昇腾创新实践课

MindSpore进阶操作

实验手册



华为技术有限公司

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

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

目录

[1 实验环境介绍 2](#_Toc119601167)

[1.1 实验介绍 2](#_Toc119601168)

[1.1.1 关于本实验 2](#_Toc119601169)

[1.1.2 实验环境介绍 2](#_Toc119601170)

[2 MindSpore进阶操作 3](#_Toc119601171)

[2.1 实验介绍 3](#_Toc119601172)

[2.1.1 关于本实验 3](#_Toc119601173)

[2.1.2 实验目的 3](#_Toc119601174)

[2.1.3 背景知识 3](#_Toc119601175)

[2.1.4 实验设计 3](#_Toc119601176)

[2.2 实验过程 3](#_Toc119601177)

[2.2.1 参数初始化 4](#_Toc119601178)

[2.2.2 运行环境配置 5](#_Toc119601179)

[2.2.3 数据集预处理 6](#_Toc119601180)

[2.2.4 数据增强 12](#_Toc119601181)

[2.2.5 进阶神经网络搭建 18](#_Toc119601182)

[2.2.6 回调函数Callback 22](#_Toc119601183)

[2.2.7 模型保存与加载 25](#_Toc119601184)

[2.3 实验总结 27](#_Toc119601185)

# 实验环境介绍

## 实验介绍

### 关于本实验

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。

本课程将介绍MindSpore的进阶操作，通过学习本课程，学员将掌握MindSpore的进阶用法，比如运行环境配置、数据增强、回调函数、模型保存与加载等，以及如何用MindSpore搭建复杂神经网络的必备知识。

### 实验环境介绍

实验、介绍、难度、软件环境、硬件环境：

实验环境介绍

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实验 | 实验介绍 | 难度 | 软件环境 | 开发环境 |
| MindSpore进阶操作 | 介绍MindSpore的进阶用法，搭建复杂神经网络的必备知识 | 简单 | Python3.7 MindSpore1.7 | PC机 |

# MindSpore进阶操作

## 实验介绍

### 关于本实验

本实验主要介绍MindSpore的进阶用法，比如参数初始化、运行环境配置、数据增强、回调函数等，让学员了解如何用MindSpore进行数据预处理以及搭建复杂神经网络，掌握MindSpore开发的较为完整的流程。

### 实验目的

理解MindSpore开发全流程。

理解MindSpore进阶模块的功能。

掌握MindSpore的进阶操作。

### 背景知识

神经网络知识，参数初始化、数据预处理、数据增强、卷积神经网络（卷积层、池化层）、损失函数。

### 实验设计

1.参数初始化

2.运行环境配置

3.数据集预处理

4.数据增强

5.进阶神经网络搭建

6.回调函数Callback

7.模型保存与加载

## 实验过程

实验中涉及的各API请查阅：

<https://www.mindspore.cn/docs/zh-CN/r1.7/index.html>

### 参数初始化

深度学习模型训练需要对参数进行不断更新，因此权重初始化对于模型性能至关重要，好的初始化参数能够加快模型收敛速度，尽可能找到最优解，缓解梯度消失和梯度爆炸的问题。

创建并初始化张量

API：mindspore.common.initializer.initializer

代码：

import numpy as np

import mindspore

from mindspore import Tensor

from mindspore.common.initializer import initializer, One

data = Tensor(np.zeros([1, 2, 3]), mindspore.float32)

tensor1 = initializer(data, [1, 2, 3], mindspore.float32)

tensor2 = initializer('ones', [1, 2, 3], mindspore.float32)

tensor3 = initializer(One(), [1, 2, 3], mindspore.float32)

tensor4 = initializer(0, [1, 2, 3], mindspore.float32)

print(data)

print(tensor1)

print(tensor2)

print(tensor3)

print(tensor4)

print(tensor4.shape)

输出：

[[[0. 0. 0.]

[0. 0. 0.]]]

[[[0. 0. 0.]

[0. 0. 0.]]]

[[[1. 1. 1.]

[1. 1. 1.]]]

[[[1. 1. 1.]

[1. 1. 1.]]]

[[[0. 0. 0.]

[0. 0. 0.]]]

(1, 2, 3)

正态分布初始化

API：mindspore.common.initializer.Normal(sigma=0.01)

用正态分布函数进行初始化。Sigma为正态分布的标准差。

代码：

from mindspore.common.initializer import initializer, Normal

tensor11 = initializer(Normal(12,4), [1, 2, 3], mindspore.int32)

tensor12 = initializer('normal', [1, 2, 3], mindspore.float32)

print(tensor11)

print(tensor12)

输出：

[[[-1 10 -3]

[ 7 7 18]]]

[[[ 0.0109255 -0.00453881 0.01092415]

[-0.00391397 0.00829144 -0.00115683]]]

### 运行环境配置

mindspore的context可用于配置当前的执行环境，包括执行模式、执行后端和其他特性切换。

设置运行环境

API：mindspore.context.set\_context

Mode：运行模式为静态图（GRAPH\_MODE/0）或动态图（PYNATIVE\_MODE/1），默认为动态图；

device\_target：运行设备为 “Ascend”, “GPU”或者 “CPU”，默认为“Ascend”;

device\_id：运行设备ID，默认为0.

