/names/Irish.txt', 'data/names/Chinese.txt', 'data/names/Czech.txt', 'data/names/Polish.txt', 'data/names/Japanese.txt', 'data/names/Scottish.txt']
```

图1-18 输出结果

定义 unicode_to_ascii 函数，将 Unicode 转换为 ASCII。

```
all_letters = string.ascii_letters + " .,;'-"
n_letters = len(all_letters) + 1

def unicode_to_ascii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )
```

```
print(unicode_to_ascii('Estéves'))
```

输出：

图1-19 输出结果

思考：为什么 n_letters 需要加一?

在后面的内容中我们看到，除了常规的字符以外，还需要定义结束符“EOS”。

步骤 3 读取文件并转换

定义 read_lines 函数，读取文件，并将文件每一行内容的编码转换为 ASCII。

```
def read_lines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split('\n')
    return [unicode_to_ascii(line) for line in lines]
```

定义 category_lines 字典和 all_categories 列表。

- category_lines：key 为语言的类别，value 为名称的列表。
- all_categories：所有语言的种类。

```
category_lines = {}
all_categories = []

for filename in find_files('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    all_categories.append(category)
    lines = read_lines(filename)
    category_lines[category] = lines

n_categories = len(all_categories)
```

将所有语言的数量和种类进行打印显示。

```
print('# categories:', n_categories, all_categories)
```

输出：

```
# categories: 18 ['German', 'Dutch', 'English', 'Italian', 'Vietnamese', 'Portuguese', 'Korean', 'Spanish', 'French', 'Russian', 'Greek', 'A
rabic', 'Irish', 'Chinese', 'Czech', 'Polish', 'Japanese', 'Scottish']
```

图1-20 输出结果

1.3.5 创建网络

该网络基于实验一教程中的 RNN 网络进行了扩展，附加了一个与输入 input 和隐藏状态 hidden 连接在一起的类别 category 张量。该张量与字母输入一样采用 one-hot 编码。

该网络的输出为下一个字母出现的概率，将最有可能出现的字母作为下一次迭代的输入 input。

与上一个网络结构略有不同，为了有更好的效果，在 output combined 层之后我们又添加了一个线性层 o2o。与此同时也新添加了一个 dropout 层，该层以一定的概率（此处为 0.1）将输入的部分随机归零。这一步骤通常用来防止过拟合。

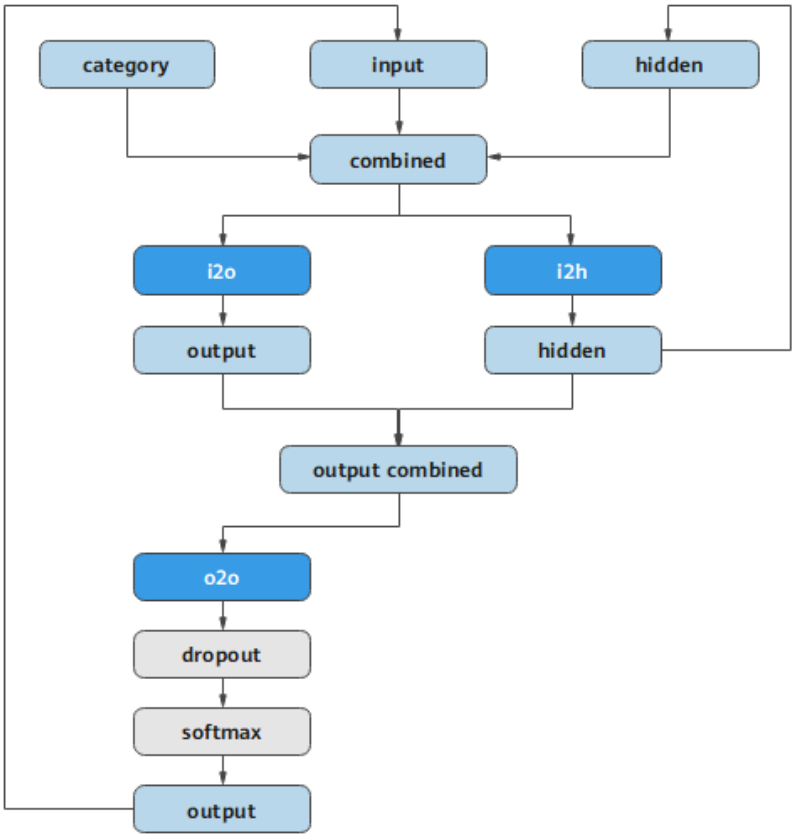

图1-21 网络结构

1.3.5.2 模型搭建代码

定义 RNN 网络基本结构。

```
import numpy as np

from mindspore import nn, ops, Tensor
from mindspore import dtype as mstype

class RNN(nn.Cell):
    """定义 RNN 网络"""
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Dense(n_categories + input_size + hidden_size, hidden_size)
        self.i2o = nn.Dense(n_categories + input_size + hidden_size, output_size)
        self.o2o = nn.Dense(hidden_size + output_size, output_size)
        self.dropout = nn.Dropout(0.1)
```

```
        self.softmax = nn.LogSoftmax(axis=1)

    # 构建 RNN 网络结构
    def construct(self, category, input, hidden):
        op = ops.Concat(axis=1)
        input_combined = op((category, input, hidden))
        hidden = self.i2h(input_combined)
        output = self.i2o(input_combined)
        output_combined = op((hidden, output))
        output = self.o2o(output_combined)
        output = self.dropout(output)
        output = self.softmax(output)
        return output, hidden

