模式识别实验报告

|  |  |
| --- | --- |
| 专业： | 信息科学与工程 |
| 学号： | 61821326 |
| 年级： | 2 |
| 姓名： | 李睿刚 |

签名：

时间：

**实验一 朴素贝叶斯分类**

1. **问题描述**
2. 利用朴素贝叶斯算法，对NMIST数据集中的测试集进行分类。

* MNIST数据集本身以二进制形式保存，所以首先需要选择合适的编程语言编写读写二进制数据的程序完成对图片、标记信息的初步提取工作。将图像二值化，得到可以用于分类的28×28个特征向量以及对应的标签数据，之后便可以交由贝叶斯分类器进行学习。
* 朴素贝叶斯算法：利用贝叶斯公式，对 作出预测。由于𝑝(𝑥)为一固定值，所以一般不在计算过程中求得 𝑝(𝑥)的具体值。在实际运用中，为了方便计算，通常假设数据特征之间相互独立，即。

1. 基于 MindSpore平台提供的官方模型库，对相同的数据集进行训练，并与自己独立实现的算法对比结果（包括但不限于准确率、算法迭代收敛次数等指标），并分析结果中出现差异的可能原因，给出使用 MindSpore的心得和建议。
2. （加分项）使用 MindSpore平台提供的相似任务数据集（例如，其他的分类任务数据集）测试自己独立实现的算法并与 MindSpore平台上的官方实现算法进行对比，进一步分析差异及其成因。
3. **实现步骤与流程**

**Task1（实现朴素贝叶斯分类器）**

1. 读取数据集

经资料查找（和ai辅助），MNIST数据集可直接用gzip库和numpy库从压缩文件中读取数据。每张图片直接读取为28\*28的向量，便于遍历。

1. 数据预处理

观察MNIST数据可知，其实每个像素点的值就是0~255的unit8，而不是实验指导书上的[0~1]浮点数。因此只要进行二值化即可。由于图片黑白占据了大部分比例，没有必要使用大津法，直接使用固定阈值127的二值化方法。

1. 收集朴素贝叶斯分类器所需的数据

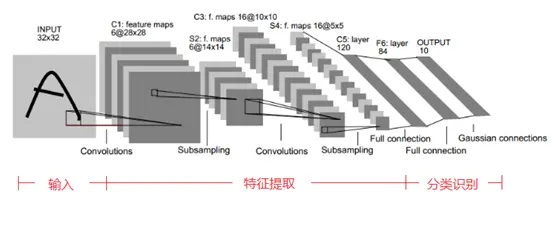
* 核心思想：利用大数定律，用频率代替概率。实际中，为了防止下溢出，对每个概率取了自然对数，并对贝叶斯公式进行了相应的改动。

1. 统计的每个数字的出现概率，取对数存于数组pw。
2. 计算数字i的样本中，同一位置j像素为白色的概率，取对数存于num\_images\_p[i][j]（还要存储黑色的概率，但实际实现中为了节省空间使用了技巧）。为了防止对数的真数为0，对概率使用了拉普拉斯平滑，即频率的“分子+1”“分母+10”。
3. 分类与测试

对输入的28\*28维待分类样本，先进行二值化，再对每个类，利用贝叶斯公式的对数形式计算属于每个类的,选取最大的一项对应的类作为该样本的分类结果。根据分类结果得到模型在测试上的正确率，以评估模型的性能。

**Task2（使用MindSpore官方模型库训练）**

使用Lenet模型。Lenet是一个 7 层的神经网络，包含 3 个卷积层，2 个池化层，1 个全连接层。其中所有卷积层的所有卷积核都为 5x5，步长strid=1，池化方法都为全局 pooling，激活函数为 Sigmoid，网络结构如下：



MindSpore内置了lenet模型，直接调用即可。具体来说，需要选择网络类型（lenet）、优化器（动量）、损失函数（交叉熵损失函数）。设计训练流程时，同样要注意实时保存进度。具体见代码。

1. **实验结果与分析**

**Task1**

实验结果请见“bayes.html”，朴素贝叶斯分类器的正确率竟然达到了84.26%。相比于一开始的设计，最终的分类器有两点主要改进：

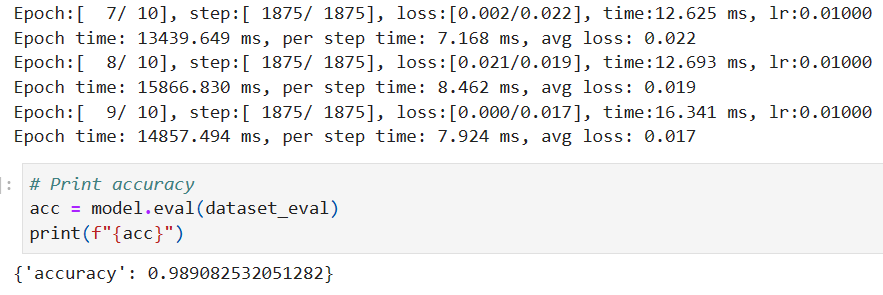
1. 二值化。观察数据可得，样本中有灰色像素。实验之初准备使用Parzen窗方法得到每个像素的颜色分布。但是实现的过程中，发现数据离散分布（只在整数处有分布）且有界，故窗口小不了；而且灰度值主要分布在0和255，中间的样本其实很少，统计精度不高。且样本有界，如果使用正态分布这类“无界”的窗函数，对边界的统计就会有很明显的损失。于是从求分布退化为二值化求概率。

2. 对数。没取对数时在测试集的正确率仅为11.37%，查看分类情况时才发现，都识别成了数字0。进一步查看每一类的“伪后验概率”，结果都是0，这才明白发生了“下溢出”。于是使用对数形式。为了防止真数为0，经过资料搜索，选用了最简单的拉普拉斯平滑。测试中发现（选取了三组参数运行），只要平滑的参数不是特别大，对最终的正确率结果没有影响。

虽然在测试集的精度很高，但对我自行创建的图片分类结果，依旧全部识别为了0。我自己的图片和数据集的图片的主要区别在于粗细和数字大小。说明该分类器的普适性不够强。

**Task2（MindSpore）**

经过10轮训练后，模型正确率高达98.9%。Mindspore官方模型对MNIST数据点的解决方案Lenet是神经网络，用到了卷积池化等对专用于图像的处理层，且能迭代，自然比简单的“朴素贝叶斯”效果好。朴素贝叶斯不能迭代，是模型的缺点。此外，假设每个像素独立，也不是很符合实际。



1. **MindSpore学习使用心得体会**
2. Mindspore有着不同于pytorch的函数组织形式。例如在搭建网络时候，对于数据集的transform并不是像pytorch的compose那样直接将module串联(即pipeline那样串联好)，而是把transform 函数都装在list中，输入mindspore封装好的dataset.map函数进行处理。在数据处理上这点要格外注意。
3. Mindspore使用过程中发现还是有较多API不支持GPU，使用过程中要注意对于代码时间算法复杂度的控制，训练上尽量用高效一点的优化方式（例如速度较快的Adam等）
4. 对于loss函数和梯度下降之间的衔接，权重更新等或许我还需要继续深入学习使用文档。对于模型的保存上，mindspore不同于pytorch的‘save check point’比较新颖，使用上要辨别清楚与其它框架的不同。
5. **代码附录**

**bayes.ipynb【将各代码块粘贴而得以下代码。实际使用jupyter notebook。】**

import gzip

import numpy as np

def load\_data():    # 读取到npArray, 每维是一个28\*28, 0~255取值的数组。

    with gzip.open('../data/train-images-idx3-ubyte.gz', 'rb') as f:

        train\_images = np.frombuffer(f.read(), np.uint8, offset=16).reshape(-1, 28\*28)

    with gzip.open('../data/train-labels-idx1-ubyte.gz', 'rb') as f:

        train\_labels = np.frombuffer(f.read(), np.uint8, offset=8)

    with gzip.open('../data/t10k-images-idx3-ubyte.gz', 'rb') as f:

        test\_images = np.frombuffer(f.read(), np.uint8, offset=16).reshape(-1, 28\*28)

    with gzip.open('../data/t10k-labels-idx1-ubyte.gz', 'rb') as f:

        test\_labels = np.frombuffer(f.read(), np.uint8, offset=8)

    return train\_images, train\_labels, test\_images, test\_labels

train\_images, train\_labels, test\_images, test\_labels = load\_data()

# 对图像二值化，阈值取127

train\_images = np.where(train\_images > 127, 1, 0)

test\_images = np.where(test\_images > 127, 1, 0)