代码：

from mindspore import context

context.set\_context(mode=context.GRAPH\_MODE) # 设置静态图

context.set\_context(mode=context.PYNATIVE\_MODE) # 设置动态图

context.set\_context(device\_target="CPU") # 设置运行设备为CPU

context.set\_context(device\_id=0) # 设置运行设备ID（只对昇腾芯片有效）

查看运行环境

mindspore.context.get\_context

代码：

# 查看运行图模式，0为静态图， 1为动态图

mode = mindspore.context.get\_context('mode')

# 查看运行设备

device\_target = mindspore.context.get\_context('device\_target')

print(mode)

print(device\_target)

输出：

1

CPU

### 数据集预处理

数据预处理在众多深度学习算法中都起着重要作用，数据预处理可以提高模型精度，加快模型收敛速度，提升模型性能，这里主要介绍MindSpore常用的数据预处理方式。

数据是深度学习的基础，良好的数据输入可以对整个深度神经网络训练起到非常积极的作用。在训练前对已加载的数据集进行数据处理，可以解决诸如数据量过大、样本分布不均等问题，从而获得更加优化的数据输入。

修改数据类型

API：mindspore.dataset.transforms.c\_transforms.TypeCast(data\_type)

将输入的Tensor转换为指定的数据类型，代码：

import mindspore.dataset.transforms.c\_transforms as c\_transforms

import mindspore.common.dtype as mstype

from mindspore import dataset as ds

dataset\_dir = "./data/train" # 数据集路径，Windows和Linux适用

mnist\_dataset = ds.MnistDataset(dataset\_dir=dataset\_dir, num\_samples=3)

# 查看mnist\_dataset “label”的数据类型

for x in mnist\_dataset.create\_dict\_iterator():

print(x['label'].dtype)

print("-----------------------------------")

# 修改mnist\_dataset “label”的数据类型为int32

type\_cast\_op = c\_transforms.TypeCast(mstype.int32)

mnist = mnist\_dataset.map(input\_columns="label", operations=type\_cast\_op)

for x in mnist.create\_dict\_iterator():

print(x['label'].dtype)

输出：

UInt32

UInt32

UInt32

-----------------------------------

Int32

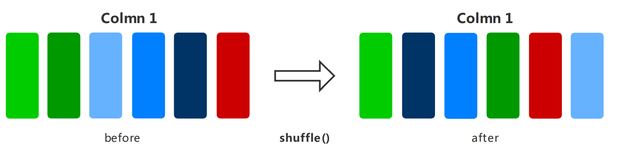
Int32

Int32

Shuffle

API：mindspore.dataset.GeneratorDataset

对数据集进行混洗，随机打乱数据顺序。设定的buffer\_size越大，混洗程度越大，但时间、计算资源消耗也会更大。



代码：

import numpy as np

import mindspore.dataset as ds

ds.config.set\_seed(0)

def generator\_func():

for i in range(5):

yield (np.array([i, i+1, i+2]),)

dataset1 = ds.GeneratorDataset(generator\_func, ["data"])

for data in dataset1.create\_dict\_iterator():

print(data)

print("-----------------------------------")

# shuffle 打乱数据集

dataset2 = dataset1.shuffle(buffer\_size=5)

for data in dataset2.create\_dict\_iterator():

print(data)

输出：

{'data': Tensor(shape=[3], dtype=Int32, value= [0, 1, 2])}

{'data': Tensor(shape=[3], dtype=Int32, value= [1, 2, 3])}

{'data': Tensor(shape=[3], dtype=Int32, value= [2, 3, 4])}

{'data': Tensor(shape=[3], dtype=Int32, value= [3, 4, 5])}

{'data': Tensor(shape=[3], dtype=Int32, value= [4, 5, 6])}

-----------------------------------

{'data': Tensor(shape=[3], dtype=Int32, value= [4, 5, 6])}

{'data': Tensor(shape=[3], dtype=Int32, value= [3, 4, 5])}

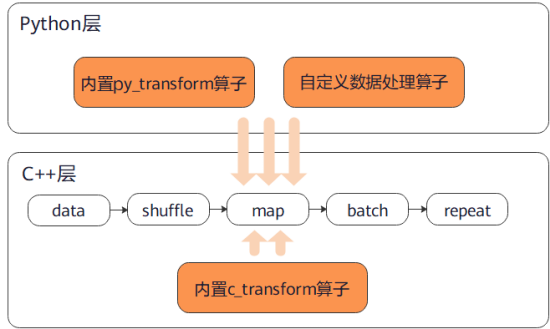
{'data': Tensor(shape=[3], dtype=Int32, value= [1, 2, 3])}

{'data': Tensor(shape=[3], dtype=Int32, value= [0, 1, 2])}

{'data': Tensor(shape=[3], dtype=Int32, value= [2, 3, 4])}

Map

将指定的函数或算子作用于数据集的指定列数据，实现数据映射操作。用户可以自定义映射函数，也可以直接使用c\_transforms或py\_transforms中的算子针对图像、文本数据进行数据增强。