    # 初始化隐层状态
    def initHidden(self):
        return Tensor(np.zeros((1, self.hidden_size)),mstype.float32)
```

- 思考：能否模仿实验一中 1.2.4.2 小节的测试代码对该网络进行测试?

1.3.6 模型训练

1.3.6.1 训练准备

通过 random_training_pair 函数随机选择一种语言和其中一个名称作为训练数据。

```
import random

# 随机选择
def random_choice(l):
    return l[random.randint(0, len(l) - 1)]

# 随机选择一种语言和一个名称
def random_training_pair():
    category = random_choice(all_categories)
    line = random_choice(category_lines[category])
    return category, line
```

对于训练集中的每个名称，该网络的输入为：(category, current letter, hidden state)，输出为：(next letter, next hidden state)。因此对于每个训练集，我们都需要 categoryTensor（代表种类的 one-hot 张量），用于输入的 inputTensor（由首字母到尾字母（不包括 EOS）组成的 one-hot 矩阵）和用于输出的 targetTensor（由第二个字母到尾字母（包括 EOS）组成的张量）。

因为我们需要预测当前字母所对应的下一个字母，所以需要拆分连续字母来组成字母对。例如：对于"ABCD<EOS>"，我们将创建出('A', 'B')，('B', 'C')，('C', 'D')，('D', 'EOS')字母对。

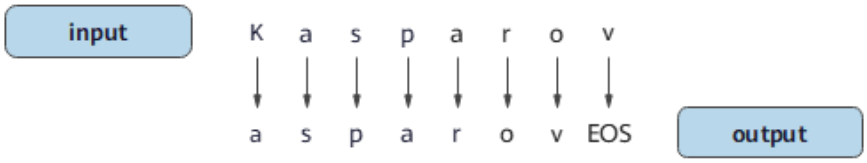

图1-22 生成流程

我们在训练时会持续将 category 张量传输至网络中，该张量维度为<1 x n_categories>的 one-hot 张量。

定义 category_to_tensor 函数将类别转换成维度为<1 x n_categories>的 one-hot 张量。

```
def category_to_tensor(category):
    li = all_categories.index(category)
    tensor = Tensor(np.zeros((1, n_categories)),mstype.float32)
    tensor[0,li] = 1.0
    return tensor
```

定义 input_to_tensor 函数，将输入转换成一个由首字母到尾字母（不包括 EOS）组成的 one-hot 矩阵。

```
def input_to_tensor(line):
    tensor = Tensor(np.zeros((len(line), 1, n_letters)), mstype.float32)
    for li in range(len(line)):
        letter = line[li]
        tensor[li,0,all_letters.find(letter)] = 1.0
    return tensor
```

定义 target_to_tensor 函数，将目标值准换成一个由第二个字母到尾字母(包括 EOS)组成的张量。

```
def target_to_tensor(line):
    letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]

    # 添加 EOS
    letter_indexes.append(n_letters - 1)

    return Tensor(np.array(letter_indexes), mstype.int64)
```

为了方便训练，我们将使用 random_training 函数来获取随机对(category, line)，并将其转换为所需格式的(category, input, target)张量。

```
def random_training():
    category, line = random_training_pair()
    category_tensor = category_to_tensor(category)
    input_line_tensor = input_to_tensor(line)
    target_line_tensor = target_to_tensor(line)
    return category_tensor, input_line_tensor, target_line_tensor
```

1.3.6.2 训练网络

与分类模型依赖最后输出作为结果不同，这里我们在每一步都进行了预测，因此每一步都需要计算损失。

定义 NLLLoss 损失函数：

```
from mindspore.ops import functional as F

class NLLLoss(nn.loss.loss._Loss):
    def __init__(self, reduction='mean'):
        super(NLLLoss, self).__init__(reduction)
        self.one_hot = ops.OneHot()
        self.reduce_sum = ops.ReduceSum()

    def construct(self, logits, label):
        label_one_hot = self.one_hot(label, F.shape(logits)[-1], F.scalar_to_array(1.0),
F.scalar_to_array(0.0))
        loss = self.reduce_sum(-1.0 * logits * label_one_hot, (1,))
        return self.get_loss(loss)
```

实例化：

```
criterion = NLLLoss()
```

MindSpore 将损失函数，优化器等操作都封装到了 Cell 中，但是本教程 rnn 网络循环的每一步都需要计算损失，所以我们需要自定义 WithLossCell 类，将网络和 Loss 连接起来。

```
class WithLossCellRnn(nn.Cell):
    """构建有损失计算的 RNN 网络"""

    def __init__(self, backbone,loss_fn):
        super(WithLossCellRnn, self).__init__(auto_prefix=True)
        self._backbone = backbone
        self._loss_fn = loss_fn

    def construct(self, category_tensor, input_line_tensor, hidden, target_line_tensor):
        loss = 0
        for i in range(input_line_tensor.shape[0]):
            output, hidden = self._backbone(category_tensor, input_line_tensor[i], hidden)
            l = self._loss_fn(output, target_line_tensor[i])
            loss += l
        return loss
```

创建优化器、WithLossCellRnn 实例和 TrainOneStepCell 训练网络。

```
rnn_cf = RNN(n_letters, 128, n_letters)
optimizer = nn.Momentum(filter(lambda x:x.requires_grad,rnn_cf.get_parameters()),0.0001,0.9)
net_with_criterion = WithLossCellRnn(rnn_cf, criterion)
net = nn.TrainOneStepCell(net_with_criterion, optimizer)
net.set_train()
```