# 可视化一部分数据检验解析正确性

# import matplotlib.pyplot as plt

# fig, axs = plt.subplots(4, 4, figsize=(10, 10))

# for i in range(16):

# axs[i//4, i%4].imshow(train\_images[i].reshape(28, 28), cmap='gray')

# axs[i//4, i%4].axis('off')

# axs[i//4, i%4].set\_title(train\_labels[i])

# plt.show()

# 统计每个数字的频率, pw[i]=ln(p(w\_i))

# 取对数防下溢出

pw = [0] \* 10

for i in range(len(train\_labels)):

    pw[train\_labels[i]] += 1

for i in range(10):

pw[i] = np.log(pw[i]/len(train\_labels))

# 数据按类别归类到数组

num\_images = [[] for i in range(10)]

for i in range(len(train\_labels)):

num\_images[train\_labels[i]].append(train\_images[i])

# 计算每个类每个像素为白色的概率。num\_images\_p[i][k] = p'(x\_k|w\_i)【平滑过的p】

num\_images\_p = []

for i in range(10):             # 对每个类

    w = num\_images[i]

    white\_count = np.zeros(28\*28, dtype = np.float32)

    for k in range(len(w)):     # 二值化后求和即得1的个数

        white\_count += w[k]

    w = len(w)

    for k in range(28\*28):    # 频率

        # 之后要取对数。所以这里防止log0用拉普拉斯平滑

        white\_count[k] = (white\_count[k]+1)/(w+10)

num\_images\_p.append(white\_count)

# %%

# 对输入的二值化28\*28的图片用贝叶斯估计计算属于每个类的概率

def likely(img):

    likely = []                     # 属于数字i的“概率”

    for w in range(10):             # 对每类

        p = 0

        nump = num\_images\_p[w]

        for i in range(28\*28):

            p = p + np.log(nump[i] if img[i]==1 else (1-nump[i]))

        likely.append(pw[w] + p)

return likely

def test():

    right = 0

    error = []

    for i in range(len(test\_images)):

        res = np.argmax(likely(test\_images[i]))

        if res==test\_labels[i]:

            right = right + 1

        else:

            error.append(i)

    print("accuracy: ",right/len(test\_labels))

return error

errors = test() # 在测试集上统计正确率

**mindspore.ipynb【将各代码块粘贴而得以下代码。实际使用jupyter notebook】**

from mindvision.dataset import Mnist

# Download and process the MNIST dataset.

download\_train = Mnist(path="./mnist", split="train", batch\_size=32, repeat\_num=1, shuffle=True, resize=32, download=True)

download\_eval = Mnist(path="./mnist", split="test", batch\_size=32, resize=32, download=True)

dataset\_train = download\_train.run()

dataset\_eval = download\_eval.run()

# load model lenet

from mindvision.classification.models import lenet

network = lenet(num\_classes=10, pretrained=False)

import mindspore.nn as nn

from mindspore.train import Model

# Define the loss function.

net\_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')

# Define the optimizer function.

net\_opt = nn.Momentum(network.trainable\_params(), learning\_rate=0.01, momentum=0.9)

from mindspore.train.callback import ModelCheckpoint, CheckpointConfig

# Save model

config\_ck = CheckpointConfig(save\_checkpoint\_steps=1875, keep\_checkpoint\_max=10)

ckpoint = ModelCheckpoint(prefix="lenet", directory="./lenet", config=config\_ck)

from mindvision.engine.callback import LossMonitor

# Ini model parameter

model = Model(network, loss\_fn=net\_loss, optimizer=net\_opt, metrics={'accuracy'})

# Train the network model.

model.train(10, dataset\_train, callbacks=[ckpoint, LossMonitor(0.01, 1875)])

# Print accuracy

acc = model.eval(dataset\_eval)

print(f"{acc}")

# load model

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

# Load the saved model used for testing.

param\_dict = load\_checkpoint("./lenet/lenet-1\_1875.ckpt")

# Load parameters to the network.

load\_param\_into\_net(network, param\_dict)

# use model

import numpy as np

from mindspore import Tensor

import matplotlib.pyplot as plt

mnist = Mnist("./mnist", split="test", batch\_size=6, resize=32)

dataset\_infer = mnist.run()

ds\_test = dataset\_infer.create\_dict\_iterator()

data = next(ds\_test)

images = data["image"].asnumpy()

labels = data["label"].asnumpy()

plt.figure()

for i in range(1, 7):

    plt.subplot(2, 3, i)

    plt.imshow(images[i-1][0], interpolation="None", cmap="gray")

plt.show()

# Use the model.predict function to predict the classification of the image.

output = model.predict(Tensor(data['image']))

predicted = np.argmax(output.asnumpy(), axis=1)

# Output the predicted classification and the actual classification.

print(f'Predicted: "{predicted}", Actual: "{labels}"')

**实验二 隐马尔可夫模型**

1. **问题描述**
2. 利用隐马尔可夫模型进行中文语句的分词。数据集是人民日报1998年1月份的语料库，对600多万字节的中文文章加入了词性标注以及分词处理，由北京大学开发，是中文统计的常用资料 可以在语料库基础上构建词典、进行统计、机器学习等。
3. 基于 MindSpore平台提供的官方模型库，对相同的数据集进行训练，并与自己独立实现的算法对比结果（包括但不限于准确率、算法迭代收敛次数等指标），并分析结果中出现差异的可能原因，给出使用 MindSpore的心得和建议。
4. （加分项）使用 MindSpore平台提供的相似任务数据集（例如，其他的分类任务数据集）测试自己独立实现的算法并与 MindSpore平台上的官方实现算法进行对比，进一步分析差异及其成因。
5. **实现步骤与流程**

**Task1:（实现隐马尔科夫分词）**

1. 读取数据集

由train.txt可知，样本以行为单位，每行包含若干个词语，词语之间用空格分开。因此应该一行一行输出训练，一行一行进行分词。

1. 数据预处理

（第一轮实验后新增的步骤）用提供的训练集训练的模型结果不如人意。观察训练集得知，这个训练集的分词一塌糊涂。进一步发现，这份训练集根本不是北京大学开发的语料库。于是自行搜寻下载了正宗的北大语料库（存放于data/199801文件夹内）用以训练。原始数据还有出处、词性等标注，需要删除。此外，里面的数字字母都是全角字符，而一般使用的都是半角（测试集也都是半角数字），因此需要转换。

1. 编写隐马尔可夫类

**显状态集合**

本问题的显状态集合为字符集。

**隐状态集合**

按照中文词语形态，为每个字符定义了四种隐状态：B（词语开始）、M（中间字）、E（词语结尾）、S（单字成词）。例如，“我 爱 北京 天安门”对应于“S S BE BME”。

**参数估计方法**

隐马尔可夫模型学习问题分为有监督学习和无监督学习，其中课本上的Baum-Welch算法是无监督学习算法，适用于没有分词的训练集。而本实验已经有了分词，根据词语长度即可获取标签，属于有监督学习，可以使用最大似然估计法，即用频率代替概率。具体有以下参数的估计方法：

**初始分布**

是一个1\*4的向量。根据每句话第一个字所属隐状态，投票统计四个状态在第一个字的出现频率，获取初始分布。

**隐状态转移矩阵**

是一个4\*4的矩阵。根据标签序列，统计每种相邻状态出现的频数，进而得到转移概率。注意由频数到频率后，矩阵行和为1。

**发射概率矩阵**

理论上，统计每个隐状态下激发出某个字的频率，得到一个向量。四个状态得到四个向量，合并为一个矩阵。考虑到有些字不会属于某状态，比如标点不可能属于B，故此为稀疏矩阵。因此每个隐状态下使用键值对查找表的方式记录频率。

类似“朴素贝叶斯”实验，为了防止数据下溢出，以上所有概率都取了对数。为了防止log0，使用了拉普拉斯平滑。

**分词**

根据字符串（一个句子），使用Viterbi寻找最大概率（对数概率）路径，返回隐状态序列。据此再用简单的法则对原句进行分词。对于不在字典中的词，用对应集合中最小的概率的十倍代替，以减小未知对整体的影响。

1. 训练与使用

对训练集，先按行拆分为句子；对每句，按空格拆分为词语；对每个词语，根据词语长度赋予标签。进而统计频率得到参数估计。对测试集，先按行拆分为句子，逐句输入模型进行分词。

**Task2:（使用MindSpore）**

MindSpore上，分词仅为自然语言处理中最最基本的功能，而适合中文分词的模型，仅有“jieba”。MindSpore为jieba做了接口。为了达到较好的分词结果，使用jieba模型一般都从模型库下载预训练的模型与字典。加载模型，对测试集分词即可。