代码：

import numpy as np

import mindspore.dataset as ds

def generator\_func():

for i in range(5):

yield (np.array([i, i+1, i+2]),)

def pyfunc(x):

return x\*2

dataset1 = ds.GeneratorDataset(generator\_func, ["data"])

for data in dataset1.create\_dict\_iterator():

print(data)

print("------ after processing ------")

dataset2 = dataset1.map(operations=pyfunc, input\_columns=["data"])

for data in dataset2.create\_dict\_iterator():

print(data)

输出：

{'data': Tensor(shape=[3], dtype=Int32, value= [0, 1, 2])}

{'data': Tensor(shape=[3], dtype=Int32, value= [1, 2, 3])}

{'data': Tensor(shape=[3], dtype=Int32, value= [2, 3, 4])}

{'data': Tensor(shape=[3], dtype=Int32, value= [3, 4, 5])}

{'data': Tensor(shape=[3], dtype=Int32, value= [4, 5, 6])}

------ after processing ------

{'data': Tensor(shape=[3], dtype=Int32, value= [0, 2, 4])}

{'data': Tensor(shape=[3], dtype=Int32, value= [2, 4, 6])}

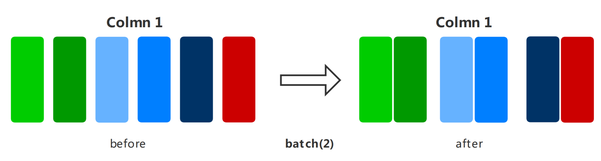
{'data': Tensor(shape=[3], dtype=Int32, value= [4, 6, 8])}

{'data': Tensor(shape=[3], dtype=Int32, value= [ 6, 8, 10])}

{'data': Tensor(shape=[3], dtype=Int32, value= [ 8, 10, 12])}

Batch

将数据集分批，分别输入到训练系统中进行训练，可以减少训练轮次，达到加速训练过程的目的。



代码：

import numpy as np

import mindspore.dataset as ds

def generator\_func():

for i in range(5):

yield (np.array([i, i+1, i+2]),)

dataset1 = ds.GeneratorDataset(generator\_func, ["data"])

dataset1 = dataset1.batch(batch\_size=2, drop\_remainder=False)

for data in dataset1.create\_dict\_iterator():

print(data)

print("------ drop remainder ------")

dataset2 = ds.GeneratorDataset(generator\_func, ["data"])

dataset2 = dataset2.batch(batch\_size=2, drop\_remainder=True)

for data in dataset2.create\_dict\_iterator():

print(data)

输出：

{'data': Tensor(shape=[2, 3], dtype=Int32, value=

[[0, 1, 2],

[1, 2, 3]])}

{'data': Tensor(shape=[2, 3], dtype=Int32, value=

[[2, 3, 4],

[3, 4, 5]])}

{'data': Tensor(shape=[1, 3], dtype=Int32, value=

[[4, 5, 6]])}

------ drop remainder ------

{'data': Tensor(shape=[2, 3], dtype=Int32, value=

[[0, 1, 2],

[1, 2, 3]])}

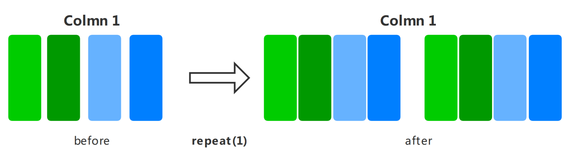
{'data': Tensor(shape=[2, 3], dtype=Int32, value=

[[2, 3, 4],

[3, 4, 5]])}

Repeat

对数据集进行重复，达到扩充数据量的目的。repeat和batch操作的顺序会影响训练batch的数量，建议将repeat置于batch之后。



代码：

import numpy as np

import mindspore.dataset as ds

def generator\_func():

for i in range(5):

yield (np.array([i, i+1, i+2]),)

dataset1 = ds.GeneratorDataset(generator\_func, ["data"])

dataset1 = dataset1.repeat(count=2)

for data in dataset1.create\_dict\_iterator():

print(data)

输出：

{'data': Tensor(shape=[3], dtype=Int32, value= [0, 1, 2])}

{'data': Tensor(shape=[3], dtype=Int32, value= [1, 2, 3])}

{'data': Tensor(shape=[3], dtype=Int32, value= [2, 3, 4])}

{'data': Tensor(shape=[3], dtype=Int32, value= [3, 4, 5])}

{'data': Tensor(shape=[3], dtype=Int32, value= [4, 5, 6])}

{'data': Tensor(shape=[3], dtype=Int32, value= [0, 1, 2])}

{'data': Tensor(shape=[3], dtype=Int32, value= [1, 2, 3])}

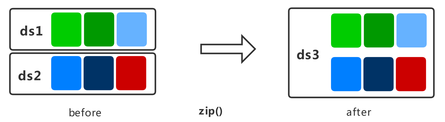
{'data': Tensor(shape=[3], dtype=Int32, value= [2, 3, 4])}