```
# 训练网络
def train(category_tensor,input_line_tensor, target_line_tensor):
    new_shape = list(target_line_tensor.shape)
    new_shape.append(1)
    target_line_tensor = target_line_tensor.reshape(new_shape)
    hidden = rnn_cf.initHidden()
    loss = net(category_tensor, input_line_tensor, hidden,target_line_tensor)
    return  loss / input_line_tensor.shape[0]
```

思考 1：... nn.Momentum(filter(lambda x:x.requires_grad,rnn_cf.get_parameters ... 如何理解这行代码?

其中使用了过滤函数 filter，过滤网络中需要进行梯度计算的参数。

思考 2：rnn_cf 网络结构中 combined 层的维度是什么？（答案参考 1.3.5 小节中的结构图）

输入：

```
for m in rnn_cf.parameters_and_names():
    print(m)
```

输出：

```
('i2h.weight', Parameter (name=_backbone.i2h.weight, shape=(128, 205), dtype=Float32, requires_grad=True))
('i2h.bias', Parameter (name=_backbone.i2h.bias, shape=(128,), dtype=Float32, requires_grad=True))
('i2o.weight', Parameter (name=_backbone.i2o.weight, shape=(59, 205), dtype=Float32, requires_grad=True))
('i2o.bias', Parameter (name=_backbone.i2o.bias, shape=(59,), dtype=Float32, requires_grad=True))
('o2o.weight', Parameter (name=_backbone.o2o.weight, shape=(59, 187), dtype=Float32, requires_grad=True))
('o2o.bias', Parameter (name=_backbone.o2o.bias, shape=(59,), dtype=Float32, requires_grad=True))
```

图1-23 输出结果

通过命令我们打印出了网络的结构，可以看到 combined 层的维度应该为 205，对应 n_categories+input_size+hidden_size。

为了跟踪网络模型训练过程中的耗时，定义 time_since 函数，用来计算训练运行的时间，方便我们持续看到训练的整个过程。

```
import time
import math

# 定义可读时间回调字符串
def time_since(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)
```

在训练过程中，每经过 print_every（500）次迭代就打印一次，分别打印迭代所用时间、迭代次数、迭代进度和损失值。同时，根据 plot_every 的值计算平均损失，将其添加进 all_losses 列表，以便于后面绘制训练过程种损失函数的图像。

```
n_iters = 7500
print_every = 500
plot_every = 100
all_losses = []
```

```
# 每经过 100 次迭代，就重置为 0
total_loss = 0

start = time.time()

for iter in range(1, n_iters + 1):
    #output, loss = train(*random_training())
    loss = train(*random_training())
    total_loss += loss

    # 分别打印迭代所用时间、迭代次数、迭代进度和损失值
    if iter % print_every == 0:
        print('%s (%d %d%%) %.4f'% (time_since(start), iter, iter / n_iters * 100, loss.asnumpy()))

    if iter % plot_every == 0:
        all_losses.append((total_loss / plot_every).asnumpy())
        total_loss = 0
```

输出：

```
5m 56s (500 6%) 3.7287
11m 37s (1000 13%) 4.1077
17m 8s (1500 20%) 4.0800
22m 51s (2000 26%) 4.1025
28m 28s (2500 33%) 4.0878
34m 3s (3000 40%) 4.0277
39m 38s (3500 46%) 4.0859
45m 22s (4000 53%) 3.9899
51m 7s (4500 60%) 3.5648
56m 52s (5000 66%) 4.0283
63m 34s (5500 73%) 4.0877
71m 50s (6000 80%) 4.0858
79m 42s (6500 86%) 4.1082
88m 11s (7000 93%) 3.7557
97m 32s (7500 100%) 4.0461
```

图1-24 输出结果

使用 matplotlib.pyplot 绘制训练过程中损失函数的图像。

```
import matplotlib.pyplot as plt

plt.figure()
plt.plot(all_losses)
```

输出：

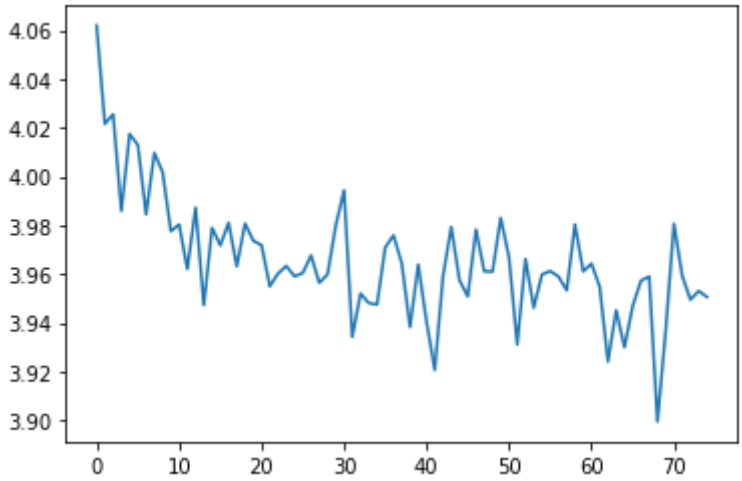

图1-25 输出结果

1.3.6.3 验证模型

在训练结束后，对获得的模型进行验证。这里，我们向网络中输入一个字母并推理得出下一个字母。将输出的字母作为下一步的输入，重复直到 EOS 标记处。

输入：