1. **实验结果与分析**

**Task1:（自己的HMM）**

由于使用了自己找的训练集，因此对使用两个训练集的结果进行比对分析。记我新找的训练集结果为new，记原本训练集的训练结果为raw。以下文本都在程序文件夹内，具体请看“README.md”文件的说明。

**在测试集上的对比：**

【测试集第6行】

new: “青年 的 命运 ，从来 都 同 时代 紧密 相连 。”

raw: “青年 的 命运 ，从 来 都 同时 代紧 密相 连 。”

【测试集第8行】

new: “一 批 又 一 批仁 人 志士 为 救国 救民 而 苦苦 追寻”

raw: “一批 又 一 批仁 人志 士为 救国 救民 而苦苦 追 寻”

【测试集第67行】

new: “要 紧扣 服务 青年 的 工作 生命 线 ，履行 巩固 和 扩大 党 执政 的 青年 群众 基础 这 一 政治责任 ，既把 青年 的 温度 如实 告诉 党 ，也把党 的 温暖 充分 传递 给 青年 。”

raw: “要紧 扣 服务 青年 的 工作 生命 线 ， 履行 巩固 和 扩大 党 执政 的 青年 群众 基础 这 一 政治 责任 ， 既 把 青年 的 温度 如 实告 诉 党 ， 也 把 党 的 温暖 充分 传递 给 青年 。”

【测试集第81行】

new: “要 培养 担当 实干 的 工作 作风 ，不尚 虚谈 、 多务 实功，勇于 到 艰苦 环境 和 基层 一线 去 担苦 、 担难 、 担重 、 担险，老老 实实 做 人 ，踏踏实实 干事 。”

raw: “要 培养 担当 实干 的 工作 作风 ，不 尚 虚谈 、多务 实功 ，勇于 到 艰苦 环境 和 基层 一线 去 担苦 、 担难 、 担重 、 担险 ，老 老 实实 做人 ，踏 踏 实实 干事 。”

观察分词结果发现，两个结果在很多地方分词结果都很正常，但是也有很大一部分结果奇怪。两个训练集结果各有千秋，难以判别出哪个更好。但是注意，new训练集的大小不到raw训练集的一半，却已经达到了类似的效果，已经能说明训练集对训练的重要性。

**在raw训练集上的对比**

我用new训练集训练的模型在raw训练集（清除了分词信息）上进行了分词，将两者（new对raw的分词结果，和raw）进行对比。

【raw第297951行】

New: “便 形成 了 迥然 不同 的 情感 色彩 。”

Raw: “便 形成 了 迥 然 不同 的 情感 色彩 。”

【raw第297893行】

New: “内容 和 形式 、 思想 和 艺术 尽可 能 完美 的 统一”

Raw: “内容 和 形式 、 思想 和 艺术 尽可能 完 美 的 统一”

【raw第156327行】

New:“某种 意义 上 讲 也 就 是 帮着 另 一方 提高 竞技 水平 ，创造 出好 的 成绩 。”

Raw:“某种 意义 上 讲 也 就是 帮 着 另 一方 提高 竞技 水平 ，创造 出 好 的 成绩 。”

同样难以评估。由于原训练集只有一句话中部分词语分词不好，且数据量较大，因此总体效果还是比模型分词结果好一些。但是绝大多数地方，模型结果也十分合理。

总之，分词结果并不令人满意。一方面和训练集的准确性、数据量有关，一方面基于最大似然估计的HMM模型过于简单而不够准确，不能增量训练而没有泛用性。由于“分词”本身就有主观性，各种分法都有道理，本实验缺乏用以评估模型分词效果的标准，无法进行进一步的分析。

**Task2:（MindSpore）**

如下是与上面相同的样本时，jieba模型的结果：

“青年 的 命运 ， 从来 都 同时代 紧密 相连 。”

“一批 又 一批 仁人志士 为 救国救民 而 苦苦 追寻 ， 一大批 先进青年 在 觉醒 年代 纷纷 觉醒 。”

“要 紧扣 服务 青年 的 工作 生命线 ， 履行 巩固 和 扩大 党执政 的 青年 群众 基础 这一 政治责任 ， 既 把 青年 的 温度 如实 告诉 党 ， 也 把 党 的 温暖 充分 传递 给 青年 。”

“要 培养 担当 实干 的 工作作风 ， 不尚 虚谈 、 多 务实 功 ， 勇于 到 艰苦环境 和 基层 一线 去 担苦 、 担 难 、 担 重 、 担 险 ， 老老实实 做人 ， 踏踏实实 干事 。”

结果比自行实现的好太多。观察“hmm\_model”文件，发现基本原理类同（都是BEMS四个状态）、数据形式类同（都是对数概率）。而jieba能达到这样的效果，一是“hmm\_model”文件大小是我的模型参数的3倍，训练样本多、范围广；二是它配有前缀词典“jieba.dict”，有额外的数据来源。

具体来说，jieba先基于前缀词典实现高效的词图扫描，生成句子中汉字所有可能成词情况所构成的有向无环图，再动态规划查找最大概率路径, 找出基于词频的最大切分组合。对于未登录词，采用了基于汉字成词能力的 HMM 模型，使用了 Viterbi 算法。两个算法相辅相成，一个保证了收录词语的精确度（因此成语的分词情况非常好），一个能处理未知的字词。而自行实现的HMM分词算法就像摸黑，全部当新词语处理，因此常常切断词语。

1. **MindSpore学习使用心得体会**

也许是分词过于基础，没有可供训练的模型。所以选择了直接用训练好的。

选择模型的时候，认识到可以使用mindspore\_hub加载现成的模型。但是由于对文档啃得并不透彻等因素，最后还是没有用起来。但是也学习到了用外部数据加载内置模型的方法。使用的jieba模型是早就耳闻并且使用了，mindspore内置了它的接口省去了不少功夫。从调用方式“dataset.map()”，可以看出分词的确是很基础的功能。

关于放弃mindspore\_hub，是因为版本问题。环境配置时，mindspore\_hub和mindspore的版本必须一致。mindspore\_hub的版本比mindspore滞后，因此制约了mindspore的版本。且hub最新版本（1.10.0）还没有模型可用，也没有相关文档。而使用老版本的api会报错。故放弃。

1. **代码附录**

**hmm.py(隐马尔可夫类)**

import pickle

import codecs   # 避免乱码问题

import math

'''

显状态：{字集合}

隐状态：{'B', 'M', 'E', 'S'}

数据结构： （全是log）

    A: 4\*4的矩阵 状态转移矩阵

    B: [

        {"字1": 由状态B激发字1的概率, ...},

        {...},

        ...

    ] 隐状态激发矩阵

    Pi: [P("B"), P("M"), P("E"), P("S")] 初始概率分布

'''

class HMM:

    # 静态属性和方法

    state\_map = {0: "B", 1: "M", 2: "E", 3: "S",

                'B': 0, "M": 1, "E": 2, "S": 3}

    def makeLabel(text):    # 为一个词语做标注

        l = len(text)

        if l == 0:

            return []

        elif l == 1:

            return [3]

        else:

            return [0] + [1] \* (l - 2) + [2]

    def fromLabel(text, label):  # label是字母不是序号 根据label进行分词

        new = []

        lastB = False

        for i in zip(text,label):

            if i[1] == 'B':

                if lastB:

                    new.append(' ')

                lastB = True

                new.append(i[0])

            else:

                new.append(i[0])

                lastB = False

                if i[1] == 'S' or i[1] == 'E':

                    new.append(' ')

        return ''.join(new)

    def \_\_init\_\_(self, fromFile=None):

        if fromFile != None:

            self.fromFile(fromFile)

        else:

            self.A = None   # 隐状态转移矩阵

            self.B = None   # 发射概率矩阵

            self.Pi = None  # 初始状态

    def fromFile(self, model\_path):     # 读取模型

        with open(model\_path, 'rb') as f:

            self.A = pickle.load(f)

            self.B = pickle.load(f)

            self.Pi = pickle.load(f)

    def toFile(self, model\_path):       # 保存模型

        with open(model\_path, 'wb') as f:

            pickle.dump(self.A, f)

            pickle.dump(self.B, f)

            pickle.dump(self.Pi, f)

    def train(self, train\_path):    # 训练模型 按似然估计方法，不能增量学习，即每次从头开始

        # 观察train.txt得知，一行是一个句子，即“句子”是输入量。一个句子被分词，说明输出是分词结果

        with codecs.open(train\_path, 'r', 'utf-8') as f:

            lines = [l.strip() for l in f.readlines()]  # 删去首尾的干扰空格

        # 初始化参数【采用似然函数法，不能迭代增量学习，所以每次训练都是重新开始】

        self.B = [{}, {}, {}, {}]

        self.A = [[0 for j in self.B] for i in self.B]

        self.Pi = [0 for i in self.B]