{'data': Tensor(shape=[3], dtype=Int32, value= [3, 4, 5])}

{'data': Tensor(shape=[3], dtype=Int32, value= [4, 5, 6])}

Zip

将两个数据集进行列拼接，合并为一个数据集。如果两个数据集的列名相同，则不会合并，请注意列的命名；如果两个数据集的行数不同，合并后的行数将和较小行数保持一致。



代码：

import numpy as np

import mindspore.dataset as ds

def generator\_func():

for i in range(7):

yield (np.array([i, i+1, i+2]),)

def generator\_func2():

for i in range(4):

yield (np.array([1, 2]),)

dataset1 = ds.GeneratorDataset(generator\_func, ["data1"])

for data in dataset1.create\_dict\_iterator():

print(data)

print("-----------------------------------")

dataset2 = ds.GeneratorDataset(generator\_func2, ["data2"])

for data in dataset2.create\_dict\_iterator():

print(data)

print("-----------------------------------")

dataset3 = ds.zip((dataset1, dataset2))

for data in dataset3.create\_dict\_iterator():

print(data)

输出：

{'data1': Tensor(shape=[3], dtype=Int32, value= [0, 1, 2])}

{'data1': Tensor(shape=[3], dtype=Int32, value= [1, 2, 3])}

{'data1': Tensor(shape=[3], dtype=Int32, value= [2, 3, 4])}

{'data1': Tensor(shape=[3], dtype=Int32, value= [3, 4, 5])}

{'data1': Tensor(shape=[3], dtype=Int32, value= [4, 5, 6])}

{'data1': Tensor(shape=[3], dtype=Int32, value= [5, 6, 7])}

{'data1': Tensor(shape=[3], dtype=Int32, value= [6, 7, 8])}

-----------------------------------

{'data2': Tensor(shape=[2], dtype=Int32, value= [1, 2])}

{'data2': Tensor(shape=[2], dtype=Int32, value= [1, 2])}

{'data2': Tensor(shape=[2], dtype=Int32, value= [1, 2])}

{'data2': Tensor(shape=[2], dtype=Int32, value= [1, 2])}

-----------------------------------

{'data1': Tensor(shape=[3], dtype=Int32, value= [0, 1, 2]), 'data2': Tensor(shape=[2], dtype=Int32, value= [1, 2])}

{'data1': Tensor(shape=[3], dtype=Int32, value= [1, 2, 3]), 'data2': Tensor(shape=[2], dtype=Int32, value= [1, 2])}

{'data1': Tensor(shape=[3], dtype=Int32, value= [2, 3, 4]), 'data2': Tensor(shape=[2], dtype=Int32, value= [1, 2])}

{'data1': Tensor(shape=[3], dtype=Int32, value= [3, 4, 5]), 'data2': Tensor(shape=[2], dtype=Int32, value= [1, 2])}

### 数据增强

在计算机视觉任务中，数据量过小或是样本场景单一等问题都会影响模型的训练效果，可以通过数据增强操作对图像进行预处理，从而提升模型的泛化性。

随机裁剪图片

API：mindspore.dataset.vision.c\_transforms.RandomCrop

代码：

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

import matplotlib.pyplot as plt

# 从mnist dataset读取3张图片

mnist\_dataset = ds.MnistDataset(dataset\_dir=dataset\_dir, num\_samples=3)

i = 1

# 显示原图

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

for dic in mnist\_dataset.create\_dict\_iterator():

plt.subplot(3,3,i)

plt.imshow(dic['image'].asnumpy())

plt.axis('off')

plt.title('Orignal')

i +=1

plt.show()

# 随机裁剪：输出大小（32x32）, padding 方式 左上右下。

random\_crop\_op = CV.RandomCrop([32, 20], [4, 4, 4, 4])

mnist\_dataset1 = mnist\_dataset.map(operations=random\_crop\_op, input\_columns=["image"])

# 显示随机裁剪后的图片

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

for dic in mnist\_dataset1.create\_dict\_iterator(output\_numpy=True):

plt.subplot(3,3,i)

plt.imshow(dic['image'][:,:,0])

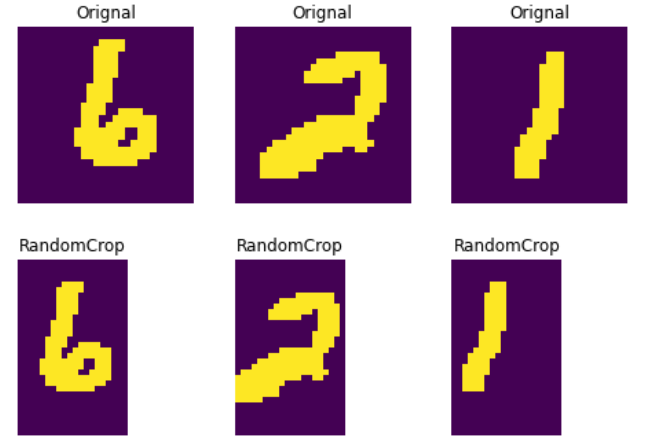
plt.axis('off')

plt.title('RandomCrop')

i +=1

plt.show()

输出：



调整图片大小

API：mindspore.dataset.vision.c\_transforms.Resize

代码：

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

# 从mnist dataset读取3张图片

mnist\_dataset = ds.MnistDataset(dataset\_dir=dataset\_dir, num\_samples=3)

i = 1

# 显示原图

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

for dic in mnist\_dataset.create\_dict\_iterator():

plt.subplot(3,3,i)

plt.imshow(dic['image'].asnumpy())

plt.axis('off')

plt.title('Orignal')

i +=1

plt.show()

