哈爾濱工業大學

人工智能实验报告

题　　目　基于 Mindspore 框架与 ModelArts 平台的 MNIST 手写体识别实验

专　　业　人工智能领域方向（2+x 模式）

学　　号　2022110829

姓　　名　杨明达

第一章——基于 Mindspore 框架的模型本地训练及预测（基础版）

一．背景简介/问题描述

本例子会实现一个简单的图片分类的功能，整体流程如下：

1、处理需要的数据集，这里使用了 MNIST 数据集。

1、 定义一个网络，这里我们使用 LeNet 网络。

2、 定义损失函数和优化器。

3、 加载数据集并进行训练，训练完成后，查看结果及保存模型文件。

4、 加载保存的模型，进行推理。

验证模型，加载测试数据集和训练后的模型，验证结果精度。

二．算法介绍

2.1 实验整体框架

本实验基于 Mindspore 框架的模型进行本地训练及预测，整体思路框架是进行数据准备、导入 python 库和模块并配置运行信息、数据处理、定义网络、定义损失函数及优化器、开始训练及验证过程，整体框架图如下图所示。

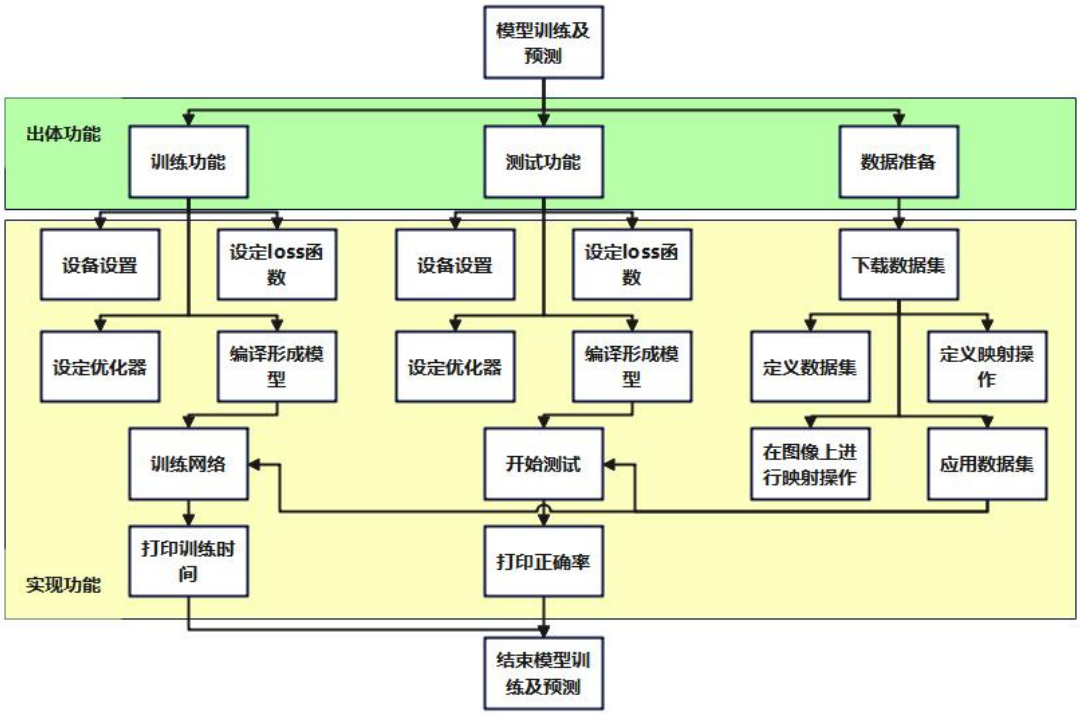

图 1-1 实验整体框架图

2.2 网络介绍——LeNet5 网络

LeNet 网络不包括输入层的情况下，共有 7 层：2 个卷积层、2 个下采样层（池化层）、3 个全连接层。每层都包含不同数量的训练参数，如下图所示：

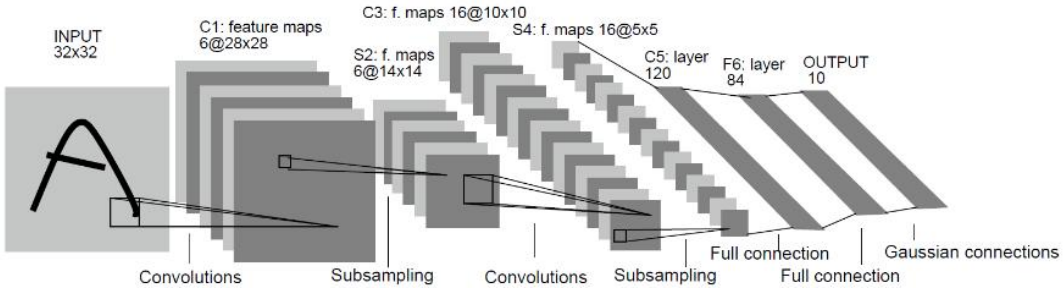

图 1-2 LeNet5 网络结构图

LeNet-5 的输入是 32×32 的灰度图像，只有一个通道。网络结构包括两组卷积层+池化层的组合，两个全连接层，输出是 84×1 维的向量，再通过一个特定的分类器得到预测值。

在 LeNet-5 中，随着网络的深入，数据的高度和宽度逐渐缩小，通道数逐渐增加。LeNet-5 是一个很小的网络，只有约 6 万个参数，现代的神经网络经常有上千万甚至上亿个参数。

三. 算法实现

3.1 数据准备

本实验用到的 MNIST 数据集是由 10 类 28*28 的灰度图片组成，训练数据集包含 60000 张图片，测试数据集包含 10000 张图片。MNIST 数据集官网提供 4 个数据集下载链接，其中前 2 个文件是训练数据需要，后 2 个文件是测试结果需要。将数据集下载并解压到本地路径下，这里将数据集解压分别存放到工作区的./MNIST_Data/train、./MNIST_Data/test 路径下。具体算法如下。

```
from download import download
url = "https://mindspore-website.obs.cn-north-4.myhuaweicloud.com/" \
      "notebook/datasets/MNIST_Data.zip"
path = download(url, "./", kind="zip", replace=True)
```