        # 统计相关数据

        for line in lines:  # 对每句话分析（一句一行）

            if not line:

                continue

            # 制作标签。例：sentence\_state = [0,1,2,3]（即['B','M','E','S']）

            words = line.split()

            sentence\_state = []

            for w in words:

                sentence\_state.extend(HMM.makeLabel(w))

            words = line.replace(' ', '')

            # 统计Pi频数

            self.Pi[sentence\_state[0]] += 1

            # 统计B

            for i in zip(sentence\_state, words):

                try:

                    self.B[i[0]][i[1]] += 1

                except:

                    self.B[i[0]][i[1]] = 1

            # 统计A

            for i in range(0, len(sentence\_state)-1):

                self.A[sentence\_state[i]][sentence\_state[i+1]] += 1

        # 得到ABPi (取log)

        # 1. 得到Pi

        s = sum(self.Pi) + 0.002

        for i in range(4):

            self.Pi[i] = math.log((self.Pi[i]+0.001)/s)

        # 2. 得到A

        for i in self.A:

            s = sum(i) + 0.002

            for j in range(len(i)):

                i[j] = math.log((i[j]+0.001)/s)

        # 3. 得到B

        for i in self.B:

            if not i:   # 如果是小样本可能会没有某状态

                continue

            s = math.log(sum(i.values()))

            for k in i:     # 不会有0

                i[k] = math.log(i[k]) - s

    def predict(self, text):    # Viterbi 根据字的序列, 求出隐状态

        if not (isinstance(text, str) and len(text) > 0):

            return None

        minValue = [10 \* min(i.values()) for i in self.B]

        # 初始化: 由每个初始分布得到第一个字的log概率

        delta = [self.Pi[i] + self.B[i].get(text[0], minValue[i]) for i in range(4)]

        path = [[], [], [], []]    # 路径记录

        delta2 = delta[:]

        # 第二个字：对每个第二个字的状态，计算第一个字的每一个状态到其的概率，保留最大值，记录路径

        for w in range(1, len(text)):

            for i in range(4):  # 对当前的每个状态

                maxpath = 0

                maxP = -math.inf

                for j in range(4):  # 对上一步的每个状态

                    temp = delta[j] + self.A[j][i] + self.B[i].get(text[w], minValue[i])

                    if temp > maxP:  # 保留最短路径(最大log概率)

                        maxP = temp

                        maxpath = j

                path[i].append(maxpath)

                delta2[i] = maxP     # 由哪个状态转移而来

            delta = delta2[:]

        # 找最终的最大值

        maxi = 0

        maxP = -math.inf

        for i in range(4):

            if delta[i] > maxP:

                maxi = i

                maxP = delta[i]

        # 回溯路径

        result = [HMM.state\_map[maxi]]

        for i in range(len(text)-2, -1, -1):

            maxi = path[maxi][i]

            result.append(HMM.state\_map[maxi])

        result.reverse()

        return result

**makedata.py(处理自己找的数据集)**

# 将199801语料库转换为只有分词的结果

import codecs

with codecs.open('../data/199801/199801.txt', 'r', encoding='gb2312', errors='ignore') as f:

    lines = f.readlines()

import unicodedata

def to\_halfwidth(text): # 数字字母转半角

    result = ""

    for char in text:

        if unicodedata.east\_asian\_width(char) in ('F', 'W'):

            code = ord(char)

            if 65281 <= code <= 65374:

                halfwidth\_char = chr(code - 65248)

                result += halfwidth\_char

                continue

        result += char

    return result

import re

pattern = r"/[a-zA-Z]+|\][a-zA-z]+|\["

with open('../data/199801\_train.txt', 'w', encoding='utf-8') as f:

    for l in lines:

        f.write(re.sub(pattern, "", to\_halfwidth(l[23:])))

**use.py(训练、保存、使用模型)**

from hmm import HMM

m = HMM()

train\_path = "../data/199801\_train.txt"

test\_path = "../data/test.txt"

# %%

# 训练并保存

print("training now...")

m.train(train\_path)

m.toFile('./hmm\_model.pkl')

print("the model has been saved to hmm\_model.pkl")

# %%

# 使用模型分词

# m = HMM(fromFile='./hmm\_model.pkl')

import codecs

with codecs.open(test\_path, 'r', 'utf-8') as f:

    lines = [l.strip() for l in f.readlines()]  # 删去首尾的干扰空格

print("predicting now...")

with codecs.open("hmm\_result.txt", 'w', 'utf-8') as f:

    for i in lines:

        x = m.predict(i)

        f.write(HMM.fromLabel(i, x)+'\n')

print("The result has been saved to hmm\_result.txt")

**mindspore.py**

import mindspore.dataset as ds

import mindspore.dataset.text as text

with open('../data/test.txt','r',encoding='utf-8') as f:

    input\_list = f.readlines()

dataset = ds.NumpySlicesDataset(input\_list, column\_names=["text"], shuffle=False)

# load model(pre trained)

HMM\_FILE = "./hmm\_model.utf8"

MP\_FILE = "./jieba.dict.utf8"

jieba\_op = text.JiebaTokenizer(HMM\_FILE, MP\_FILE)

dataset = dataset.map(operations=jieba\_op, input\_columns=["text"], num\_parallel\_workers=1)

with open('MindSpore\_fenci\_result.txt','w',encoding='utf-8') as f:

    for i in dataset.create\_dict\_iterator(num\_epochs=1, output\_numpy=True):

        f.write(' '.join(i['text']))

**实验三 神经网络**

1. **问题描述**
2. 基于神经网络模型及BP算法，根据训练集中的数据对所设计的神经网络模型进行训练，随后对给定的打乱的测试集中的数据进行分类。
3. 基于MindSpore平台提供的官方模型库，对相同的数据集进行训练，并与自己独立实现的算法对比结果（包括但不限于准确率、算法迭代收敛次数等指标），并分析结果中出现差异的可能原因。
4. （加分项）使用MindSpore平台提供的相似任务数据集（例如，其他的分类任务数据集）测试自己独立实现的算法，并与MindSpore平台上的官方实现算法进行对比，进一步分析差异及其成因。
5. **实验要求实现步骤与流程**

**Task1:**

1. 读取数据集

同“朴素贝叶斯分类器”。不再赘述。

1. 数据预处理

训练标签：模仿二分类方法，将类别转换为独热码。

训练数据：为了神经网络的设计方便，网络默认输入为增广向量，所以要将28\*28的输入向量加上偏置1拓展为(28\*28+1)维向量。

1. 编写神经网络类

设计之初担心仅用全连接层的分类效果不好（虽然最后发现已经够了），可能需要引入课本之外的诸如卷积池化等方法，故设计的结构较为开放，以便后序拓展新的结构。类的使用方法尽量向pytorch靠拢，将层结构、优化器、网络组织分离，规定了接口。不过也因此，运行效率会有所降低。

* 层结构：实现了全连接层Dense，实现输入的加权和激活。

正向传播需要考虑到层结构的连接，于是用本层的输入x和上一层的输出z共用指针。设计之初只考虑到单样本输入，共用ndarry的指针，改变值使用z[:] = ? 防止重新分配空间；而后来用到了批量输入，x和z的大小不定，故改为共用一个字典的指针，字典的value属性为传递值，实现了对单样本和批量样本的兼容。

反向传播的推导与书上的推导略有不同，如果单层输入为x，输出为z，这里只要关心在给出 的情况下，如何得到 ，以及如何得到 传递给上一层（上一层的即为本层的）。公式如下：

其中 传递给优化器更新权值。这里的x和z可以是单个数据（向量），也可以是批量数据（矩阵）。

* 优化器：实现了SGD和Adam，接收 ，更新所属层的权值。

后者是在使用前者感到训练时间长时，搜索资料得到的算法。相比于SGD，它能够对每个不同的参数调整不同的学习率，对频繁变化的参数以更小的步长进行更新，而稀疏的参数以更大的步长进行更新,但又不至于分母收缩到过小。