# 调整图像大小

resize = CV.Resize([20, 35])

mnist\_dataset1 = mnist\_dataset.map(operations=resize, input\_columns=["image"])

# 显示调整大小后的图片

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

for dic in mnist\_dataset1.create\_dict\_iterator():

plt.subplot(3,3,i)

plt.imshow(dic['image'].asnumpy())

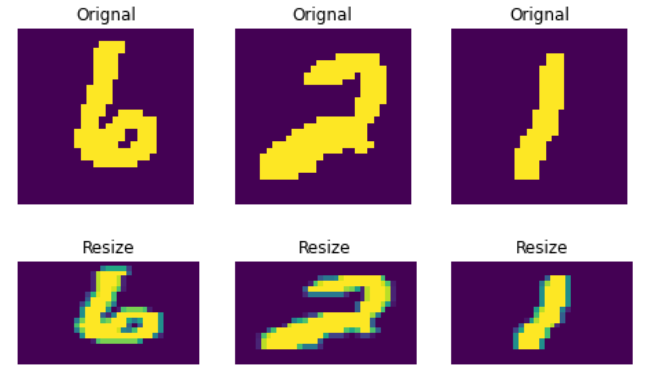
plt.axis('off')

plt.title('Resize')

i +=1

plt.show()

输出：



像素值修改

API：mindspore.dataset.vision.c\_transforms.Rescale(rescale, shift)

代码：

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

# 从mnist dataset读取3张图片

mnist\_dataset = ds.MnistDataset(dataset\_dir=dataset\_dir, num\_samples=3)

# 原图像素值

for dic in mnist\_dataset.create\_dict\_iterator(output\_numpy=True):

print("原图像素值:", dic['image'][:,:,0])

# rescale 缩放系数，shift 平移大小，相当于ax+b

rescale = CV.Rescale(rescale=3, shift=100)

mnist\_dataset1 = mnist\_dataset.map(operations=rescale, input\_columns=["image"])

# 修改后的像素值

for dic in mnist\_dataset1.create\_dict\_iterator(output\_numpy=True):

print("修改后的像素值:", dic['image'][:,:,0])

输出：

原图像素值: [[ 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 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 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]]

修改后的像素值: [[100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100.

100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100.]

[100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100.

100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100.]

…

[100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100.

100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100.]

[100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100.

100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100. 100.]]

图像标准化

API：mindspore.dataset.vision.py\_transforms.Normalize(mean, std)

将输入的NumPy图像数组（形状必须为(C, H, W)）用给定的均值和标准差进行标准化。

数组的值必须在(0, 1]范围内。

代码：

import numpy as np

import mindspore.dataset as ds

# 读取图片，decode=TRUE 解码为（H,W,C）的格式

dataset = ds.ImageFolderDataset('./data/cat', decode=True)

# 原图像素值

for x in dataset.create\_dict\_iterator():

print("原图像形状:",x['image'].shape) # 高，宽，通道数

print("原图像素值:",x['image'])

# 进行标准化操作

normalize\_op = CV.Normalize(mean=(123.68,116.78,103.94), std=[1, 1, 1])

dataset = dataset.map(operations=normalize\_op, input\_columns=["image"])

# 标准化之后的像素值

for x in dataset.create\_dict\_iterator():

print("标准化之后的像素值:",x['image'])

输出：

原图像形状: (2448, 3264, 3)

原图像素值: [[[131 145 109]

[131 145 109]

[132 146 110]

...

...

[158 105 51]

[158 105 51]

[158 105 51]]]

标准化之后的像素值: [[[ 7.3199997 28.220001 5.0599976]

[ 7.3199997 28.220001 5.0599976]

[ 8.32 29.220001 6.0599976]

...

...

[ 34.32 -11.779999 -52.940002 ]

[ 34.32 -11.779999 -52.940002 ]

[ 34.32 -11.779999 -52.940002 ]]]

随机水平翻转图像

API：mindspore.dataset.vision.c\_transforms.RandomHorizontalFlip(prob=0.5)

代码：

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

import numpy as np

import mindspore.dataset as ds

# 读取图片，decode=TRUE 解码为（H,W,C）的格式

dataset1 = ds.ImageFolderDataset('./data/cat', decode= True)

i = 1

# 显示原图

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

for dic in dataset1.create\_dict\_iterator():

plt.subplot(3,3,i)

plt.imshow(dic['image'].asnumpy())

plt.axis('off')

plt.title("Orignal")

i +=1

plt.show()

# 水平翻转图片，随机翻转概率为1

RandomHorizontalFlip = CV.RandomHorizontalFlip(1)

dataset = dataset1.map(operations=RandomHorizontalFlip, input\_columns=["image"])

# 显示翻转之后的图片

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

for dic in dataset.create\_dict\_iterator():

plt.subplot(3,3,i)

plt.imshow(dic['image'].asnumpy())