3.2 导入 Python 库和模块并配置运行信息

在使用前，导入需要的 Python 库，并通过 context.set_context 来配置运行需要的信息，譬如运行模式、后端信息、硬件等信息。导入 context 模块，配置运行需要的信息。

```
parser = argparse.ArgumentParser(description='MindSpore Lenet Example')
# 设备设置
```

```
  parser.add_argument('--device_target', type=str, default="CPU", choices=['Ascend', 'GPU', 'CPU'],
                      help='device where the code will be implemented (default: Ascend)')
  parser.add_argument('--data_path', type=str, default="./MNIST_Data",
                      help='path where the dataset is saved')
  parser.add_argument('--ckpt_path', type=str, default="./ckpt", help='if is test, must provide\
                     path where the trained ckpt file')
  parser.add_argument('--dataset_sink_mode', type=ast.literal_eval, default=True,
                      help='dataset_sink_mode is False or True')

  args = parser.parse_args()


  context.set_context(mode=context.GRAPH_MODE, device_target=args.device_target)
  ds_train = create_dataset(os.path.join(args.data_path, "train"),cfg.batch_size)

  network = LeNet5(cfg.num_classes)
```

3.3 数据处理

由于数据集对于训练非常重要，并且好的数据集可以有效提高训练精度和效率。所以在加载数据集前，本实验对数据集进行一些处理。

在实验中，本文定义一个函数 create_dataset 来创建数据集。在这个函数中，我们定义好需要进行的数据增强和处理操作：

（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
```

（2）定义映射操作

```
  resize_op = CV.Resize((resize_height, resize_width), interpolation=Inter.LINEAR)  # Bilinear mode
  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)
```

（3）在图像上进行映射操作

```
  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)
```

（4）应用数据集

```
  buffer_size = 10000
  mnist_ds = mnist_ds.shuffle(buffer_size=buffer_size)  # 10000 as in LeNet train script
  mnist_ds = mnist_ds.batch(batch_size, drop_remainder=True)

  mnist_ds = mnist_ds.repeat(repeat_size)
```

3.4 定义网络

本实验用到了 LeNet5 网络进行训练和测试。网络中，第一层是卷积层，其输入是原始的图像像素；第二层是池化层 这一层的输入是第一层的输出；第三层是卷积层，本层的输入矩阵大小为 14x14x6，使用的过滤器大小为 5x5，深度为 16。第四层是池化层；第五层是全连接层；第六层是全连接层；第七层是全连接层。具体实现代码如下。

```
  def __init__(self, num_class=10, num_channel=1):
      super(LeNet5, self).__init__()
      #定义卷积层，ReLU 激活函数，平坦层和全连接层
      #conv2d 的输入通道为 1 维，输出为 6 维，卷积核尺寸为 5*5，步长为 1，不适用 padding
      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):
        #构建 Lenet5 架构，x 代表网络的输入
        x = self.max_pool2d(self.relu(self.conv1(x)))
        x = self.max_pool2d(self.relu(self.conv2(x)))
        x = self.flatten(x)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

3.5 定义损失函数

损失函数，又叫目标函数，用于衡量预测值与实际值差异的程度。深度学习通过不停地迭代来缩小损失函数的值。定义一个好的损失函数，可以有效提高模型的性能。定义了损失函数后，可以得到损失函数关于权重的梯度。本实验选择 SoftmaxCrossEntropyWithLogits 损失函数。

```
net_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")
```

3.6 设定优化器

优化器，用于最小化损失函数，从而在训练过程中改进模型。

```
net_opt = nn.Momentum(network.trainable_params(), cfg.lr, cfg.momentum)
time_cb = TimeMonitor(data_size=ds_train.get_dataset_size())
config_ck = CheckpointConfig(save_checkpoint_steps=cfg.save_checkpoint_steps,
                             keep_checkpoint_max=cfg.keep_checkpoint_max)
ckpoint_cb = ModelCheckpoint(prefix="checkpoint_lenet", directory=args.ckpt_path, config=config_ck)
```

四．讨论及结论

4.1 训练及测试验证

图 1-3 训练数据和测试数据

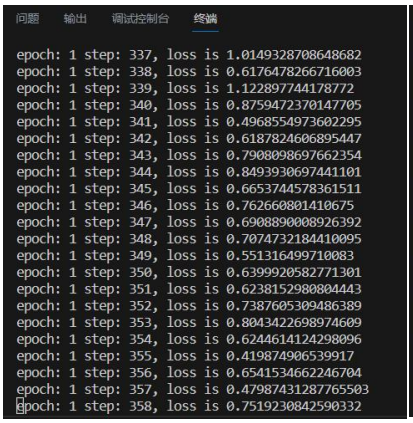

图 1-4 训练过程

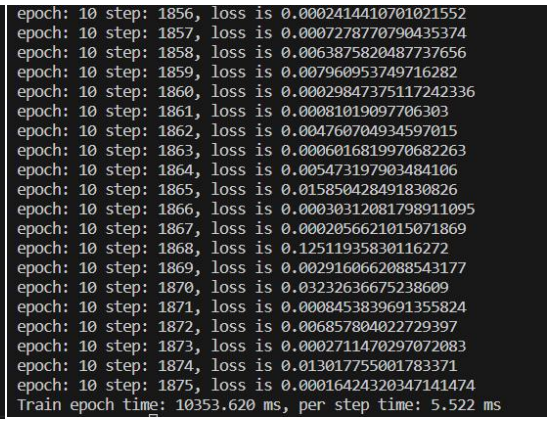

图 1-5 训练结束

图 1-6 预测过程

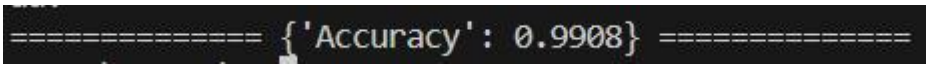

图 1-7 预测准确性

第二章——基于 Mindspore 框架的模型本地训练及预测（进阶版——老师说的拓展实验）

一．背景简介/问题描述

本例子会实现一个简单的图片分类的功能，整体流程如下：

1、处理需要的数据集，这里使用了 MNIST 数据集。

5、 定义一个网络，这里我们使用 LeNet 网络。

6、 定义损失函数和优化器。

7、 加载数据集并进行训练，训练完成后，查看结果及保存模型文件。

8、 加载保存的模型，进行推理。

验证模型，加载测试数据集和训练后的模型，验证结果精度。

除此之外，本实验还在原有数据集上添加噪声，将网络从 LeNet5 更换成 ResNet18。对比预测效果，依次作为拓展部分。

二．算法介绍

2.1 实验整体框架

本实验基于 Mindspore 框架的模型进行本地训练及预测，整体思路框架是进行数据准备、导入 python 库和模块并配置运行信息、数据处理（添加了将 MNIST 转成 PNG、对图像添加噪声、将 PNG 转成 MNIST）、定义网络、定义损失函数及优化器、开始训练及验证过程，整体框架图如下图所示。