* 网络组织：主要用于连接各层，组织正向传播和反向传播。
* 激活函数和初始化方法：以函数的形式体现。

考虑到运算速度，我加入了relu函数，并在节点较多的层使用。在使用中曾遇到数据溢出的问题，是因为初始值设置太大。于是搜索资料，加入了两种初始化方法：Xavier用于一般的初始化，Kaiming用于relu的初始化。

为了使用网络，我设计了用JSON格式导出或读取每个类的必要参数，参数包括函数（用base64库将dill转换的函数变为字符串保存）。

1. 训练网络

经过多次尝试（见实验结果与分析），选定了现在使用的网络结构：

m = nn.nn([

    nn.Dense(28\*28+1,201,nn.Adam(0.0003),nn.relu,nn.drelu,nn.Kaiming),

    nn.Dense(201,10,nn.Adam(0.0003),nn.sigmoid,nn.dsigmoid,nn.Xavier)

])

即输入为(28\*28+1)维，加权后经过relu激活函数到201维的隐藏层，加权后经过sigmoid激活函数输出10维，分别使用Kaiming和Xavier方法初始化权值，都使用学习率为0.0003的Adam优化器进行更新。

经测试，使用batch\_size=10的批训练可以兼顾速度和准确率。每一轮将所有样本打乱，10个一组输入网络，更新权值。轮末将测试集样本一个个输入以统计正确率。保存网络参数后开始下一轮批训练。

**Task2:**

思路同上，由于和朴素贝叶斯共用数据集，任务要求相同，故不在赘述详细流程。但是本次我使用了另一个lenet的变体：LeNet5。MindSpore中的LeNet5和lenet都是基于LeNet模型的变体，它们的主要区别在于网络结构和参数数量的不同。

LeNet5是最早提出的卷积神经网络之一，由Yann LeCun等人于1998年提出。它的结构包括两个卷积层、两个池化层和三个全连接层。LeNet5中的卷积层和池化层使用的是sigmoid激活函数，输出层使用的是softmax激活函数。

lenet是在LeNet5的基础上做了一些改进，主要用于CIFAR-10数据集的分类任务。lenet与LeNet5的主要区别在于卷积核的大小和数量、池化方式、激活函数的选择等。具体来说，lenet中的卷积核大小为5x5，比LeNet5中的小；卷积核的数量也有所改变；池化方式从平均池化变为最大池化；激活函数使用的是ReLU，可以更好地避免梯度消失的问题。

此外，MindSpore中的LeNet5和lenet还有一些实现细节上的差异，比如输入图片的大小、批次大小、优化算法等。

1. **实验结果与分析**

**Task1:（自己的网络）**

按时间顺序，我进行了4个网络的训练，最终达到了98%的正确率

**Model\_1:** 【**最终模型存放于model1文件夹**】

此时并未兼容批训练，也没有用特别的初始化方法。

m = nn.nn([

    nn.Dense(28\*28+1, 500, nn.SGD(0.001), nn.relu, nn.drelu),

    nn.Dense(500, 10, nn.SGD(0.001), nn.relu, nn.drelu)

])

|  |  |  |
| --- | --- | --- |
| 学习率 | 轮数 | 正确率 |
| 0.001 | 1 | 0.3372 |
| 2 | 0.5570 |
| 3 | 0.6610 |
| 4 | 0.7185 |
| 0.0005 | 5 | 0.7621 |
| 6 | 0.8407 |
| 7 | 0.8418 |
| 8 | 0.8530 |
| 9 | 0.8637 |

采用了手动调节学习率的方法。到后来准确率的提升速度明显降低，学习较慢，效果不好。

**Model\_2:【最终模型参数存放于model2文件夹】**

不满足于第一个模型的准确率，试图加大网络的复杂度。

m = nn.nn([

    nn.Dense(28\*28+1, 300, nn.SGD(0.0002), nn.relu, nn.drelu),

    nn.Dense(300, 200, nn.SGD(0.0002), nn.relu, nn.drelu),

    nn.Dense(200, 100, nn.SGD(0.0002), nn.relu, nn.drelu),

    nn.Dense(100, 10, nn.SGD(0.0002), nn.relu, nn.drelu)

])

|  |  |  |
| --- | --- | --- |
| 学习率 | 轮数 | 正确率 |
| 0.0002 | 1 | 0.8507 |
| 2 | 0.8981 |
| 3 | 0.9144 |
| 4 | 0.9267 |
| 5 | 0.9328 |
| 0.0001 | 6 | 0.9367 |
| 7 | 0.9384 |
| 8 | 0.9412 |
| 0.00012 | 9 | 0.9433 |
| 10 | 0.9444 |
| 11 | 0.9454 |
| 0.00005 | 12 | 0.9468 |
| 13 | 0.9473 |
| 14 | 0.9479 |

事实证明网络大了效果确实好了，但是很快，正确率的上升就缓慢起来。但是，深层网络的训练速度慢了。

训练之初遇到了问题：矩阵相乘结果溢出。此时使用的是[-1,1]随机取值初始化，虽然缩小了取值范围解决了问题，但是没有普适性。本次模型训练后给Dense类加入了初始化方法。此外， “手动改变学习率”很奇怪，由于缺乏知识，我基本上是瞎调，不知道最后是因为学习率小了还是收敛了。所以本次模型训练后加入了Adam方法。

**Model\_3:【训练过程存放于model3/ann.html】**

使用了Adam优化器，使用了Kaiming初始化方法。但依旧是单样本输入。

m = nn.nn([

    nn.Dense(28\*28+1,200,nn.Adam(0.0003),nn.relu,nn.drelu, nn.Kaiming),

    nn.Dense(200,10,nn.Adam(0.0003),nn.relu,nn.drelu,nn.Kaiming)

])

|  |  |  |  |
| --- | --- | --- | --- |
| 批次 | 正确率 | 批次 | 正确率 |
| 1 | 0.9678 | 6 | 0.9786 |
| 2 | 0.9734 | 7 | 0.9786 |
| 3 | 0.9758 | 8 | 0.9776 |
| 4 | 0.9775 | 9 | 0.9767 |
| 5 | 0.978 | 10 | 0.9769 |

一开始的正确率就很高，第二轮就已经突破97%，证明了Adam方法的有效性。值得注意的是，第7轮开始正确率下降，猜测发生了过拟合。此外，单个输入的训练方式时间太长，用了89分钟。

**Model\_4:【结果存放于model4文件夹，是最终的模型】**

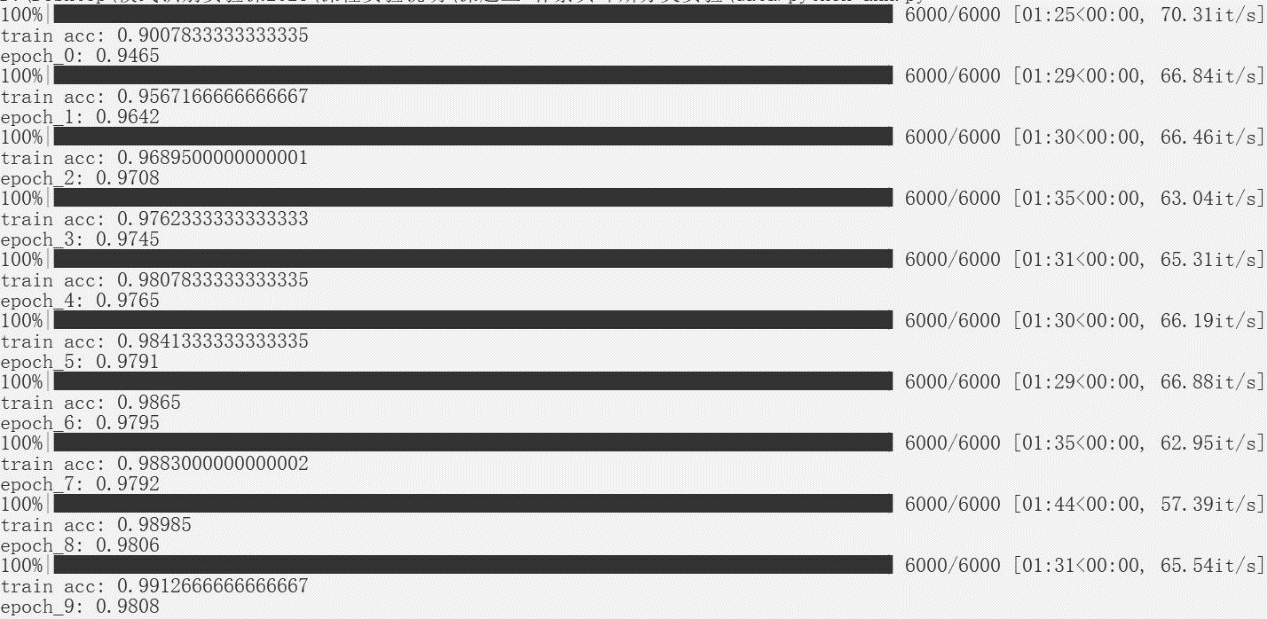
相比于model\_3，本模型适应了批训练。对层参数做出了微调。选用batch\_size=10的训练方法进行训练。训练过程的图片如下，记录了训练时间、在训练集上的正确率（train\_acc后面的数字）、在验证集上的正确率（epoch\_i后面的数字）。

m = nn.nn([

    nn.Dense(28\*28+1,201,nn.Adam(0.0003),nn.relu,nn.drelu,nn.Kaiming),

    nn.Dense(201,10,nn.Adam(0.0003),nn.sigmoid,nn.dsigmoid,nn.Xavier)