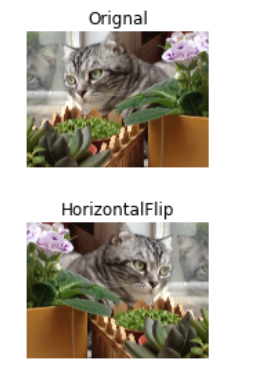
plt.axis('off')

plt.title("HorizontalFlip")

i +=1

plt.show()

输出：



转换图像通道

API：mindspore.dataset.vision.c\_transforms.HWC2CHW

代码：

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

import numpy as np

import mindspore.dataset as ds

# 读取图片，decode=TRUE 解码为（H,W,C）的格式

dataset = ds.ImageFolderDataset('./data/cat', decode= True)

# 原图shape

for x in dataset.create\_dict\_iterator():

print("原图形状：", x['image'].shape)

# 转置输入图像从形状(H, W, C)到形状(C, H, W)

HWC2CHW = CV.HWC2CHW()

dataset1 = dataset.map(operations=HWC2CHW, input\_columns=["image"])

# 通道转换之后的shape

for x in dataset1.create\_dict\_iterator():

print("通道转换之后的形状：",x['image'].shape)

输出：

原图形状： (2448, 3264, 3)

通道转换之后的形状： (3, 2448, 3264)

### 进阶神经网络搭建

卷积神经网络

卷积层

API：mindspore.nn.Conv2d

in\_channels：输入通道

out\_channels：输出通道

kernel\_size ：卷积核高度和宽度；整型或两个整型的tuple；

stride ：步长

pad\_mode ：padding方式（“same”, “valid”, “pad”）. Default: “same”.

Padding：padding补齐数字

has\_bias：是否有偏置项Default: False.

weight\_init： 权重初始化 Default: ‘normal’.

bias\_init：偏置项初始化Default: ‘zeros’.

data\_format：数据形状（ ‘NHWC’ or ‘NCHW’ ）. Default: ‘NCHW’

代码：

import mindspore

import mindspore.nn as nn

from mindspore import Tensor

# mindspore.nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride=1, pad\_mode='same', padding=0, dilation=1,

# group=1, has\_bias=False, weight\_init='normal', bias\_init='zeros', data\_format='NCHW')

# 图片数，通道数，图像高，图像宽

input = Tensor(np.ones([1, 3, 1080, 960]), mindspore.float32)

# 输入通道数为3，输出通道数为24， 卷积核大小为5， 步长为1，padding方式same，有偏置项，权重初始化为normal

net = nn.Conv2d(in\_channels=3, out\_channels=24, kernel\_size=5, stride=1, pad\_mode='same', has\_bias=True, weight\_init='normal')

# 图片数，通道数，图像高，图像宽

output = net(input).shape

print(output)

输出：

(1, 24, 1080, 960)

激活函数

矫正线性单元激活函数

API：mindspore.nn.ReLU

代码：

input\_x = Tensor(np.array([-1, 2, -3, 2, -1]), mindspore.float16)

relu = nn.ReLU()

output = relu(input\_x)

print(output)

输出：

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

池化层

2维数据最大池化

API：mindspore.nn.MaxPool2d

kernel\_size ：池化大小 Default: 1.

Stride ：池化步长 Default: 1.

pad\_mode：padding方式（“same” or “valid”）. Default: “valid”.

data\_format：数据形式（‘NHWC’ or ‘NCHW’ ）. Default: ‘NCHW’

代码：

input\_x = np.random.randint(0, 10, [1, 2, 4, 4])

print(input\_x)

print("-----------------------------------")

# 最大池化，池化大小2x2， 步长为2

pool = nn.MaxPool2d(kernel\_size=2, stride=2)

output = pool(Tensor(input\_x, mindspore.float32))

print(output)

输出：

[[[[8 1 3 3]

[3 7 0 1]

[9 9 0 4]

[7 3 2 7]]

[[2 0 0 4]

[5 5 6 8]

[4 1 4 9]

[8 1 1 7]]]]

-----------------------------------

[[[[8. 3.]

[9. 7.]]

[[5. 8.]

[8. 9.]]]]

Flatten层

API：mindspore.nn.Flatten

Flatten层可以把多维的输入一维化，常用在卷积层到全连接层的过渡。

代码：

from mindspore import Tensor

import mindspore

import mindspore.nn as nn

input = Tensor(np.array([[[1, 1], [2, 2]], [[3, 3], [4, 4]]]), mindspore.float32)

print(input)

print("-----------------------------------")

net = nn.Flatten()

output = net(input)

print(output)

输出：

[[[1. 1.]

[2. 2.]]

[[3. 3.]

[4. 4.]]]

-----------------------------------

[[1. 1. 2. 2.]

[3. 3. 4. 4.]]

全连接层

全连接层

API：mindspore.nn.Dense

in\_channels：输入通道

out\_channels：输出通道

weight\_init：权重初始化，Default 'normal'.

代码：

import mindspore.nn as nn

from mindspore import Tensor

# 构造输入张量

input = Tensor(np.array([[1, 1, 1], [2, 2, 2]]), mindspore.float32)

print(input)

print("-----------------------------------")

# 构造全连接网络，输入通道为3，输出通道为3

net = nn.Dense(in\_channels=3, out\_channels=3, weight\_init=1)

output = net(input)

print(output)

输出：

[[1. 1. 1.]