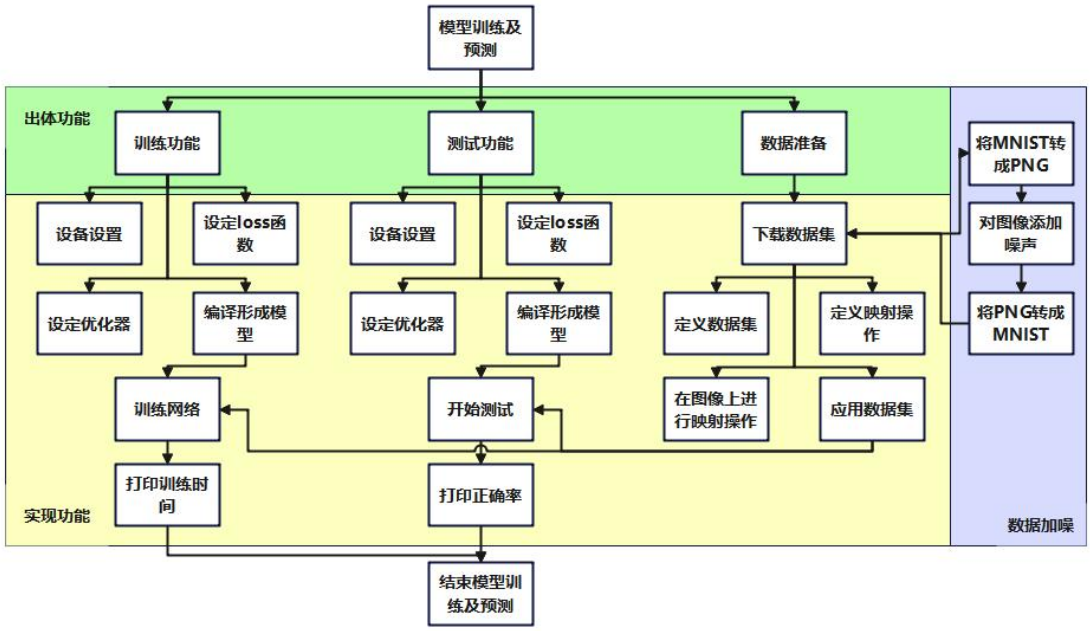

图 2-1 实验整体框架图

2.2 网络介绍——ResNet18 网络

ResNet-18 是一种常用的卷积神经网络（CNN）架构，它以其在训练非常深的网络方面的有效性而闻名，通过解决梯度消失的问题。ResNet-18 的关键思想是引入残差连接或快捷连接。这些连接允许网络学习残差映射，而不是直接学习所需的底层映射关系。通过使用这些快捷连接，ResNet-18 可以有效地训练更深的网络，而不会出现准确率随网络深度增加而降低的问题。在其结构中，有卷积层、残差块、下采样、全局平均池化、全连接层。该网络结构如下图所示。

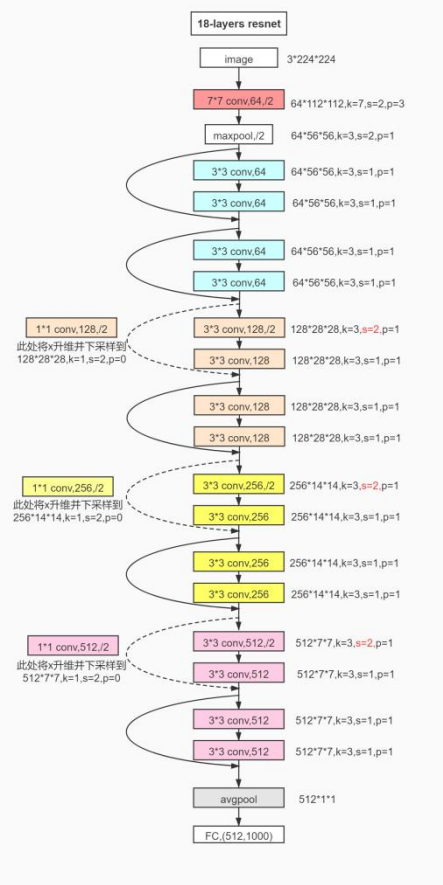

图 2-2 ResNet18 网络结构图

三. 算法实现——由于该部分绝大多数代码与第一部分相同，故在本章省略相同部分，添加拓展的独特算法代码

3.1 将 MNIST 转成 PNG

由于从官网下载的数据集是 MNIST 格式的，所以为了便于对数据图像添加噪声，该算法将其转成 PNG 格式。该操作也可达到批量处理的效果，将数据集存放到主文件夹中，在其中有 10 个子文件夹，分别存放 0 到 9 的上百张图像。

```
import numpy as np
import struct

from PIL import Image
import os

data_file = 'E:/AI_Code/Data/train-images.idx3-ubyte' #需要修改的路径

data_file_size = 47040016
data_file_size = str(data_file_size - 16) + 'B'

data_buf = open(data_file, 'rb').read()

magic, numImages, numRows, numColumns = struct.unpack_from(
    '>IIII', data_buf, 0)
datas = struct.unpack_from(
    '>' + data_file_size, data_buf, struct.calcsize('>IIII'))
datas = np.array(datas).astype(np.uint8).reshape(
    numImages, 1, numRows, numColumns)

label_file = 'E:/AI_Code/Data/train-labels.idx1-ubyte' #需要修改的路径


label_file_size = 60008
label_file_size = str(label_file_size - 8) + 'B'

label_buf = open(label_file, 'rb').read()

magic, numLabels = struct.unpack_from('>II', label_buf, 0)

labels = struct.unpack_from(
    '>' + label_file_size, label_buf, struct.calcsize('>II'))
labels = np.array(labels).astype(np.int64)

datas_root = 'E:/AI_Code/Data/mnist_train' #需要修改的路径
if not os.path.exists(datas_root):
    os.mkdir(datas_root)
```

```
37.
38.  for i in range(10):
39.      file_name = datas_root + os.sep + str(i)
40.      if not os.path.exists(file_name):
41.          os.mkdir(file_name)
42.
43.  for ii in range(numLabels):
44.      img = Image.fromarray(datas[ii, 0, 0:28, 0:28])
45.      label = labels[ii]
46.      file_name = datas_root + os.sep + str(label) + os.sep
     + \
47.          'mnist_train_' + str(ii) + '.png'
48.      img.save(file_name)
```