])

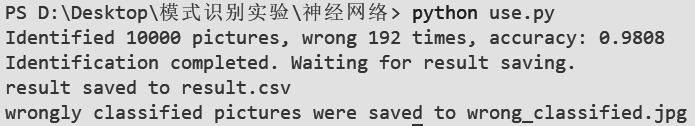


虽然第一次的训练结果不如model3，但是训练速度快（平均01:32s一轮）、上限高，仅9轮正确率就突破了98%。第九和第十轮的正确率仅增加了0.0002，即将过拟合，故不再继续训练。事实证明，批训练能显著降低训练时间、防止过拟合。

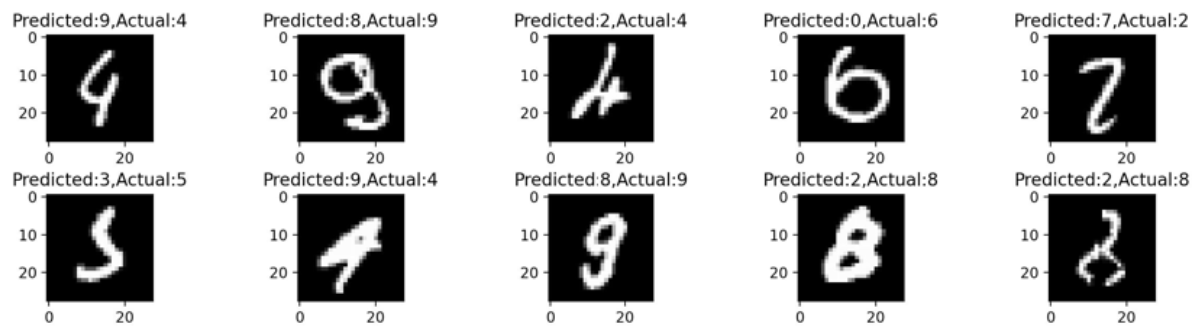
Model4前还有3次测试没有记录，分别选用了batch\_size为200、100、64的批训练，观察前两次的训练结果，发现200的效果类似100，而64的效果优于前者，而现在使用的10的效果又优于64。

正确率达98%，已经不需要再加入卷积层了。但因为我的神经网络的模块化设计的缘故，指针操作和参数传递比一体化的神经网络更多，严重影响了运行速度（发现往届学长学姐类似规模、层数写死的网络有训练十轮仅需3分钟的）。但是这样的设计让我得以尝试更多结构，以找到较优的网络。这次编写的轻量神经网络也便于今后的使用和拓展。

**使用网络：运行use.py**



**可视化查看分错的结果(报告中仅选取10个展示)：**

****

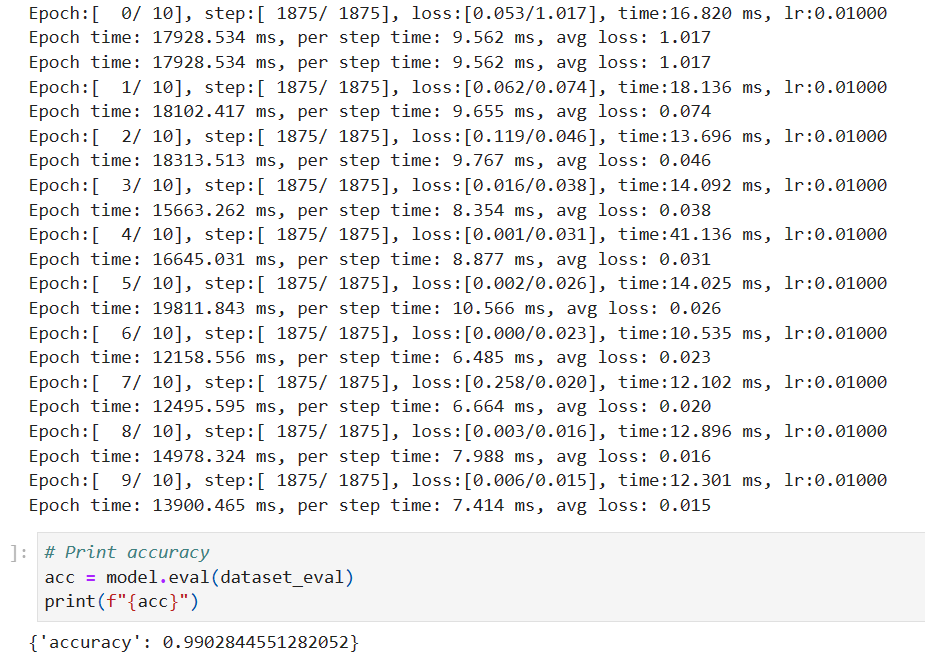
可以发现，错分的一部分连我也难以识别。总览所有识别错误的图，数字“8”的错误率格外高。很多特征明显的也没有正确分类。说明仍有很大改进空间。此外，我用自己的图（朴素贝叶斯实验中的那个）让网络识别，10个错了4个。说明也缺乏普适性，但比起朴素贝叶斯分类器，已经是巨大的进步。

**Task2:**

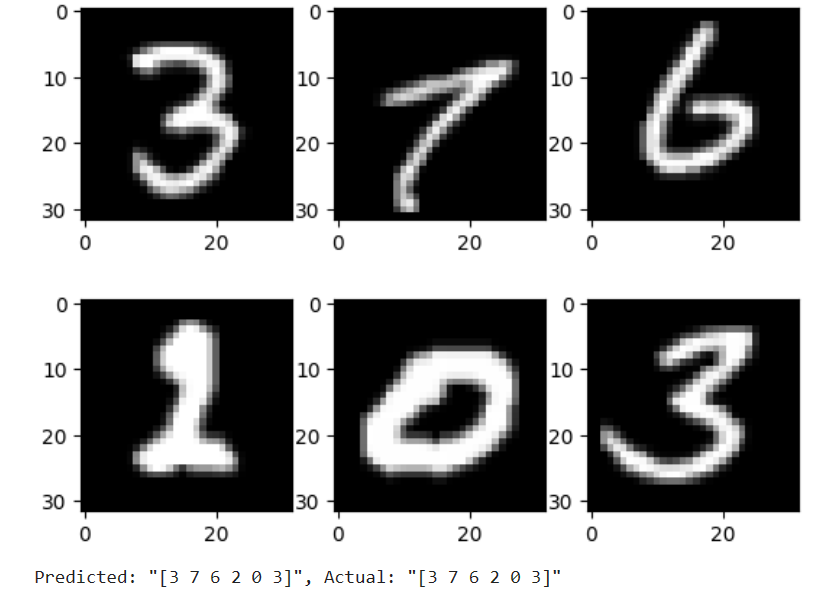
经过10轮训练后，LeNet5模型正确率高达99%，甚至比改进版的lenet还要好。当然这和初始化也有关系，但总的来说达到了同样的精度。与我的相比，同样是神经网络，同样是10轮，从结果来看，我的模型仅比MindSpore官方模型正确率低0.009，十分接近，都已经到了很高的精度。而我的模型，仅有一个隐藏层，相比lenet复杂度降低了很多，能达到这个精度，已经很不错。如果要进一步提升正确率，可以按原想法，加入卷积、池化这种专门针对图片处理的神经网络层。不过由“奥卡姆剃刀原理”，我认为现在的网络已经足矣。

但是，用时长还是我模型的巨大缺点。MindSpore版本仅用了2分40秒，远远比我的少。但如果我用C写或者仅仅把模型写死（不考虑拓展性，发挥numpy的速度），我觉得我的模型也能达到同样的速度。