[2. 2. 2.]]

-----------------------------------

[[3. 3. 3.]

[6. 6. 6.]]

搭建卷积神经网络模型

所有神经网络的基类

API：mindspore.nn.Cell

代码：

import mindspore.nn as nn

class MyCell(nn.Cell):

# 定义算子

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

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

# 卷积层

self.conv = nn.Conv2d()

# 激活函数

self.relu = nn.ReLU()

# 最大池化层

self.max\_pool2d = nn.MaxPool2d()

# 网络展开

self.flatten = nn.Flatten()

# 全连接层

self.fc = nn.Dense()

# 建构网络

def construct(self, x):

x = self.conv(x)

x = self.relu(x)

x = self.max\_pool2d(x)

x = self.flatten(x)

x = self.fc(x)

return x

### 回调函数Callback

Callback回调函数在MindSpore中被实现为一个类，Callback机制类似于一种监控模式，可以帮助用户观察网络训练过程中各种参数的变化情况和网络内部的状态，还可以根据用户的指定，在达到特定条件后执行相应的操作，在训练过程中，Callback列表会按照定义的顺序执行Callback函数。Callback机制让用户可以及时有效地掌握网络模型的训练状态，并根据需要随时作出调整，可以极大地提升用户的开发效率。

ModelCheckpoint

与模型训练过程相结合，保存训练后的模型和网络参数，方便进行再推理或再训练。ModelCheckpoint一般与CheckpointConfig配合使用，CheckpointConfig是一个参数配置类，可自定义配置checkpoint的保存策略。

API：mindspore.train.callback.ModelCheckpoint

prefix：保存文件名称，默认'CKP'；

directory：保存文件路径，默认None；

API：mindspore.train.callback.CheckpointConfig

save\_checkpoint\_steps：每多少步（step）保存一次模型，默认为1；

keep\_checkpoint\_max：最多保存多少个模型, 默认为5；

代码：

from mindspore.train.callback import ModelCheckpoint,CheckpointConfig

# 检查点设置，这里设定每5个step保存一次，最多保存1个次

config\_ck = CheckpointConfig(save\_checkpoint\_steps=5,

keep\_checkpoint\_max=1)

# 模型保存，定义保存文件名以及路径

ckpoint\_cb = ModelCheckpoint(prefix='CKP', directory='./', config=config\_ck) # directory是传入参数

LossMonitor

监控训练过程中的loss变化情况，当loss为NAN或INF时，提前终止训练。可以在日志中输出loss，方便用户查看。

API：mindspore.train.callback.LossMonitor(per\_print\_times=1)

per\_print\_times：每多少步（step）打印一次损失值

代码：

from mindspore.train.callback import LossMonitor

# 每500步打印一次损失值

loss\_cb = LossMonitor(per\_print\_times=500)

TimeMonitor

监控训练过程中每个epoch，每个step的运行时间。

API：mindspore.train.callback.TimeMonitor(data\_size=None)

data\_size：数据集大小

代码：

from mindspore.train.callback import TimeMonitor

# 监控每次迭代的时间

# time\_cb = TimeMonitor(data\_size=data\_train.get\_dataset\_size()) # data\_train.get\_dataset\_size()是传入参数

MindSpore自定义回调函数

当用户有自己的特殊需求时，可以基于Callback基类，自定义满足用户自身需求的回调函数。Callback可以把训练过程中的重要信息记录下来，通过一个字典类型变量cb\_params传递给Callback对象， 用户可以在各个自定义的Callback中获取到相关属性，执行自定义操作。

以下面两个场景为例，介绍自定义Callback回调函数的功能：

实现记录模型训练过程中的评估指标，用户可以在训练过程中，在每个epoch结束时，记录模型评估指标的变化。

代码：

from mindspore.train.callback import Callback

# 记录模型每个epoch的loss

class TrainHistroy(Callback):

"""

记录模型训练时每个epoch的loss的回调函数

Args:

history (list): 传入list以保存模型每个epoch的loss

"""

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

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

self.history = history

# 每个epoch结束时执行

def epoch\_end(self, run\_context):

cb\_params = run\_context.original\_args()

loss = cb\_params.net\_outputs.asnumpy()

self.history.append(loss)

# 记录每次迭代的模型损失值

# train\_hist\_cb = TrainHistroy(hist['loss'])

实现保存训练过程中精度最高的checkpoint文件，用户可以自定义在每一轮迭代后都保存当前精度最高的模型。

代码：

# 测试并记录模型在测试集的loss和accuracy，每个epoch结束时进行模型测试并记录结果，跟踪并保存准确率最高的模型网络参数

class EvalHistory(Callback):

"""

记录模型训练时每个epoch在测试集的loss和accuracy的回调函数，并保存准确率最高的模型网络参数

Args:

model (Cell): 模型，评估loss和accuracy用

loss\_history (list): 传入list以保存模型每个epoch在测试集的loss

acc\_history (list): 传入list以保存模型每个epoch在测试集的accuracy

eval\_data (Dataset): 测试集，评估模型loss和accuracy用

"""

#保存accuracy最高的网络参数

best\_param = None

def \_\_init\_\_(self, model, loss\_history, acc\_history, eval\_data):