3.2 对图像加噪声

添加噪声的原理是通过向图像中引入随机扰动，改变像素值或者图像的特征，从而使得图像数据发生变化。本实验采用对 png 格式的图像添加高斯噪声的方法。在本算法中，可以将相同数字的图像全部批量添加噪声，其中噪声的 mean=50，sigma=100，这样的参数能更明显的看到噪声的添加，便于后序实验的观测。

```
1.   import os
2.   import cv2
3.   import numpy as np
4.
5.   def add_gaussian_noise(image, mean=50, sigma=100):
6.       h, w, c = image.shape
7.       noise = np.random.normal(mean, sigma, (h, w, c))
8.       noisy_image = np.clip(image + noise, 0, 255).astype(n
     p.uint8)
9.       return noisy_image
10.
11.  if __name__ == "__main__":
12.      # 输入和输出目录
13.      input_dir = "E:/AI_Code/Data/mnist_test/9/"
14.      output_dir = "E:/AI_Code/Noise/noise_test/9/"
15.
16.      # 确保输出目录存在
17.      if not os.path.exists(output_dir):
18.          os.makedirs(output_dir)
19.
20.      # 处理所有图片
21.      for filename in os.listdir(input_dir):
22.          if filename.endswith(".jpg") or filename.endswith
     (".png"):
```

```
                # 读取图片
                image = cv2.imread(os.path.join(input_dir, filename))
                # 添加高斯噪声
                noisy_image = add_gaussian_noise(image)

                # 保存处理后的图片
                output_path = os.path.join(output_dir, filename)
                cv2.imwrite(output_path, noisy_image)
```

3.3 将噪声图像从 PNG 转成 MNIST

由于 mindspore 框架的模型训练的数据集规定为 MNIST 格式，故该算法将大量的噪声后的 PNG 合并转成 MNIST 格式，在该算法中，为了达到和原数据集相同的目的，批量读取主文件夹中的 10 个子文件夹，将其中的所有图片一起制成 MNIST 格式的数据集。

```
import os
import numpy as np
import struct
import cv2

def load_images_and_labels(data_folder):
    images = []
    labels = []

    label_map = {}  # 用于存储类别名称到数字标签的映射
    label_counter = 0  # 用于生成唯一的数字标签

    for folder_name in sorted(os.listdir(data_folder)):
        label_folder = os.path.join(data_folder, folder_name)
        if os.path.isdir(label_folder):
            if folder_name not in label_map:
                label_map[folder_name] = label_counter
                label_counter += 1
            label = label_map[folder_name]
            for filename in sorted(os.listdir(label_folder)):
                if filename.endswith(".jpg") or filename.endswith(".png"):
                    image_path = os.path.join(label_folder, filename)
                    # 加载图像并转换为灰度图
```

```
                    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
                    image = image.astype(np.uint8)
                    images.append(image)
                    labels.append(label)

    return np.array(images), np.array(labels), label_map


def save_idx3_ubyte(images, output_file):
    # 保存成 idx3-ubyte
    with open(output_file, 'wb') as f:
        magic = 2051
        num_images = len(images)
        if num_images > 0:
            rows, cols = images[0].shape
            f.write(struct.pack('>IIII', magic, num_images, rows, cols))
            for image in images:
                f.write(struct.pack('>' + 'B' * (rows * cols), *image.flatten()))
        else:
            print("No images to save.")

def save_idx1_ubyte(labels, output_file):
    # 保存成 idx1-ubyte
    with open(output_file, 'wb') as f:
        magic = 2049
        num_items = len(labels)
        if num_items > 0:
            f.write(struct.pack('>II', magic, num_items))
            f.write(struct.pack('>' + 'B' * num_items, *labels))
        else:
            print("No labels to save.")

if __name__ == "__main__":
    # 数据文件夹路径
    data_folder = "E:/AI_Code/Noise/noise_test/"

    # 加载图像和生成标签
    images, labels, label_map = load_images_and_labels(data_folder)
```

```
61.
62.      # 保存为 MNIST 数据集格式的文件
63.      save_idx3_ubyte(images, "E:/AI_Code/To_Mnist/t10k-images-idx3-ubyte")
64.      save_idx1_ubyte(labels, "E:/AI_Code/To_Mnist/t10k-labels-idx1-ubyte")
65.
66.      # 保存标签映射
67.      with open("label_map.txt", 'w') as f:
68.          for label_name, label_id in label_map.items():
69.              f.write(f"{label_name}: {label_id}\n")
```

3.4 定义网络

在本章中，将 LeNet 网络更换成更复杂的 ResNet 网络，在算法中，ResidualBlock 类是残差块的定义，它包含两个卷积层、批标准化层、ReLU 激活函数和跳跃连接。残差块通过跳跃连接允许网络学习残差映射，即输入与期望输出之间的差异。ResNet18 类是 ResNet-18 网络的定义，它包含了卷积层、批标准化层、ReLU 激活函数、最大池化层和四个阶段的残差块。conv1 是第一个卷积层，用于处理输入图像，将通道数从 num_channel 转换为 64。bn1 是第一个批标准化层，用于对第一个卷积层的输出进行归一化处理。relu 是 ReLU 激活函数，将非线性特征引入网络。maxpool 是最大池化层，用于降低特征图的空间维度。layer1 到 layer4 这四个阶段，每个阶段包含多个残差块。make_layer 方法用于构建每个阶段。mean 是对特征图在宽度和高度维度上进行平均池化，得到固定长度的特征向量。flatten 将特征向量展平为一维向量，为全连接层做准备。Fc 是全连接层，用于生成最终的类别概率或预测结果。

```
1.  class ResidualBlock(nn.Cell):
2.      def __init__(self, in_channels, out_channels, stride=1):
3.          super(ResidualBlock, self).__init__()
4.          self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, pad_mode='pad')
5.          self.bn1 = nn.BatchNorm2d(out_channels)
6.          self.relu = nn.ReLU()
7.          self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, pad_mode='pad')
8.          self.bn2 = nn.BatchNorm2d(out_channels)
9.          self.downsample = None
10.         if stride != 1 or in_channels != out_channels:
```

```
            self.downsample = nn.SequentialCell([
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, padding=0),
                nn.BatchNorm2d(out_channels)
            ])
        self.add = P.TensorAdd()

    def construct(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample is not None:
            identity = self.downsample(identity)
        out = self.add(out, identity)
        out = self.relu(out)
        return out

class ResNet18(nn.Cell):
    def __init__(self, num_class=10, num_channel=1):
        super(ResNet18, self).__init__()
        self.conv1 = nn.Conv2d(num_channel, 64, kernel_size=7, stride=2, padding=3, pad_mode='pad')
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1, pad_mode='pad')
        self.layer1 = self.make_layer(64, 64, stride=1, blocks=2)
        self.layer2 = self.make_layer(64, 128, stride=2, blocks=2)
        self.layer3 = self.make_layer(128, 256, stride=2, blocks=2)
        self.layer4 = self.make_layer(256, 512, stride=2, blocks=2)
        self.mean = P.ReduceMean(keep_dims=True)
        self.flatten = nn.Flatten()
        self.fc = nn.Dense(512, num_class, weight_init=Normal(0.02))

    def make_layer(self, in_channels, out_channels, stride, blocks):
```

```
            layers = []
            layers.append(ResidualBlock(in_channels, out_channels, stride))
            for _ in range(1, blocks):
                layers.append(ResidualBlock(out_channels, out_channels, stride=1))
            return nn.SequentialCell(layers)

        def construct(self, x):
            x = self.conv1(x)
            x = self.bn1(x)
            x = self.relu(x)
            x = self.maxpool(x)
            x = self.layer1(x)
            x = self.layer2(x)
            x = self.layer3(x)
            x = self.layer4(x)
            x = self.mean(x, (2, 3))
            x = self.flatten(x)
            x = self.fc(x)
            return x
```