```
max_length = 20

# 根据类别、起始字母、隐藏状态开始推理
def sample(category, start_letter='A'):
    category_tensor = category_to_tensor(category)
    input = input_to_tensor(start_letter)
    hidden = rnn_cf.initHidden()
    output_name = start_letter

    for i in range(max_length):
        output, hidden = rnn_cf(category_tensor, input[0], hidden)
        topk = ops.TopK(sorted=True)
        topv, topi = topk(output,1)
        topi = topi[0,0]
        if topi == n_letters - 1:
            break
        else:
            letter = all_letters[topi]
            output_name += letter
        input = input_to_tensor(letter)

    return output_name

# 遍历提供的字母，得到输出名称
def samples(category, start_letters='ABC'):
    for start_letter in start_letters:
```

```
        print('语言类型：%s 首字母：%s 输出结果：%s' %(category, start_letter, sample(category,
start_letter)))

samples('Russian', 'RUS')
samples('German', 'GER')
samples('Spanish', 'SPA')
samples('Chinese', 'CHI')
```

输出：

```
语言类型：Russian 首字母：R 输出结果：Rrnarao
语言类型：Russian 首字母：U 输出结果：Uilehidaauritai
语言类型：Russian 首字母：S 输出结果：Sa
语言类型：German 首字母：G 输出结果：Gallh
语言类型：German 首字母：E 输出结果：Ehuhkiakena
语言类型：German 首字母：R 输出结果：Rcsahonuianah
语言类型：Spanish 首字母：S 输出结果：Stadlalugtnaaa
语言类型：Spanish 首字母：P 输出结果：Perahaiarsaorrol
语言类型：Spanish 首字母：A 输出结果：Aaan
语言类型：Chinese 首字母：C 输出结果：Cadco
语言类型：Chinese 首字母：H 输出结果：Hn
语言类型：Chinese 首字母：I 输出结果：I
```

图1-26 输出结果

1.4 实验三：使用 Transformer 模型实现文本翻译

1.4.1 概述

Transformer 是谷歌的研究人员提出的一种全新的模型，Transformer 在被提出之后，很快就席卷了整个自然语言处理领域。与循环神经网络等传统模型不同，Transformer 模型仅仅使用一种被称作自注意力机制的方法和标准的前馈神经网络，完全不依赖任何循环单元或者卷积操作。自注意力机制的优点在于可以直接对序列中任意两个单元之间的关系进行建模，这使得长距离依赖等问题可以更好地被求解。本节实验将探索 Transformer 在中英文机器翻译任务中的应用。

本教程采用的数据为中英文对照翻译的句子集合，共有 2 万多条中英文句子对，示例如下：

```
Hi .——»嗨 。
Hi .——»你 好 。
Run .——»你 用 跑 的 。
Wait !——»等 等 !
Wait !——»等 一 下 !
Hello !»你 好 。
I won !»我 赢 了 。
Oh no !»不 会 吧 。
Cheers !——»干 杯 !
Got it ?——»你 懂 了 吗 ?
He ran .——»他 跑 了 。
Hop in .——»跳 进 来 。
I quit .——»我 退 出 。
I ' m OK .——»我 没 事 。
I ' m up .——»我 已 经 起 来 了 。
Listen .——»听 着 。
No way !——»不 可 能 !
No way !——»没 门 !
Really ?——»你 确 定 ?
Try it .——»试 试 吧 。
We try .——»我 们 来 试 试 。
Why me ?——»为 什 么 是 我 ?
Ask Tom .——»去 问 汤 姆 。
Awesome !——»好 棒 !
Be calm .——»冷 静 点 。
Be fair .——»公 平 点 。
Be kind .——»友 善 点 。
Be nice .——»和 气 点 。
Be nice .——»友 善 点 。
Call me .——»联 系 我 。
```

图1-27 中英文翻译数据

1.4.2 准备阶段

1.4.2.1 训练数据下载

下载训练数据并解压：

```
!wget https://certification-data.obs.cn-north-4.myhuaweicloud.com:443/CHS/HCIP-AI-MindSpore%20Developer/V1.0/chap5/ms_transformer.zip
!unzip ms_transformer.zip
```

输出：

```
Resolving proxy-notebook.modelarts-dev-proxy.com (proxy-notebook.modelarts-dev-proxy.com)... 192.168.0.172
Connecting to proxy-notebook.modelarts-dev-proxy.com (proxy-notebook.modelarts-dev-proxy.com)|192.168.0.172|:8083... connected.
Proxy request sent, awaiting response... 200 OK
Length: 685176 (669K) [application/zip]
Saving to: 'ms_transformer.zip'

ms_transformer.zip  100%[===================>] 669.12K  --.-KB/s    in 0.005s

(122 MB/s) - 'ms_transformer.zip' saved [685176/685176]

Archive:  ms_transformer.zip
   creating: data/
  inflating: data/ch_en_all.txt
  inflating: data/ch_en_vocab.txt
   creating: src/
  inflating: src/beam_search.py
  inflating: src/data_utils.py
  inflating: src/lr_schedule.py
  inflating: src/tokenization.py
  inflating: src/train_util.py
  inflating: src/transformer_for_train.py
  inflating: src/transformer_model.py
  inflating: src/weight_init.py
```

图1-28 输出结果

1.4.2.2 导入依赖库

输入：

```
import os
import numpy as np
from easydict import EasyDict as edict

import mindspore.nn as nn
from mindspore import context
import mindspore.dataset.engine as de
import mindspore.common.dtype as mstype
from mindspore.mindrecord import FileWriter
from mindspore.common.parameter import Parameter
import mindspore.dataset.transforms.c_transforms as deC
from mindspore.common.tensor import Tensor
from mindspore.nn.optim import Adam
from mindspore.train.model import Model
from mindspore.train.loss_scale_manager import DynamicLossScaleManager
from mindspore.train.callback import CheckpointConfig, ModelCheckpoint
from mindspore.train.callback import Callback, TimeMonitor
from mindspore.train.serialization import load_checkpoint, load_param_into_net

from src import tokenization
from src.train_util import LossCallBack
from src.lr_schedule import create_dynamic_lr
from src.transformer_model import TransformerConfig, TransformerModel
from src.data_utils import create_training_instance, write_instance_to_file
from src.transformer_for_train import TransformerTrainOneStepCell, TransformerNetworkWithLoss,
TransformerTrainOneStepWithLossScaleCell
```

1.4.2.3 设置运行模式

输入：

