实验 6 手写数字识别之数据处理

前面使用“横纵式”教学法中的纵向极简方案快速完成手写数字识别任务的建模，但模型测试效果并未达成预期。下面从横向展开，逐个环节优化，以达到最优训练效果，如图 1 所示。

图 1 数据处理优化

前面通过调用飞桨提供的 paddle.vision.datasets.MNIST API 加载 MNIST 数据集。但实践中，我们面临的任务和数据环境千差万别，通常需要自己编写适合当前任务的数据处理程序，一般涉及如下五个环节：读入数据，划分数据集，生成批次数据，训练样本集乱序，校验数据有效性。

6.1 读入数据并划分数据集

在实际应用中，保存到本地的数据存储格式多种多样，如 MNIST 数据集以 json 格式存储在本地，其数据存储结构如图 2 所示。

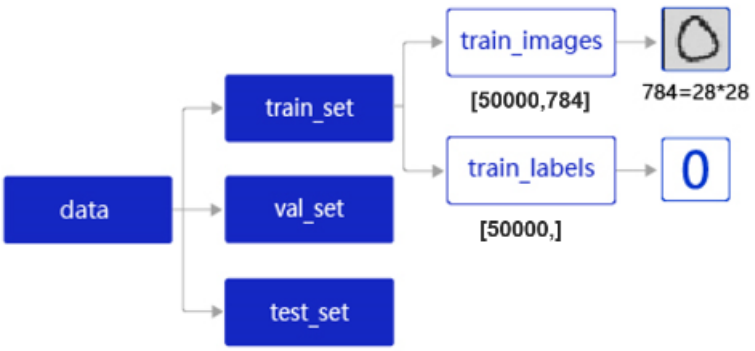

图 2 MNIST 数据集的存储结构

data 包含三个元素的列表：训练集 train_set、验证集 val_set、测试集 test_set，分别为 50 000 条训练样本、10 000 条验证样本和 10 000 条测试样本。每条样本数据都包含手写数字的图像和对应的标签。

训练集：用于确定模型参数。

验证集：用于调节模型超参数（如多个网络结构、正则化权重的最优选择）。

测试集：用于估计应用效果（没有在模型中应用过的数据，更贴近模型在真实场景应用的效果）。

train_set 包含两个元素的列表：train_images、train_labels。

train_images：[50 000, 784]的二维列表，包含 50 000 张图片。每张图片用一个长度为 784 的向量表示，内容是 28×28 像素的灰度值（黑白图片）。

train_labels：[50 000,]的列表，表示这些图片对应的分类标签，即 0~9 之间的一个数字。

在本地目录下读取文件名称为 mnist.json.gz 的 MNIST 数据，并拆分成训练集、验证集和测试集，代码实现如下。

```
# 数据处理部分之前的代码，加入部分数据处理的库
import paddle
from paddle.nn import Linear
import paddle.nn.functional as F
import os
import gzip
import json
import random
import numpy as np
```

```
# 声明数据集文件位置
datafile = 'D:/datasets/pics/projects/digitalRecognition/mnist.json.gz'
print('loading mnist dataset from {} ......'.format(datafile))
# 加载json数据文件
data = json.load(gzip.open(datafile))
print('mnist dataset load done')
# 读取到的数据区分训练集，验证集，测试集
train_set, val_set, eval_set = data

# 观察训练集数据
imgs, labels = train_set[0], train_set[1]
print("训练数据集数量: ", len(imgs))

# 观察验证集数量
imgs, labels = val_set[0], val_set[1]
print("验证数据集数量: ", len(imgs))

# 观察测试集数量
imgs, labels = val= eval_set[0], eval_set[1]
print("测试数据集数量: ", len(imgs))
print(len(imgs[0]))
```

6.2 训练样本乱序、生成批次数据

（1）训练样本乱序：先将样本按顺序进行编号，建立 ID 集合 index_list。然后将 index_list 乱序，最后按乱序后的顺序读取数据。

说明：通过大量实验发现，模型对最后出现的数据印象更加深刻。训练数据导入后，越接近模型训练结束，最后几个批次数据对模型参数的影响越大。为了避免模型记忆影响训练效果，需要进行样本乱序操作。

（2）生成批次数据：先设置合理的 batch size，再将数据转变成符合模型输入要求的 np.array 格式返回。同时，在返回数据时将 Python 生成器设置为 yield 模式，以减少内存占用。