四. 讨论及结论

4.1 训练及测试验证

图 2-3 加入噪声后的图片

图 2-4 未加入噪声后的图片

```
d.
============== {'Accuracy': 0.9574} ==============
PS E:\AI_Code>
```

图 2-5 原模型预测加入噪声的图片

```
ad.
============== {'Accuracy': 0.9914} ==============
```

图 2-6 ResNet18 预测未加入噪声的图片

```
ad.
============== {'Accuracy': 0.9914} ==============
```

图 2-7 ResNet18 预测加入噪声的图片

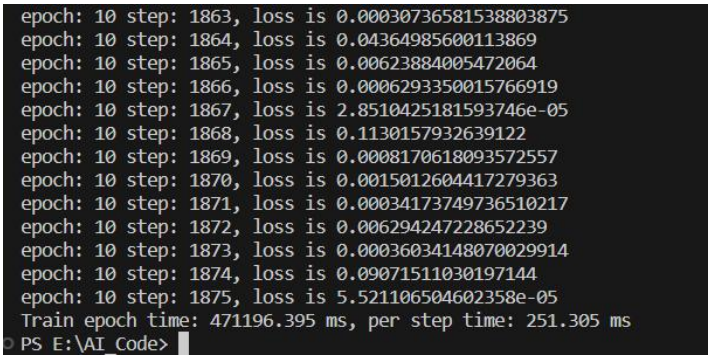

图 2-8 ResNet18 训练时间

由此可知，LeNet 模型预测加入噪声的图片的准确率低于预测未加入噪声的图片的准确率，ResNet18 预测图片的准确率高于 LeNet5 预测图片的准确率并且其在加入噪声后仍然保持高正确率，但 ResNet18 的训练时间远远比 LeNet5 的训练时间长。

第三章——基于 Mindspore 平台框架和 Tensorflow 框架的模型训练及部署

一、背景简介/问题描述

利用 Mindspore 平台框架和 Tensorflow 框架的模型训练数据集及部署，最终进行预测。本案例用于指导用户使用 PyTorch1.8 实现手写数字图像识别，示例采用的数据集为 MNIST 官方数据集。

通过学习本案例，您可以了解如何在 ModelArts 平台上训练作业、部署推理模型并预测的完整流程。

二、算法介绍及实现

2.1 训练脚本

```
# base on https://github.com/pytorch/examples/blob/main/mnist/main.py

from __future__ import print_function

import os
import gzip
import codecs
import argparse
from typing import IO, Union

import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.optim.lr_scheduler import StepLR

import shutil


# 定义网络模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
```

```
27.         self.conv1 = nn.Conv2d(1, 32, 3, 1)
28.         self.conv2 = nn.Conv2d(32, 64, 3, 1)
29.         self.dropout1 = nn.Dropout(0.25)
30.         self.dropout2 = nn.Dropout(0.5)
31.         self.fc1 = nn.Linear(9216, 128)
32.         self.fc2 = nn.Linear(128, 10)
33.
34.     def forward(self, x):
35.         x = self.conv1(x)
36.         x = F.relu(x)
37.         x = self.conv2(x)
38.         x = F.relu(x)
39.         x = F.max_pool2d(x, 2)
40.         x = self.dropout1(x)
41.         x = torch.flatten(x, 1)
42.         x = self.fc1(x)
43.         x = F.relu(x)
44.         x = self.dropout2(x)
45.         x = self.fc2(x)
46.         output = F.log_softmax(x, dim=1)
47.         return output
48.
49.
50. # 模型训练，设置模型为训练模式，加载训练数据，计算损失函数，执行梯度下降
51. def train(args, model, device, train_loader, optimizer, epoch):
52.     model.train()
53.     for batch_idx, (data, target) in enumerate(train_loader):
54.         data, target = data.to(device), target.to(device)
55.         optimizer.zero_grad()
56.         output = model(data)
57.         loss = F.nll_loss(output, target)
58.         loss.backward()
59.         optimizer.step()
60.         if batch_idx % args.log_interval == 0:
61.             print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
62.                 epoch, batch_idx * len(data), len(train_loader.dataset),
63.                 100. * batch_idx / len(train_loader), loss.item()))
```

```
                if args.dry_run:
                    break


    # 模型验证，设置模型为验证模式，加载验证数据，计算损失函数和准确率
    def test(model, device, test_loader):
        model.eval()
        test_loss = 0
        correct = 0
        with torch.no_grad():
            for data, target in test_loader:
                data, target = data.to(device), target.to(device)
                output = model(data)
                test_loss += F.nll_loss(output, target, reduction='sum').item()
                pred = output.argmax(dim=1, keepdim=True)
                correct += pred.eq(target.view_as(pred)).sum().item()

        test_loss /= len(test_loader.dataset)

        print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
            test_loss, correct, len(test_loader.dataset),
            100. * correct / len(test_loader.dataset)))


    # 以下为 pytorch mnist
    # https://github.com/pytorch/vision/blob/v0.9.0/torchvision/datasets/mnist.py
    def get_int(b: bytes) -> int:
        return int(codecs.encode(b, 'hex'), 16)


    def open_maybe_compressed_file(path: Union[str, IO]) -> Union[IO, gzip.GzipFile]:
        """Return a file object that possibly decompresses 'path' on the fly.
            Decompression occurs when argument `path` is a string and ends with '.gz' or '.xz'.
        """
        if not isinstance(path, torch._six.string_classes):
```

```
            return path
        if path.endswith('.gz'):
            return gzip.open(path, 'rb')
        if path.endswith('.xz'):
            return lzma.open(path, 'rb')
        return open(path, 'rb')


SN3_PASCALVINCENT_TYPEMAP = {
        8: (torch.uint8, np.uint8, np.uint8),
        9: (torch.int8, np.int8, np.int8),
        11: (torch.int16, np.dtype('>i2'), 'i2'),
        12: (torch.int32, np.dtype('>i4'), 'i4'),
        13: (torch.float32, np.dtype('>f4'), 'f4'),
        14: (torch.float64, np.dtype('>f8'), 'f8')
}


def read_sn3_pascalvincent_tensor(path: Union[str, IO], strict: bool = True) -> torch.Tensor:
        """Read a SN3 file in "Pascal Vincent" format (Lush file 'libidx/idx-io.lsh').
            Argument may be a filename, compressed filename, or file object.
        """
        # read
        with open_maybe_compressed_file(path) as f:
            data = f.read()
        # parse
        magic = get_int(data[0:4])
        nd = magic % 256
        ty = magic // 256
        assert 1 <= nd <= 3
        assert 8 <= ty <= 14
        m = SN3_PASCALVINCENT_TYPEMAP[ty]
        s = [get_int(data[4 * (i + 1): 4 * (i + 2)]) for i in range(nd)]
        parsed = np.frombuffer(data, dtype=m[1], offset=(4 * (nd + 1)))
        assert parsed.shape[0] == np.prod(s) or not strict
        return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)