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

self.loss\_history = loss\_history

self.acc\_history = acc\_history

self.eval\_data = eval\_data

self.model = model

# 每个epoch结束时执行

def epoch\_end(self, run\_context):

cb\_params = run\_context.original\_args()

res = self.model.eval(self.eval\_data, dataset\_sink\_mode=False)

if len(self.acc\_history)==0 or res['accuracy']>=max(self.acc\_history):

self.best\_param = copy.deepcopy(cb\_params.network)

self.loss\_history.append(res['loss'])

self.acc\_history.append(res['accuracy'])

print('acc\_eval: ',res['accuracy'])

# 训练结束后执行

def end(self, run\_context):

# 保存最优网络参数

best\_param\_path = os.path.join(ckpt\_path, 'best\_param.ckpt')

if os.path.exists(best\_param\_path):

# best\_param.ckpt已存在时MindSpore会覆盖旧的文件，这里修改文件读写权限防止报错

os.chmod(best\_param\_path, stat.S\_IWRITE)

save\_checkpoint(self.best\_param, best\_param\_path)

# 测试并记录模型在验证集的loss和accuracy，并保存最优网络参数

# eval\_hist\_cb = EvalHistory(model = model,

# loss\_history = hist['loss\_eval'],

# acc\_history = hist['acc\_eval'],

# eval\_data = test\_data)

### 模型保存与加载

在模型训练过程中，可以添加检查点(CheckPoint)用于保存模型的参数，以便执行推理及再训练使用。如果想继续在不同硬件平台上做推理，可通过网络和CheckPoint格式文件生成对应的MindIR、AIR和ONNX格式文件。

MindIR：MindSpore的一种基于图表示的函数式IR，定义了可扩展的图结构以及算子的IR表示，它消除了不同后端的模型差异。可以把在Ascend 910训练好的模型，在Ascend 310以及MindSpore Lite端侧上执行推理。

ONNX：全称Open Neural Network Exchange，是一种针对机器学习所设计的开放式的文件格式，用于存储训练好的模型。

我们已在回调函数的ModelCheckpoint中介绍了如何保存CheckPoint格式文件，接下来我们将介绍导出MindIR和ONNX格式文件的方法。

导出MindIR格式文件

如果想跨平台或硬件执行推理(Lite、Ascend 310)，可以通过网络定义和CheckPoint生成MindIR格式模型文件。当前支持基于静态图，且不包含控制流语义的推理网络导出。

input用来指定导出模型的输入shape以及数据类型，input为export方法的入参，代表网络的输入，如果网络有多个输入，需要一同传进export方法。 例如：export(network, Tensor(input1), Tensor(input2), file\_name='network', file\_format='MINDIR')

导出的文件名称会自动添加”.mindir”后缀。

代码：

import numpy as np

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

# resnet = ResNet50()

# # load the parameter into net

# load\_checkpoint("resnet50-2\_32.ckpt", net=resnet)

# input = np.random.uniform(0.0, 1.0, size=[32, 3, 224, 224]).astype(np.float32)

# export(resnet, Tensor(input), file\_name='resnet50-2\_32', file\_format='MINDIR')

导出ONNX格式文件

当有了CheckPoint文件后，如果想继续在昇腾AI处理器、GPU、CPU等多种硬件上做推理，需要通过网络和CheckPoint生成对应的ONNX格式模型文件。

input用来指定导出模型的输入shape以及数据类型，input为export方法的入参，代表网络的输入，如果网络有多个输入，需要一同传进export方法。 例如：export(network, Tensor(input1), Tensor(input2), file\_name='network', file\_format='ONNX')

导出的文件名称会自动添加”.onnx”后缀。

目前ONNX格式导出仅支持ResNet系列网络。

代码：

import numpy as np

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

# resnet = ResNet50()

# # load the parameter into net

# load\_checkpoint("resnet50-2\_32.ckpt", net=resnet)

# input = np.random.uniform(0.0, 1.0, size=[32, 3, 224, 224]).astype(np.float32)

# export(resnet, Tensor(input), file\_name='resnet50-2\_32', file\_format='ONNX')

模型加载

针对仅推理场景可以使用load\_checkpoint把参数直接加载到网络中，以便进行后续的推理验证。

load\_checkpoint方法会把参数文件中的网络参数加载到模型中。加载后，网络中的参数就是CheckPoint保存的。

eval方法会验证训练后模型的精度。

代码：

# resnet = ResNet50()

# load\_checkpoint("resnet50-2\_32.ckpt", net=resnet)

# dateset\_eval = create\_dataset(os.path.join(mnist\_path, "test"), 32, 1) # define the test dataset

# loss = CrossEntropyLoss()

# model = Model(resnet, loss, metrics={"accuracy"})

# acc = model.eval(dataset\_eval)

## 实验总结

本实验介绍了MindSpore的参数初始化方法，运行环境配置，数据集预处理和数据增强，以及卷积神经网络的搭建，如何设置回调函数Callback，模型的保存与加载等，让学生全面掌握MindSpore的开发流程，熟悉MindSpore的进阶用法，能够使用MindSpore搭建并训练复杂神经网络，为后续实验打下良好基础。