```
context.set_context(mode=context.GRAPH_MODE, device_target="Ascend")
```

1.4.2.4 定义数据处理相关参数

```
data_cfg = edict({
        'input_file': './data/ch_en_all.txt',
        'vocab_file': './data/ch_en_vocab.txt',
        'train_file_mindrecord': './data/train.mindrecord',
        'eval_file_mindrecord': './data/test.mindrecord',
        'train_file_source': './data/source_train.txt',
        'eval_file_source': './data/source_test.txt',
        'num_splits':1,
        'clip_to_max_len': False,
        'max_seq_length': 40
})
```

1.4.3 数据处理

1.4.3.1 定义相关参数

输入：

```
data_cfg = edict({
        'input_file': './data/ch_en_all.txt',
        'vocab_file': './data/ch_en_vocab.txt',
        'train_file_mindrecord': './data/train.mindrecord',
        'eval_file_mindrecord': './data/test.mindrecord',
        'train_file_source': './data/source_train.txt',
        'eval_file_source': './data/source_test.txt',
        'num_splits':1,
        'clip_to_max_len': False,
        'max_seq_length': 40
})
```

1.4.3.2 定义数据处理函数

加载原始数据，切分训练、测试数据，并预处理成模型输入所需的数据形式，并保存为 MindRecord 格式。

注意：第二次运行时需要删除第一次运行生成的 data 文件夹中的 txt 及 mindrecord 文件。

输入：

```
def data_prepare(cfg, eval_idx):
    tokenizer = tokenization.WhiteSpaceTokenizer(vocab_file=cfg.vocab_file)

    writer_train = FileWriter(cfg.train_file_mindrecord, cfg.num_splits)
    writer_eval = FileWriter(cfg.eval_file_mindrecord, cfg.num_splits)
    data_schema = {"source_sos_ids": {"type": "int32", "shape": [-1]},
                   "source_sos_mask": {"type": "int32", "shape": [-1]},
                   "source_eos_ids": {"type": "int32", "shape": [-1]},
                   "source_eos_mask": {"type": "int32", "shape": [-1]},
```

```
                    "target_sos_ids": {"type": "int32", "shape": [-1]},
                    "target_sos_mask": {"type": "int32", "shape": [-1]},
                    "target_eos_ids": {"type": "int32", "shape": [-1]},
                    "target_eos_mask": {"type": "int32", "shape": [-1]}
                    }

    writer_train.add_schema(data_schema, "tranformer train")
    writer_eval.add_schema(data_schema, "tranformer eval")

    index = 0
    f_train = open(cfg.train_file_source, 'w', encoding='utf-8')
    f_test = open(cfg.eval_file_source,'w',encoding='utf-8')
    f = open(cfg.input_file, "r", encoding='utf-8')
    for s_line in f:
        print("finish {}/{}".format(index, 23607), end='\r')

        line = tokenization.convert_to_unicode(s_line)

        source_line, target_line = line.strip().split("\t")
        source_tokens = tokenizer.tokenize(source_line)
        target_tokens = tokenizer.tokenize(target_line)

        if len(source_tokens) >= (cfg.max_seq_length-1) or len(target_tokens) >= (cfg.max_seq_length-
1):
            if cfg.clip_to_max_len:
                source_tokens = source_tokens[:cfg.max_seq_length-1]
                target_tokens = target_tokens[:cfg.max_seq_length-1]
            else:
                continue

        index = index + 1
        # print(source_tokens)
        instance = create_training_instance(source_tokens, target_tokens, cfg.max_seq_length)

        if index in eval_idx:
            f_test.write(s_line)
            features = write_instance_to_file(writer_eval, instance, tokenizer, cfg.max_seq_length)
        else:
            f_train.write(s_line)
            features = write_instance_to_file(writer_train, instance, tokenizer, cfg.max_seq_length)
    f.close()
    f_test.close()
    f_train.close()
    writer_train.commit()
    writer_eval.commit()
```

1.4.3.3 选取测试数据

随机选 20%作为测试数据：

```
sample_num = 23607
eval_idx = np.random.choice(sample_num, int(sample_num*0.2), replace=False)
data_prepare(data_cfg, eval_idx)
```

1.4.3.4 定义数据加载函数

```
def load_dataset(batch_size=1, data_file=None):
    """
    Load mindrecord dataset
    """
    ds = de.MindDataset(data_file,
                        columns_list=["source_eos_ids", "source_eos_mask",
                                      "target_sos_ids", "target_sos_mask",
                                      "target_eos_ids", "target_eos_mask"],
                        shuffle=False)
    type_cast_op = deC.TypeCast(mstype.int32)
    ds = ds.map(input_columns="source_eos_ids", operations=type_cast_op)
    ds = ds.map(input_columns="source_eos_mask", operations=type_cast_op)
    ds = ds.map(input_columns="target_sos_ids", operations=type_cast_op)
    ds = ds.map(input_columns="target_sos_mask", operations=type_cast_op)
    ds = ds.map(input_columns="target_eos_ids", operations=type_cast_op)
    ds = ds.map(input_columns="target_eos_mask", operations=type_cast_op)
    # apply batch operations
    ds = ds.batch(batch_size, drop_remainder=True)
    ds.channel_name = 'transformer'
    return ds
```

1.4.3.5 测试数据加载

输入：