```

```
def read_label_file(path: str) -> torch.Tensor:
    with open(path, 'rb') as f:
        x = read_sn3_pascalvincent_tensor(f, strict=False)
    assert(x.dtype == torch.uint8)
    assert(x.ndimension() == 1)
    return x.long()


def read_image_file(path: str) -> torch.Tensor:
    with open(path, 'rb') as f:
        x = read_sn3_pascalvincent_tensor(f, strict=False)
    assert(x.dtype == torch.uint8)
    assert(x.ndimension() == 3)
    return x


def extract_archive(from_path, to_path):
    to_path = os.path.join(to_path, os.path.splitext(os.path.basename(from_path))[0])
    with open(to_path, "wb") as out_f, gzip.GzipFile(from_path) as zip_f:
        out_f.write(zip_f.read())
# --- 以上为 pytorch mnist
# --- end


# raw mnist 数据处理
def convert_raw_mnist_dataset_to_pytorch_mnist_dataset(data_url):
    """
    raw

    {data_url}/
        train-images-idx3-ubyte.gz
        train-labels-idx1-ubyte.gz
        t10k-images-idx3-ubyte.gz
        t10k-labels-idx1-ubyte.gz

    processed

    {data_url}/
        train-images-idx3-ubyte.gz
```

```
        train-labels-idx1-ubyte.gz
        t10k-images-idx3-ubyte.gz
        t10k-labels-idx1-ubyte.gz
        MNIST/raw
            train-images-idx3-ubyte
            train-labels-idx1-ubyte
            t10k-images-idx3-ubyte
            t10k-labels-idx1-ubyte
        MNIST/processed
            training.pt
            test.pt
    """
    resources = [
        "train-images-idx3-ubyte.gz",
        "train-labels-idx1-ubyte.gz",
        "t10k-images-idx3-ubyte.gz",
        "t10k-labels-idx1-ubyte.gz"
    ]

    pytorch_mnist_dataset = os.path.join(data_url, 'MNIST
')

    raw_folder = os.path.join(pytorch_mnist_dataset, 'raw
')
    processed_folder = os.path.join(pytorch_mnist_dataset,
 'processed')

    os.makedirs(raw_folder, exist_ok=True)
    os.makedirs(processed_folder, exist_ok=True)

    print('Processing...')

    for f in resources:
        extract_archive(os.path.join(data_url, f), raw_fo
lder)

    training_set = (
        read_image_file(os.path.join(raw_folder, 'train-i
mages-idx3-ubyte')),
        read_label_file(os.path.join(raw_folder, 'train-l
abels-idx1-ubyte'))
    )
    test_set = (
```

```
            read_image_file(os.path.join(raw_folder, 't10k-images-idx3-ubyte')),
            read_label_file(os.path.join(raw_folder, 't10k-labels-idx1-ubyte'))
        )
        with open(os.path.join(processed_folder, 'training.pt'), 'wb') as f:
            torch.save(training_set, f)
        with open(os.path.join(processed_folder, 'test.pt'), 'wb') as f:
            torch.save(test_set, f)

        print('Done!')


def main():
    # 定义可以接收的训练作业运行参数
    parser = argparse.ArgumentParser(description='PyTorch MNIST Example')

    parser.add_argument('--data_url', type=str, default=False,
                        help='mnist dataset path')
    parser.add_argument('--train_url', type=str, default=False,
                        help='mnist model path')

    parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                        help='input batch size for training (default: 64)')
    parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                        help='input batch size for testing (default: 1000)')
    parser.add_argument('--epochs', type=int, default=14, metavar='N',
                        help='number of epochs to train (default: 14)')
    parser.add_argument('--lr', type=float, default=1.0, metavar='LR',
                        help='learning rate (default: 1.0)')
```

```
    parser.add_argument('--gamma', type=float, default=0.7, metavar='M',
                        help='Learning rate step gamma (default: 0.7)')
    parser.add_argument('--no-cuda', action='store_true', default=False,
                        help='disables CUDA training')
    parser.add_argument('--dry-run', action='store_true', default=False,
                        help='quickly check a single pass')
    parser.add_argument('--seed', type=int, default=1, metavar='S',
                        help='random seed (default: 1)')
    parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                        help='how many batches to wait before logging training status')
    parser.add_argument('--save-model', action='store_true', default=True,
                        help='For Saving the current Model')
    args = parser.parse_args()

    use_cuda = not args.no_cuda and torch.cuda.is_available()

    torch.manual_seed(args.seed)

    # 设置使用 GPU 还是 CPU 来运行算法
    device = torch.device("cuda" if use_cuda else "cpu")

    train_kwargs = {'batch_size': args.batch_size}
    test_kwargs = {'batch_size': args.test_batch_size}
    if use_cuda:
        cuda_kwargs = {'num_workers': 1,
                       'pin_memory': True,
                       'shuffle': True}
        train_kwargs.update(cuda_kwargs)
        test_kwargs.update(cuda_kwargs)

    # 定义数据预处理方法
```

```
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
        ])

    # 将 raw mnist 数据集转换为 pytorch mnist 数据集
    convert_raw_mnist_dataset_to_pytorch_mnist_dataset(args.data_url)

    # 分别创建训练和验证数据集
    dataset1 = datasets.MNIST(args.data_url, train=True, download=False,
                              transform=transform)
    dataset2 = datasets.MNIST(args.data_url, train=False, download=False,
                              transform=transform)

    # 分别构建训练和验证数据迭代器
    train_loader = torch.utils.data.DataLoader(dataset1, **train_kwargs)
    test_loader = torch.utils.data.DataLoader(dataset2, **test_kwargs)

    # 初始化神经网络模型并复制模型到计算设备上
    model = Net().to(device)
    # 定义训练优化器和学习率策略，用于梯度下降计算
    optimizer = optim.Adadelta(model.parameters(), lr=args.lr)
    scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)

    # 训练神经网络，每一轮进行一次验证
    for epoch in range(1, args.epochs + 1):
        train(args, model, device, train_loader, optimizer, epoch)
        test(model, device, test_loader)
        scheduler.step()

    # 保存模型与适配 ModelArts 推理模型包规范
    if args.save_model:

        # 在 train_url 训练参数对应的路径内创建 model 目录
        model_path = os.path.join(args.train_url, 'model')
```

```
307.         os.makedirs(model_path, exist_ok = True)
308.
309.         # 按 ModelArts 推理模型包规范，保存模型到 model 目录内
310.         torch.save(model.state_dict(), os.path.join(model_path, 'mnist_cnn.pt'))
311.
312.         # 复制推理代码与配置文件到 model 目录内
313.         the_path_of_current_file = os.path.dirname(__file__)
314.         shutil.copyfile(os.path.join(the_path_of_current_file, 'infer/customize_service.py'), os.path.join(model_path, 'customize_service.py'))
315.         shutil.copyfile(os.path.join(the_path_of_current_file, 'infer/config.json'), os.path.join(model_path, 'config.json'))
316.
317. if __name__ == '__main__':
318.     main()
```