附图1：mindspore每轮训练结果：



附图2：抽样检测的结果，全部正确。



1. **MindSpore学习使用心得体会**

大体同“朴素贝叶斯”，代码几乎不变。学会了找新模型并使用。了解了模型的通用使用方法。神经网络很占内存，服务器上跑不动。

1. **代码附录**

**nn.py【神经网络类】**

import numpy as np

import base64

import dill     # 保存函数用

# 保存函数到文件

def function\_str(f):

    return base64.b64encode(dill.dumps(f)).decode('utf-8')

def str\_function(functionString):

    return dill.loads(base64.b64decode(functionString))

# 转独热码 如数字8(第7类)对应00000001000

def one\_hot(Y, n\_classes):

    return np.eye(n\_classes)[Y].astype(float)

'''

激活函数activation function

'''

# 激活函数relu

def relu(A, l=0.05):

    return np.maximum(A, A \* l)

# relu的导数

def drelu(A, l=0.05):

    return np.where(A > 0, 1., l)

# 激活函数sigmoid

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

# sigmoid的导数

def dsigmoid(x):

    temp = np.exp(-x)

    return temp / (1 + temp)\*\*2

'''

误差函数loss function

'''

# 误差函数 x: 输入 t: 目标

def L2(x, t):

    return 0.5\*((x-t)\*\*2)

def dL2(x, t):

    return x-t

'''

权值初始化

'''

def Xavier(size):

    bound = np.sqrt(6/(size[0]+size[1]))

    return np.random.uniform(-bound, bound, size)

def Kaiming(size):  # 查资料得知对relu层应使用此方法

    return np.random.normal(0, np.sqrt(2/size[0]), size)

'''

优化器

必须实现：

- getParams(params)【由层向优化器传递优化权值】

- step(d)【由层向优化器传递梯度】

- toJson()【返回参数json】

- fromJson(js)【由json构造优化器】

'''

class SGD:

    def \_\_init\_\_(self, lr=0.001, settingJson=None) -> None:

        if settingJson != None:

            self.fromJson(settingJson)

        else:

            self.lr = lr

            self.params = None

    def getParams(self, params):

        self.params = params

    def step(self, grad):

        self.params -= self.lr\*grad

    def toJson(self):

        return {

            "type": "SGD",

            "lr": self.lr

        }

    def fromJson(self, js):

        self.lr = js["lr"]

        self.params = None

class Adam:

    def \_\_init\_\_(self, lr=3e-4, beta1=0.9, beta2=0.999, settingJson=None) -> None:

        if settingJson != None:

            self.fromJson(settingJson)

        else:

            self.lr = lr

            self.beta1 = beta1

            self.beta2 = beta2

            self.t = 0.

            # 以下参数由 getParams给出具体的值

            self.params = None

            self.m = None

            self.v = None

    def getParams(self, params):    # 由所属层的构造函数调用

        self.params = params

        self.m = np.stack([np.zeros\_like(x, dtype=float) for x in params])

        self.v = np.stack([np.zeros\_like(x, dtype=float) for x in params])

    # 输入：dJ/dw 对 self.params 进行优化

    def step(self, grad):

        self.t += 1

        self.m[:] = self.m \* self.beta1 + grad \* (1 - self.beta1)

        self.v[:] = self.v \* self.beta2 + (grad \*\* 2) \* (1 - self.beta2)

        mh = self.m / (1 - self.beta1 \*\* self.t)

        vh = self.v / (1 - self.beta2 \*\* self.t)

        self.params -= self.lr \* mh / (np.sqrt(vh) + 1e-10)

    def toJson(self):   # 导出为json格式

        return {

            "type": "Adam",

            "lr": self.lr,

            "t": self.t,

            "m": self.m.tolist(),

            "v": self.v.tolist(),

            "beta": [self.beta1, self.beta2]

        }

    def fromJson(self, js):   # 导入json构造

        self.lr = js["lr"]

        self.beta1, self.beta2 = js["beta"]

        self.t = js["t"]

        self.m = np.array(js["m"])

        self.v = np.array(js["v"])

        self.params = None

'''

层

必须实现：

属性：

- inputNum 【输入维度】

- outputNum 【输出维度】

- x 【输入 指针，由网络类分配连接】

- z 【输出，指针不可变】

- w 【存储参数】

函数：

- fp(x)【正向传播】

- bp(t)【反向传播，调用优化器实现更新】

- toJson()【导出层参数为json，嵌套优化器json】

- fromJson(js)【从json建立本层】

'''

class Dense:

    def \_\_init\_\_(self, input=10, output=10, optimizer=None, f=relu, df=drelu, ini = Xavier, settingJson=None) -> None:

        if settingJson != None:     # 来自文件

            self.fromJson(settingJson)

        else:

            self.inputNum = input

            self.outputNum = output

            self.f = f

            self.df = df

            self.x = None       # 输入。和上一级的输出为同一个数组。由nn类连接。{"value": ndarray}

            self.y = None       # 净激活能 np.array 值在正向传播时确定

            self.z = {"value": None}   # 输出 和下一层共用, 故地址不可变 为了兼容批量训练, 改为此形式

            # self.w = np.random.uniform(-0.1, 0.1, (output, input))

            self.w = ini((output, input))

            optimizer.getParams(self.w)

            self.optim = optimizer

    def fp(self):   # 正向传播 要求x["value"]已经有np.array 支持批量输入

        self.y = self.x["value"] @ self.w.T

        self.z["value"] = np.array([self.f(k) for k in self.y])

    # 输入：dJ/dz; 输出：dJ/dx

    # 传递 dJ/dw 给 self.optimizer 完成优化

    def bp(self, d):    # 反向传播 d为dJ/dz, 由下一级给出 支持批量输入

        djdy = d\*[self.df(i) for i in self.y]

        djdx = djdy@self.w

        if djdy.ndim == 2:

            djdw = np.mean([np.outer(djdy[i], self.x["value"][i]) for i in range(len(djdy))], axis=0)

        else:

            djdw = np.outer(djdy, self.x["value"])

        self.optim.step(djdw)

        return djdx

    def toJson(self):   # 将参数导出为json格式

        return {

            "type": "Dense",

            "input": self.inputNum,

            "output": self.outputNum,

            "param": self.w.tolist(),

            "f": function\_str(self.f),

            "df": function\_str(self.df),

            "optim": self.optim.toJson()

        }

    def fromJson(self, js):   # 从json得到权值

        self.w = np.array(js["param"])

        self.f = str\_function(js["f"])

        self.df = str\_function(js["df"])

        self.inputNum = js["input"]

        self.outputNum = js["output"]

        self.x = None

        self.y = None

        self.z = {"value": None}

        # 注意，global为本文件的命名空间。所以以后拓展的优化器得写本文件中

        self.optim = globals()[js["optim"]["type"]](settingJson=js["optim"])

        self.optim.getParams(self.w)

'''

网络

'''

class nn:

    '''

    传参示例：layers = [

        Dense(28\*28, 200, Adam(), relu, drelu, Kaiming),

        Dense(200, 200, Adam(), sigmoid, dsigmoid),

        Dense(200, 10, Adam(), relu, drelu, Kaiming)

    ]

    '''

    def \_\_init\_\_(self, layers=None, loss=L2, dloss=dL2, settingJson=None) -> None:

        if settingJson != None:     # 来自文件

            self.fromJson(settingJson)

        else:

            self.loss = loss

            self.dloss = dloss

            self.layers = layers

            self.input = {"value": None}  # 输入层 地址不可变

            # 连接所有层

            temp = self.input

            outNum = layers[0].inputNum

            for i in layers:

                if i.inputNum == outNum:

                    i.x = temp

                    temp = i.z

                    outNum = i.outputNum

                else:

                    raise Exception("connection demension error!")

            self.output = temp

    def fp(self, x):

        self.input["value"] = x

        for i in self.layers:

            i.fp()

        return self.output["value"]

    def bp(self, t):

        d = self.dloss(self.output["value"], t)

        for i in range(len(self.layers)-1, -1, -1):

            d = self.layers[i].bp(d)

    def toJson(self):

        return {

            "param": [l.toJson() for l in self.layers],

            "f": function\_str(self.loss),

            "df": function\_str(self.dloss),

        }

    def fromJson(self, js):

        self.loss = str\_function(js["f"])

        self.dloss = str\_function(js["df"])

        self.layers = [globals()[l["type"]](settingJson=l)

                       for l in js["param"]]

        self.input = {"value": None}

        # 连接所有层

        temp = self.input

        outNum = self.layers[0].inputNum

        for i in self.layers:

            if i.inputNum == outNum:

                i.x = temp

                temp = i.z

                outNum = i.outputNum

            else:

                raise Exception("connection demension error!")