```
next(load_dataset(data_file=data_cfg.train_file_mindrecord).create_dict_iterator())['source_eos_ids'][0]
```

输出：

```
Tensor(shape=[40], dtype=Int32, value= [3983,    3,    2,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0])
```

图1-29 输出结果

1.4.4 定义训练

1.4.4.1 定义训练参数

输入：

```
train_cfg = edict({
    #----------------------------------nework config----------------------------------
    'transformer_network': 'base',
    'init_loss_scale_value': 1024,
```

```
    'scale_factor': 2,
    'scale_window': 2000,

    'lr_schedule': edict({
        'learning_rate': 1.0,
        'warmup_steps': 8000,
        'start_decay_step': 16000,
        'min_lr': 0.0,
    }),
    #----------------------------------save model config------------------------
    'enable_save_ckpt': True ,              #Enable save checkpointdefault is true.
    'save_checkpoint_steps':590,     #Save checkpoint steps, default is 590.
    'save_checkpoint_num':2,         #Save checkpoint numbers, default is 2.
    'save_checkpoint_path': './checkpoint',       #Save checkpoint file path,default is ./checkpoint/
    'save_checkpoint_name':'transformer-32_40',
    'checkpoint_path':'',         #Checkpoint file path


    #------------------------------device config----------------------------
    'enable_data_sink':False,     #Enable data sink, default is False.
    'device_id':0,
    'device_num':1,
    'distribute':False,

    # -----------------mast same with the dataset----------------------
    'seq_length':40,
    'vocab_size':10067,

    #------------------------------------------------------------------------------
    'data_path':"./data/train.mindrecord",     #Data path
    'epoch_size':15,
    'batch_size':32,
    'max_position_embeddings':40,
    'enable_lossscale': False,            #Use lossscale or not, default is False.
    'do_shuffle':True            #Enable shuffle for dataset, default is True.
})
```

网络参数：

```
if train_cfg.transformer_network == 'base':
    transformer_net_cfg = TransformerConfig(
        batch_size=train_cfg.batch_size,
        seq_length=train_cfg.seq_length,
        vocab_size=train_cfg.vocab_size,
        hidden_size=512,
        num_hidden_layers=6,
        num_attention_heads=8,
        intermediate_size=2048,
        hidden_act="relu",
```

```
            hidden_dropout_prob=0.2,
            attention_probs_dropout_prob=0.2,
            max_position_embeddings=train_cfg.max_position_embeddings,
            initializer_range=0.02,
            label_smoothing=0.1,
            input_mask_from_dataset=True,
            dtype=mstype.float32,
            compute_type=mstype.float16)
elif train_cfg.transformer_network == 'large':
    transformer_net_cfg = TransformerConfig(
            batch_size=train_cfg.batch_size,
            seq_length=train_cfg.seq_length,
            vocab_size=train_cfg.vocab_size,
            hidden_size=1024,
            num_hidden_layers=6,
            num_attention_heads=16,
            intermediate_size=4096,
            hidden_act="relu",
            hidden_dropout_prob=0.2,
            attention_probs_dropout_prob=0.2,
            max_position_embeddings=train_cfg.max_position_embeddings,
            initializer_range=0.02,
            label_smoothing=0.1,
            input_mask_from_dataset=True,
            dtype=mstype.float32,
            compute_type=mstype.float16)
else:
    raise Exception("The src/train_confige of transformer_network must base or large. Change the
str/train_confige file and try again!")
```

1.4.4.2 定义训练函数

输入：

```
def train(cfg):
    """
    Transformer training.
    """

    train_dataset = load_dataset(cfg.batch_size, data_file=cfg.data_path)

    netwithloss = TransformerNetworkWithLoss(transformer_net_cfg, True)

    if cfg.checkpoint_path:
        parameter_dict = load_checkpoint(cfg.checkpoint_path)
        load_param_into_net(netwithloss, parameter_dict)

    lr = Tensor(create_dynamic_lr(schedule="constant*rsqrt_hidden*linear_warmup*rsqrt_decay",
                                  training_steps=train_dataset.get_dataset_size()*cfg.epoch_size,
```

```
                                    learning_rate=cfg.lr_schedule.learning_rate,
                                    warmup_steps=cfg.lr_schedule.warmup_steps,
                                    hidden_size=transformer_net_cfg.hidden_size,
                                    start_decay_step=cfg.lr_schedule.start_decay_step,
                                    min_lr=cfg.lr_schedule.min_lr), mstype.float32)
    optimizer = Adam(netwithloss.trainable_params(), lr)

    callbacks = [TimeMonitor(train_dataset.get_dataset_size()), LossCallBack()]
    if cfg.enable_save_ckpt:
        ckpt_config = CheckpointConfig(save_checkpoint_steps=cfg.save_checkpoint_steps,
                                       keep_checkpoint_max=cfg.save_checkpoint_num)
        ckpoint_cb = ModelCheckpoint(prefix=cfg.save_checkpoint_name,
directory=cfg.save_checkpoint_path, config=ckpt_config)
        callbacks.append(ckpoint_cb)

    if cfg.enable_lossscale:
        scale_manager = DynamicLossScaleManager(init_loss_scale=cfg.init_loss_scale_value,
                                                scale_factor=cfg.scale_factor,
                                                scale_window=cfg.scale_window)
        update_cell = scale_manager.get_update_cell()
        netwithgrads = TransformerTrainOneStepWithLossScaleCell(netwithloss,
optimizer=optimizer,scale_update_cell=update_cell)
    else:
        netwithgrads = TransformerTrainOneStepCell(netwithloss, optimizer=optimizer)

    netwithgrads.set_train(True)
    model = Model(netwithgrads)
    model.train(cfg.epoch_size, train_dataset, callbacks=callbacks,
dataset_sink_mode=cfg.enable_data_sink)
```

1.4.4.3 启动训练

输入：