2.2 推理脚本

```
1.  import os
2.  import log
3.  import json
4.
5.  import torch.nn.functional as F
6.  import torch.nn as nn
7.  import torch
8.  import torchvision.transforms as transforms
9.
10. import numpy as np
11. from PIL import Image
12.
13. from model_service.pytorch_model_service import PTServingBaseService
14.
15. logger = log.getLogger(__name__)
16.
17. # 定义模型预处理
18. infer_transformation = transforms.Compose([
19.     transforms.Resize(28),
20.     transforms.CenterCrop(28),
21.     transforms.ToTensor(),
22.     transforms.Normalize((0.1307,), (0.3081,))
23. ])
```

```
24.
25.  # 模型推理服务
26.  class PTVisionService(PTServingBaseService):
27.
28.      def __init__(self, model_name, model_path):
29.          # 调用父类构造方法
30.          super(PTVisionService, self).__init__(model_name,
     model_path)
31.
32.          # 调用自定义函数加载模型
33.          self.model = Mnist(model_path)
34.
35.          # 加载标签
36.          self.label = [0,1,2,3,4,5,6,7,8,9]
37.
38.      # 接收 request 数据，并转换为模型可以接受的输入格式
39.      def _preprocess(self, data):
40.          preprocessed_data = {}
41.          for k, v in data.items():
42.              input_batch = []
43.              for file_name, file_content in v.items():
44.                  with Image.open(file_content) as image1:

45.                      # 灰度处理
46.                      image1 = image1.convert("L")
47.                      if torch.cuda.is_available():
48.                          input_batch.append(infer_transfor
    mation(image1).cuda())
49.                      else:
50.                          input_batch.append(infer_transfor
    mation(image1))
51.              input_batch_var = torch.autograd.Variable(tor
    ch.stack(input_batch, dim=0), volatile=True)
52.              print(input_batch_var.shape)
53.              preprocessed_data[k] = input_batch_var
54.
55.          return preprocessed_data
56.
57.      # 将推理的结果进行后处理，得到预期的输出格式，该结果就是最
    终的返回值
58.      def _postprocess(self, data):
59.          results = []
60.          for k, v in data.items():
61.              result = torch.argmax(v[0])
```

```
62.                 result = {k: self.label[result]}
63.                 results.append(result)
64.             return results
65.
66.         # 对于输入数据进行前向推理，得到推理结果
67.         def _inference(self, data):
68.
69.             result = {}
70.             for k, v in data.items():
71.                 result[k] = self.model(v)
72.
73.             return result
74.
75.     # 定义网络
76.     class Net(nn.Module):
77.         def __init__(self):
78.             super(Net, self).__init__()
79.             self.conv1 = nn.Conv2d(1, 32, 3, 1)
80.             self.conv2 = nn.Conv2d(32, 64, 3, 1)
81.             self.dropout1 = nn.Dropout(0.25)
82.             self.dropout2 = nn.Dropout(0.5)
83.             self.fc1 = nn.Linear(9216, 128)
84.             self.fc2 = nn.Linear(128, 10)
85.
86.         def forward(self, x):
87.             x = self.conv1(x)
88.             x = F.relu(x)
89.             x = self.conv2(x)
90.             x = F.relu(x)
91.             x = F.max_pool2d(x, 2)
92.             x = self.dropout1(x)
93.             x = torch.flatten(x, 1)
94.             x = self.fc1(x)
95.             x = F.relu(x)
96.             x = self.dropout2(x)
97.             x = self.fc2(x)
98.             output = F.log_softmax(x, dim=1)
99.             return output
100.
101.
102. def Mnist(model_path, **kwargs):
103.     # 生成网络
104.     model = Net()
105.
```

```
    # 加载模型
    if torch.cuda.is_available():
        device = torch.device('cuda')
        model.load_state_dict(torch.load(model_path, map_location="cuda:0"))
    else:
        device = torch.device('cpu')
        model.load_state_dict(torch.load(model_path, map_location=device))

    # CPU 或者 GPU 映射
    model.to(device)

    # 声明为推理模式
    model.eval()

    return model
```

三、实验步骤

3.1 准备数据集

本案例使用的数据是 MNIST 数据集：

“train-images-idx3-ubyte.gz”：训练集的压缩包文件，共包含 60000 个样本。

“train-labels-idx1-ubyte.gz”：训练集标签的压缩包文件，共包含 60000 个样本的类别标签。

“t10k-images-idx3-ubyte.gz”：验证集的压缩包文件，共包含 10000 个样本。

“t10k-labels-idx1-ubyte.gz”：验证集标签的压缩包文件，共包含 10000 个样本的类别标签。

3.2 准备训练文件和推理文件

在本地电脑中创建训练脚本“train.py”和推理脚本“customize_service.py”内容见上一节“算法介绍及实现”

3.3 创建 OBS 桶并上传文件

将上一步中的数据和代码文件、推理代码文件与推理配置文件，从本地上传到 OBS 桶中。在 ModelArts 上运行训练作业时，需要从 OBS 桶中读取数据和代码文件。