在执行如上两个操作之前，需要先将数据处理代码封装成 load_data 函数，方便后续调用。load_data 有三种模型：train、valid、eval，分为对应返回的数据是训练集、验证集、测试集。

```python
imgs, labels = train_set[0], train_set[1]
print("训练数据集数量: ", len(imgs))
# 获得数据集长度
imgs_length = len(imgs)
# 定义数据集每个数据的序号，根据序号读取数据
index_list = list(range(imgs_length))
# 读入数据时用到的批次大小
BATCHSIZE = 100

# 随机打乱训练数据的索引序号
random.shuffle(index_list)

# 定义数据生成器，返回批次数据
def data_generator():
    imgs_list = []
    labels_list = []
    for i in index_list:
        # 将数据处理成希望的类型
        img = np.array(imgs[i]).astype('float32')
        label = np.array(labels[i]).astype('float32')
        imgs_list.append(img)
        labels_list.append(label)
        if len(imgs_list) == BATCHSIZE:
            # 获得一个batchsize的数据，并返回
            yield np.array(imgs_list), np.array(labels_list)
            # 清空数据读取列表
            imgs_list = []
            labels_list = []

    # 如果剩余数据的数目小于BATCHSIZE，
    # 则剩余数据一起构成一个大小为len(imgs_list)的mini-batch
    if len(imgs_list) > 0:
        yield np.array(imgs_list), np.array(labels_list)
    return data_generator
```

```python
# 声明数据读取函数，从训练集中读取数据
train_loader = data_generator
# 以迭代的形式读取数据
for batch_id, data in enumerate(train_loader()):
    image_data, label_data = data
    if batch_id == 0:
        # 打印数据shape和类型
        print("打印第一个batch数据的维度:")
        print("图像维度: {}, 标签维度: {}".format(image_data.shape, label_data.shape))
    break
```

6.3 校验数据有效性

在实际应用中，原始数据可能存在标注不准确、数据杂乱或格式不统一等情况。因此在完成数据处理流程后，还需要进行数据校验，一般有两种方式：

机器校验：加入一些校验和清理数据的操作。

人工校验：先打印数据输出结果，观察是否是设置的格式。再从训练的结果验证数据处理和读取的有效性。

6.3.1 机器校验

如果数据集中的图片数量和标签数量不等，说明数据逻辑存在问题，可使用 assert 语句校验图像数量和标签数据是否一致。

```python
imgs_length = len(imgs)

assert len(imgs) == len(labels), \
        "length of train_imgs({}) should be the same as train_labels({})".format(len(imgs), len(labels))
```

6.3.2 人工校验

人工校验是指打印数据输出结果，观察是否是预期的格式。实现数据处理和加载函数后，我们可以调用它读取一次数据，观察数据的形状和类型是否与函数中设置的一致。

```
# 声明数据读取函数，从训练集中读取数据
train_loader = data_generator
# 以迭代的形式读取数据
for batch_id, data in enumerate(train_loader()):
    image_data, label_data = data
    if batch_id == 0:
        # 打印数据shape和类型
        print("打印第一个batch数据的维度，以及数据的类型:")
        print("图像维度: {}, 标签维度: {}, 图像数据类型: {}, 标签数据类型: {}".format(image_data.shape, label_data.shape, type(image_data), type(label_data)))
    break
```

6.4 封装数据读取与处理函数

将读取数据、划分数据集、打乱训练数据、构建数据读取器以及数据数据校验这些步骤放在一个函数中实现，方便在神经网络训练时直接调用。

```
def load_data(mode='train'):
    datafile = 'D:/datasets/pics/projects/digitalRecognition/mnist.json.gz'
    print('loading mnist dataset from {} ......'.format(datafile))
    # 加载json数据文件
    data = json.load(gzip.open(datafile))
    print('mnist dataset load done')

    # 读取到的数据区分训练集，验证集，测试集
    train_set, val_set, eval_set = data
    if mode=='train':
        # 获得训练数据集
        imgs, labels = train_set[0], train_set[1]
    elif mode=='valid':
        # 获得验证数据集
        imgs, labels = val_set[0], val_set[1]
    elif mode=='eval':
        # 获得测试数据集
        imgs, labels = eval_set[0], eval_set[1]
    else:
        raise Exception("mode can only be one of ['train', 'valid', 'eval']")
    print("训练数据集数量: ", len(imgs))

    # 校验数据
    imgs_length = len(imgs)

    assert len(imgs) == len(labels), \
          "length of train_imgs({}) should be the same as train_labels({})".format(len(imgs), len(labels))
```

```
    # 获得数据集长度
    imgs_length = len(imgs)

    # 定义数据集每个数据的序号，根据序号读取数据
    index_list = list(range(imgs_length))
    # 读入数据时用到的批次大小
    BATCHSIZE = 100

    # 定义数据生成器
    def data_generator():
        if mode == 'train':
            # 训练模式下打乱数据
            random.shuffle(index_list)
        imgs_list = []
        labels_list = []
        for i in index_list:
            # 将数据处理成希望的类型
            img = np.array(imgs[i]).astype('float32')
            label = np.array(labels[i]).astype('float32')
            imgs_list.append(img)
            labels_list.append(label)
            if len(imgs_list) == BATCHSIZE:
                # 获得一个batchsize的数据，并返回
                yield np.array(imgs_list), np.array(labels_list)
                # 清空数据读取列表
                imgs_list = []
                labels_list = []

        # 如果剩余数据的数目小于BATCHSIZE，
        # 则剩余数据一起构成一个大小为len(imgs_list)的mini-batch
        if len(imgs_list) > 0:
            yield np.array(imgs_list), np.array(labels_list)
    return data_generator
```