```
train(train_cfg)
```

```
time: 1167727, epoch: 1, step: 1, outputs are [10.03706]
time: 1168041, epoch: 1, step: 2, outputs are [9.874242]
time: 1168081, epoch: 1, step: 3, outputs are [10.024283]
time: 1168119, epoch: 1, step: 4, outputs are [9.862464]
time: 1168155, epoch: 1, step: 5, outputs are [9.945937]
time: 1168191, epoch: 1, step: 6, outputs are [9.940716]
time: 1168227, epoch: 1, step: 7, outputs are [9.963706]
time: 1168262, epoch: 1, step: 8, outputs are [9.984086]
time: 1168297, epoch: 1, step: 9, outputs are [9.972837]
time: 1168334, epoch: 1, step: 10, outputs are [9.978711]
time: 1168372, epoch: 1, step: 11, outputs are [10.030728]
time: 1168408, epoch: 1, step: 12, outputs are [9.898652]
time: 1168445, epoch: 1, step: 13, outputs are [9.97211]
time: 1168484, epoch: 1, step: 14, outputs are [9.8448]
time: 1168521, epoch: 1, step: 15, outputs are [9.8156805]
time: 1168557, epoch: 1, step: 16, outputs are [9.743874]
time: 1168593, epoch: 1, step: 17, outputs are [9.799403]
time: 1168630, epoch: 1, step: 18, outputs are [9.913165]
time: 1168667, epoch: 1, step: 19, outputs are [9.760037]
time: 1168703, epoch: 1, step: 20, outputs are [9.657992]
time: 1168739, epoch: 1, step: 21, outputs are [9.724083]
time: 1168775, epoch: 1, step: 22, outputs are [9.591026]
time: 1168811, epoch: 1, step: 23, outputs are [9.80297]
time: 1168847, epoch: 1, step: 24, outputs are [9.827316]
time: 1168883, epoch: 1, step: 25, outputs are [9.559591]
time: 1168920, epoch: 1, step: 26, outputs are [9.630765]
time: 1168956, epoch: 1, step: 27, outputs are [9.463372]
time: 1168994, epoch: 1, step: 28, outputs are [9.59371]
time: 1169030, epoch: 1, step: 29, outputs are [9.637667]
time: 1169067, epoch: 1, step: 30, outputs are [9.627168]
time: 1169103, epoch: 1, step: 31, outputs are [9.598762]
time: 1169140, epoch: 1, step: 32, outputs are [9.431324]
time: 1169177, epoch: 1, step: 33, outputs are [9.587903]
time: 1169213, epoch: 1, step: 34, outputs are [9.280119]
time: 1169250, epoch: 1, step: 35, outputs are [9.260801]
time: 1169285, epoch: 1, step: 36, outputs are [9.311937]
```

图1-30 部分输出结果

1.4.5 定义评估

1.4.5.1 定义参数配置

输入：

```
eval_cfg = edict({
    'transformer_network': 'base',

    'data_file': './data/test.mindrecord',
    'test_source_file':'./data/source_test.txt',
    'model_file': './checkpoint/transformer-32_40-15_590.ckpt' , #查看 checkpoint 文件夹中的文件是否存
在
    'vocab_file':'./data/ch_en_vocab.txt',
    'token_file': './token-32-40.txt',
    'pred_file':'./pred-32-40.txt',

    # ------------------mast same with the train config and the datsset-----------------------
    'seq_length':40,
    'vocab_size':10067,

    #-----------------------------------eval config----------------------------
    'batch_size':32,
    'max_position_embeddings':40          # mast same with the train config
})

'''
```

```
two kinds of transformer model version
'''
if eval_cfg.transformer_network == 'base':
    transformer_net_cfg = TransformerConfig(
        batch_size=eval_cfg.batch_size,
        seq_length=eval_cfg.seq_length,
        vocab_size=eval_cfg.vocab_size,
        hidden_size=512,
        num_hidden_layers=6,
        num_attention_heads=8,
        intermediate_size=2048,
        hidden_act="relu",
        hidden_dropout_prob=0.0,
        attention_probs_dropout_prob=0.0,
        max_position_embeddings=eval_cfg.max_position_embeddings,
        label_smoothing=0.1,
        input_mask_from_dataset=True,
        beam_width=4,
        max_decode_length=eval_cfg.seq_length,
        length_penalty_weight=1.0,
        dtype=mstype.float32,
        compute_type=mstype.float16)

elif eval_cfg.transformer_network == 'large':
    transformer_net_cfg = TransformerConfig(
        batch_size=eval_cfg.batch_size,
        seq_length=eval_cfg.seq_length,
        vocab_size=eval_cfg.vocab_size,
        hidden_size=1024,
        num_hidden_layers=6,
        num_attention_heads=16,
        intermediate_size=4096,
        hidden_act="relu",
        hidden_dropout_prob=0.0,
        attention_probs_dropout_prob=0.0,
        max_position_embeddings=eval_cfg.max_position_embeddings,
        label_smoothing=0.1,
        input_mask_from_dataset=True,
        beam_width=4,
        max_decode_length=80,
        length_penalty_weight=1.0,
        dtype=mstype.float32,
        compute_type=mstype.float16)
else:
    raise Exception("The src/eval_confige of transformer_network must base or large and same with the
train_confige confige. Change the str/eval_confige file and try again!")
```

1.4.5.2 定义评估测试函数

输入：