1.登录 OBS 管理控制台，按照如下示例创建 OBS 桶和文件夹

2.上传 Step1 准备训练数据下载的 MNIST 数据集压缩包文件到 OBS 中。

3.上传训练脚本“train.py”到“mnist-code”文件夹中。

4.上传推理脚本“customize_service.py”和推理配置文件“config.json”到“infer”文件中。

3.4 创建训练作业

1.登录 ModelArts 管理控制台，选择和 OBS 桶相同的区域。

2.在“全局配置”中检查当前账号是否已完成访问授权的配置。如未完成，请参考使用委托授权。针对之前使用访问密钥授权的用户，建议清空授权，然后使用委托进行授权。

3.在左侧导航栏选择“训练管理 > 训练作业”进入训练作业页面，单击“创建训练作业”。

4.填写创建训练作业相关信息。

5.单击“提交”，确认训练作业的参数信息，确认无误后单击“确定”。

页面自动返回“训练作业”列表页，当训练作业状态变为“已完成”时，即完成了模型训练过程。

6.单击训练作业名称，进入作业详情界面查看训练作业日志信息，观察日志是否有明显的 Error 信息，如果有则表示训练失败，请根据日志提示定位原因并解决。

7.在训练详情页左下方单击训练输出路径（如图 11），跳转到 OBS 目录，查看是否存在 model 文件夹，且 model 文件夹中是否有生成训练模型（如图 12）。如果未生成 model 文件夹或者训练模型，可能是训练输入数据不完整导致，请检查训练数据上传是否完整，并重新训练。

3.5 推理部署

模型训练完成后，可以创建 AI 应用，将 AI 应用部署为在线服务。

1.在 ModelArts 管理控制台，单击左侧导航栏中的“AI 应用管理>AI 应用”，进入“我的 AI 应用”页面，单击“创建”。

2.在“创建 AI 应用”页面，填写相关参数，然后单击“立即创建”。在“元模型来源”中，选择“从训练中选择”页签，选择步骤 5：创建训练作业中完成

的训练作业，勾选“动态加载”。

3.在 AI 应用列表页面，当 AI 应用状态变为“正常”时，表示 AI 应用创建成功。单击 AI 应用名称左侧的单选按钮，在列表页底部展开“版本列表”，单击操作列“部署>在线服务”，将 AI 应用部署为在线服务。

4.在“部署”页面，参考下图填写参数，然后根据界面提示完成在线服务创建。本案例适用于 CPU 规格，节点规格需选择 CPU。完成服务部署后，返回在线服务页面列表页，等待服务部署完成，当服务状态显示为“运行中”，表示服务已部署成功。

3.6 预测结果

1.在“在线服务”页面，单击在线服务名称，进入服务详情页面。

2.单击“预测”页签，请求类型选择“multipart/form-data”，请求参数填写“image”，单击“上传”按钮上传示例图片，然后单击“预测”。

预测完成后，预测结果显示区域将展示预测结果，根据预测结果内容，可识别出此图片的数字是“2”。

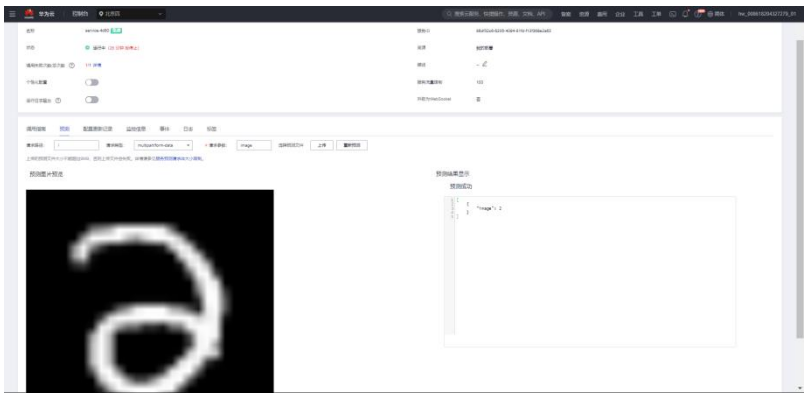

图 3-1 预测 2（正确）

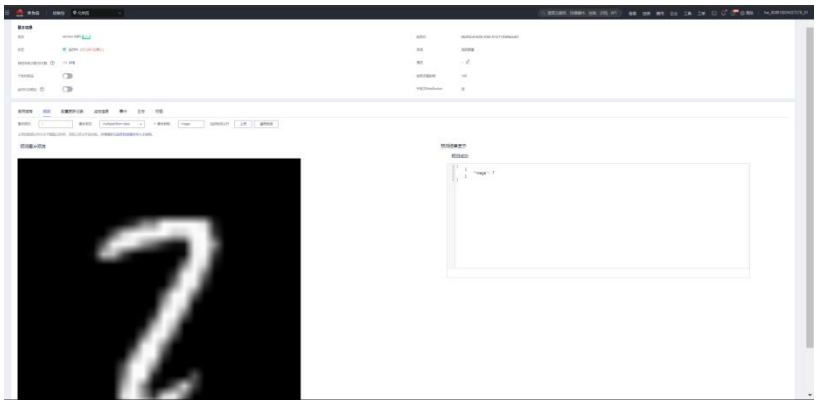

图 3-1 预测 2（错误）

四、讨论及结论

4.1 分析模型预测正确率仍达不到100%（老师留的拓展任务）

利用神经网络构建模型预测手写数字，虽然正确率很高，但仍有一部分数据预测错误，造成这个现象的原因有五方面：

第一，手写数字的输入数据可能存在噪声或模糊性，这些因素会导致一些样本的特征被模糊或混淆，从而增加了分类错误的可能性。

第二，神经网络需要足够的训练数据来学习手写数字的模式和特征。如果训练数据集过小或者样本分布不均衡，网络可能无法充分学习到所有可能的变化和情况，从而导致一些数据预测错误。

第三，神经网络的结构和复杂度对于学习和表示复杂模式非常重要。如果网络的规模不够大或层数不够深，它可能无法捕捉到手写数字数据中的更高级特征和关系，从而导致一些错误的预测。

第四，神经网络在训练过程中可能过度拟合训练数据，即过于专注于训练数据的细节而无法泛化到新的未见过的数据。这会导致网络在测试或实际应用中出现错误的预测。

第五，训练神经网络是一个迭代的优化过程，可能会陷入局部最优解而无法达到全局最优解。这可能导致网络在某些情况下做出错误的预测。

4.2 实验体会

通过本次实验（基于Mindspore框架与MondelArts平台的MNIST手写体识别实验），我完成了基于Mindspore框架的模型本地训练及预测，也完成了基于Modelarts平台和Tensorflow框架的模型训练及部署，又在本地进行了模型训练的拓展任务（对数据集添加噪声，更改更为复杂的神经网络）。在本次实验中，我直接接触了人工智能相关的工作，充分理解模型识别手写体的完整流程，锻炼了自己的编程能力和解决问题的能力。

参考文献

[1] https://www.mindspore.cn/install/

[2] http://yann.lecun.com/exdb/mnist/

[3] https://blog.csdn.net/qq_36076233/article/details/122881833

[4] https://support.huaweicloud.com/bestpractice-modelarts/modelarts_10_0080.html

[5] http://yann.lecun.com/exdb/lenet/