下面定义一层神经网络，利用定义好的数据处理函数，完成神经网络的训练。

```
# 数据处理部分之后的代码,数据读取的部分调用Load_data函数
# 定义网络结构,同上一节所使用的网络结构
class MNIST(paddle.nn.Layer):
    def __init__(self):
        super(MNIST, self).__init__()
        # 定义一层全连接层，输出维度是1
        self.fc = paddle.nn.Linear(in_features=784, out_features=1)

    def forward(self, inputs):
        outputs = self.fc(inputs)
        return outputs
```

```
# 训练配置，并启动训练过程
def train(model):
    model = MNIST()
    model.train()
    #调用加载数据的函数
    train_loader = load_data('train')
    opt = paddle.optimizer.SGD(learning_rate=0.001, parameters=model.parameters())
    EPOCH_NUM = 10
    for epoch_id in range(EPOCH_NUM):
        for batch_id, data in enumerate(train_loader()):
            #准备数据，变得更加简洁
            images, labels = data
            images = paddle.to_tensor(images)
            labels = paddle.to_tensor(labels)

            #前向计算的过程
            predits = model(images)

            #计算损失，取一个批次样本损失的平均值
            loss = F.square_error_cost(predits, labels)
            avg_loss = paddle.mean(loss)

            #每训练了200批次的数据，打印下当前Loss的情况
            if batch_id % 200 == 0:
                print("epoch: {}, batch: {}, loss is: {}".format(epoch_id, batch_id, avg_loss.numpy()))

            #后向传播，更新参数的过程
            avg_loss.backward()
            opt.step()
            opt.clear_grad()

    # 保存模型
    paddle.save(model.state_dict(), 'D:/datasets/pics/projects/digitalRecognition/mnist.pdparams')
# 创建模型
model = MNIST()
# 启动训练过程
train(model)
```

6.5 数据增强

但在实际任务中，原始数据集未必完全含有解决任务所需要的充足信息。这时候，通过分析任务场景，有针对性的做一些数据增强的策略，往往可以显著的提高模型效果。

数据增强是一种挖掘数据集潜力的方法，可以让数据集蕴含更多让模型有效学习的信息。这些方法是领域和任务特定的，通过分析任务场景的复杂性和当前数据集的短板，对现有数据做有针对性的修改，以提供更加多样性的、匹配任务场景复杂性的新数据。

6.5.1 基础的数据增强方法

图 3 所示为一些基础的图像增强方法，如果我们发现数据集中的猫均是标准姿势，而真实场景中的猫时常有倾斜身姿的情况，那么在原始图片数据的基础上采用旋转的方法造一批数据加入到数据集会有助于提升模型效果。类似的，如果数据集中均是高清图片，而真实场景中经常有拍照模糊或曝光异常的情况，则采用降采样和调整饱和度的方式造一批数据，会

有助于提升模型的效果。

图 3 基础的图像增强方法

（1）亮度调整

```
import numpy as np
from PIL import Image
from paddle.vision.transforms import functional as F

img_path = "D:/datasets/pics/projects/digitalRecognition/cat.jpeg"

image=Image.open(img_path)

# adjust_brightness对输入图像进行亮度值调整
new_img = F.adjust_brightness(image, 0.4)

# 显示图像
display(image.resize((300,400)))
display(new_img.resize((300,400)))
```

（2）色调调整

```
import numpy as np
from PIL import Image
from paddle.vision.transforms import functional as F

img_path = "D:/datasets/pics/projects/digitalRecognition/cat.jpeg"

image=Image.open(img_path)

# adjust_hue对输入图像进行色调的调整
F.adjust_hue(image, 0.3)

# 显示图像
display(image.resize((300,400)))
display(new_img.resize((300,400)))
```

（3）随机旋转

```
import numpy as np
from PIL import Image
from paddle.vision.transforms import RandomRotation

img_path = "D:/datasets/pics/projects/digitalRecognition/cat.jpeg"

image=Image.open(img_path)

# RandomRotation依据90度，按照均匀分布随机产生一个角度对图像进行旋转
transform = RandomRotation(90)

new_img = transform(image)

# 显示图像
display(image.resize((300,400)))
display(new_img.resize((300,400)))
```