```
class TransformerInferCell(nn.Cell):
    """
    Encapsulation class of transformer network infer.
    """
    def __init__(self, network):
        super(TransformerInferCell, self).__init__(auto_prefix=False)
        self.network = network

    def construct(self,
                  source_ids,
                  source_mask):
        predicted_ids = self.network(source_ids, source_mask)
        return predicted_ids

def load_weights(model_path):
    """
    Load checkpoint as parameter dict, support both npz file and mindspore checkpoint file.
    """
    if model_path.endswith(".npz"):
        ms_ckpt = np.load(model_path)
        is_npz = True
    else:
        ms_ckpt = load_checkpoint(model_path)
        is_npz = False

    weights = {}
    for msname in ms_ckpt:
        infer_name = msname
        if "tfm_decoder" in msname:
            infer_name = "tfm_decoder.decoder." + infer_name
        if is_npz:
            weights[infer_name] = ms_ckpt[msname]
        else:
            weights[infer_name] = ms_ckpt[msname].data.asnumpy()
    weights["tfm_decoder.decoder.tfm_embedding_lookup.embedding_table"] = \
        weights["tfm_embedding_lookup.embedding_table"]

    parameter_dict = {}
    for name in weights:
        parameter_dict[name] = Parameter(Tensor(weights[name]), name=name)
    return parameter_dict

def evaluate(cfg):
    """
```

```
    Transformer evaluation.
    """
    context.set_context(mode=context.GRAPH_MODE, device_target="Ascend",
reserve_class_name_in_scope=False)

    tfm_model = TransformerModel(config=transformer_net_cfg, is_training=False,
use_one_hot_embeddings=False)
    print(cfg.model_file)
    parameter_dict = load_weights(cfg.model_file)
    load_param_into_net(tfm_model, parameter_dict)
    tfm_infer = TransformerInferCell(tfm_model)
    model = Model(tfm_infer)

    tokenizer = tokenization.WhiteSpaceTokenizer(vocab_file=cfg.vocab_file)
    dataset = load_dataset(batch_size=cfg.batch_size, data_file=cfg.data_file)
    predictions = []
    source_sents = []
    target_sents = []
    f2 = open(cfg.test_source_file, 'r', encoding='utf-8')
    for batch in dataset.create_dict_iterator():
        source_sents.append(batch["source_eos_ids"])
        target_sents.append(batch["target_eos_ids"])
        source_ids = Tensor(batch["source_eos_ids"], mstype.int32)
        source_mask = Tensor(batch["source_eos_mask"], mstype.int32)
        predicted_ids = model.predict(source_ids, source_mask)
        #predictions.append(predicted_ids.asnumpy())
        # ----------------------------------------decode and write to file(token file)----------------------
        batch_out = predicted_ids.asnumpy()
        for i in range(transformer_net_cfg.batch_size):
            if batch_out.ndim == 3:
                batch_out = batch_out[:, 0]
            token_ids = [str(x) for x in batch_out[i].tolist()]
            token=" ".join(token_ids)
            #------------------------------token_ids to real output file------------------------------
            token_ids = [int(x) for x in token.strip().split()]
            tokens = tokenizer.convert_ids_to_tokens(token_ids)
            sent = " ".join(tokens)
            sent = sent.split("<s>")[-1]
            sent = sent.split("</s>")[0]

            label_sent = f2.readline().strip()+'\t'
            print("source: {}".format(label_sent))
            print("result: {}".format(sent.strip()))
```

1.4.5.3 启动评估测试

```
evaluate(eval_cfg)
```

部分输出：

```
source: Wait ! 等 等 !
result: 等 着 !
source: Wait ! 等 一 下 !
result: 等 着 !
source: He ran . 他 跑 了 。
result: 他 跑 了 。
source: I ' m up . 我 已 经 起 来 了 。
result: 我 已 经 起 床 单 了 。
source: Listen . 听 着 。
result: 听 着 她 听 着 的 话 , 听 着 听 着 听 着 。
source: No way ! 没 门 !
result: 没 有 !
source: We try . 我 们 来 试 试 。
result: 我 们 试 着 试 试 试 试 著 试 试 。
source: Be fair . 公 平 点 。
result: 公 平 点 半 平 公 平 。
source: Be kind . 友 善 点 。
result: 友 善 点 。
source: Call me . 联 系 我 。
result: 打 电 话 给 我 打 电 话 打 电 话 打 电 话 。
source: Goodbye ! 告 辞 !
result: 再 见 !
source: Hang on ! 坚 持 。
result: 等 一 下 !
source: Hug Tom . 请 抱 紧 汤 姆 。
result: 抱 紧 汤 姆 , 请 抱 紧 紧 汤 姆 。
source: I ' m wet . 我 湿 了 。
result: 我 湿 了 。
source: Keep it . 留 着 吧 。
result: 留 着 吧 。
source: Kiss me . 吻 我 。
result: 吻 我 。
source: Perfect ! 完 美 !
```

图1-31 部分输出结果

1.5 思考题

问题：实验三中两种 Transformer 的模型配置，主要的差别在哪里?

答案：隐藏层的维度，多头注意力的数量，中间连接层的维度。

1.6 缩略语表

NLP：Natural Language Processing，自然语言处理。

RNN：Recurrent Neural Network，循环神经网络。