        self.output = temp

if \_\_name\_\_ == "\_\_main\_\_":      # 测试用

    import json

    # 创建神经网络

    m = nn([

        Dense(2, 2, SGD(), sigmoid, dsigmoid),

        Dense(2, 1, SGD(), sigmoid, dsigmoid)

    ])

    # 保存到文件

    with open('nntest.json', 'w', encoding="utf-8") as f:

        json.dump(m.toJson(), f)

    # 从文件读取网络

    with open("nntest.json", "r", encoding="utf-8") as f:

        nnn = json.load(f)

    m = nn(settingJson=nnn)

    # 使用网络

    print("out: ", m.fp(np.array([2, 4])))

    m.bp(np.array([1]))

    print(f"after bp: {m.fp()}")

    # 保存更改

    with open('nntest2.json', 'w', encoding="utf-8") as f:

        json.dump(m.toJson(), f)

**ann.py【训练网络】**

import gzip

import numpy as np

import json

import nn

from tqdm import tqdm

def load\_data():    # 读取到npArray, 每维是一个28\*28, 0~255取值的数组。

    with gzip.open('../data/train-images-idx3-ubyte.gz', 'rb') as f:

        train\_images = np.frombuffer(

            f.read(), np.uint8, offset=16).reshape(-1, 28\*28)

    with gzip.open('../data/train-labels-idx1-ubyte.gz', 'rb') as f:

        train\_labels = np.frombuffer(f.read(), np.uint8, offset=8)

    with gzip.open('../data/t10k-images-idx3-ubyte.gz', 'rb') as f:

        test\_images = np.frombuffer(

            f.read(), np.uint8, offset=16).reshape(-1, 28\*28)

    with gzip.open('../data/t10k-labels-idx1-ubyte.gz', 'rb') as f:

        test\_labels = np.frombuffer(f.read(), np.uint8, offset=8)

    return train\_images, train\_labels, test\_images, test\_labels

train\_images, train\_labels, test\_images, test\_labels = load\_data()

# 数据加入偏置，标签改为独热码

train\_labels = np.stack([nn.one\_hot(i, 10) for i in train\_labels])

test\_images = np.stack(

    [np.insert(i.astype('float64')/255, 0, 1) for i in test\_images])

train\_images = np.stack(

    [np.insert(i.astype('float64')/255, 0, 1) for i in train\_images])

# %%

m = nn.nn([

    nn.Dense(28\*28+1, 201, nn.Adam(0.0003), nn.relu, nn.drelu, nn.Kaiming),

    nn.Dense(201, 10, nn.Adam(0.0003), nn.sigmoid, nn.dsigmoid)

])

# 乱序分批

def iterate(X, Y, batch\_size=100):

    idx = np.arange(len(X))

    np.random.shuffle(idx)

    return [([X[idx[i:i+batch\_size]], Y[idx[i:i+batch\_size]]]) for i in range(0, len(X), batch\_size)]

# 批训练

def batch\_train():

    bar = iterate(train\_images, train\_labels, 10)

    accs = []

    for sample in tqdm(bar):

        X, Y = sample

        res = m.fp(X)

        m.bp(Y)

        # 计算一个batch中的正确率

        acc = np.mean(np.argmax(res, axis=-1) == np.argmax(Y, axis=-1))

        accs.append(acc)

    print(f"train acc: {np.mean(accs)}")

# 验证

def test():

    right = 0

    for i in range(len(test\_labels)):

        t = np.argmax(m.fp(test\_images[i]))

        if t == test\_labels[i]:

            right += 1

    return right/len(test\_labels)

# %%

for i in range(10):     # 训练

    batch\_train()

    print(f"epoch\_{i} test acc: {test()}")

    with open(f'./model/model\_{i}.json', 'w', encoding="utf-8") as f:

        json.dump(m.toJson(), f)

**use.py【使用训练好的神经网络】**

# 使用训练好的神经网络预测

# 加载数据

import json

import nn

import gzip

import numpy as np

def load\_data():

    with gzip.open('../data/t10k-images-idx3-ubyte.gz', 'rb') as f:

        test\_images = np.frombuffer(

            f.read(), np.uint8, offset=16).reshape(-1, 28\*28)

    with gzip.open('../data/t10k-labels-idx1-ubyte.gz', 'rb') as f:

        test\_labels = np.frombuffer(f.read(), np.uint8, offset=8)

    return test\_images, test\_labels

test\_images, test\_labels = load\_data()

# 加偏置

test\_images\_b = np.stack([np.insert(i.astype('float64')/255, 0, 1) for i in test\_images])

# %%

# 加载网络

with open("./model4/model\_10batch\_final.json", "r", encoding="utf-8") as f:

    nnn = json.load(f)

m = nn.nn(settingJson=nnn)

# %%

# 识别

result = []

wrong = []  # 错误的编号

for i in range(len(test\_labels)):

        t = np.argmax(m.fp(test\_images\_b[i]))

        result.append(t)

        if t != test\_labels[i]:

            wrong.append(i)

print(f"Identified {len(result)} pictures, wrong {len(wrong)} times, accuracy: {1-len(wrong)/len(result)}")

print("Identification completed. Waiting for result saving.")

# %%

# 保存结果

with open('result.csv','w',encoding="utf-8") as f:

    f.write("label,ann\_result\n")

    for i in range(len(test\_labels)):

         f.write(f"{test\_labels[i]},{result[i]}\n")

print("result saved to result.csv")

# %%

# 保存识别错误的图片

import matplotlib.pyplot as plt

fig, axs = plt.subplots(len(wrong)//5+1, 5, figsize=(15, 0.4\*len(wrong)))

for i in range(len(wrong)):

    axs[i//5, i%5].imshow(test\_images[wrong[i]].reshape(28,28), cmap='gray')

    title = f"Predicted:{result[wrong[i]]},Actual:{test\_labels[wrong[i]]}"

    axs[i//5, i%5].set\_title(title)

plt.subplots\_adjust(hspace=0.5)

fig.savefig("wrong\_classified.jpg", dpi=200)

print("wrongly classified pictures were saved to wrong\_classified.jpg")

**mindspore代码与朴素贝叶斯基本相同，只是模型改了，只给出改动部分：**

from mindvision.classification.models.backbones import LeNet5

network = LeNet5()

**心得体会**

非常感谢这次实验，我对课本知识有了特别深入的体会。在学习理论知识时，我一度很看不起贝叶斯分类器等不能迭代的方法，我想这样的模型效果必然不会好。然而，在朴素贝叶斯和隐马尔可夫得到了远超预期的效果后，我有些改观了。这就是经典的魅力呀。

对课上提到的技巧也有了深切的认识。此处特指防下溢出的对数概率，并且引出了数据平滑的操作。这是理论和实践的关系与区别。

最重要的，我对于神经网络的搭建与训练有了诸多心得。搭建以及各种尝试大概用了我5天的时间，是用时最长的实验，测试了很多参数和结构，收获颇丰。虽然之前也从头写过神经网络，但这是第一次编写这样一个系统的、考虑了升级的、模块解耦的网络架构，颇有成就感。对反向传播公式的推导更加得心应手。关于网络规模，虽然设计之初考虑的是大网络，最终实际使用的却仅有1个隐藏层。规模需要根据任务而定，复杂的网络不一定比小网络实用。关于训练方式，认识到了批训练的重要性。在学习本课程之前自己搭建的网络一直都是单输入的。这次实验，批训练带来的速度和效果让人震惊。

总之，不实验，真体会不到这些方法的必要性和好处。

最后是对numpy库的感叹。熟练使用确实可以大大加快程序构建速度和运行速度，而我对numpy的认识与使用还十分初级，看到别人的代码仿佛发现了新大陆，尝试着自己使用了部分。代码还有加速的空间。

有关mindspore：

通过查阅Mindspore文档，我学会了基本的网络搭建。学习不同的深度学习框架如Mindspore能够更好地帮助我对网络搭建的理解。华为平台的Mindspore在设计上有一些精妙的地方。并且在使用时候，感受这些完整框架对代码的优美封装，是对自己代码美化的一大动力——提醒自己：提高代码复用性、开放性，记得对代码进行封装、解释并且提供函数api。

Mindspore有着不同于pytorch的函数组织形式。Mindspore较多API不支持GPU，使用过程中要注意对于代码时间算法复杂度的控制。Mindspore的文档我还没有吃透，在环境配置方面花费了很大的功夫。我认为官方文档缺少提纲挈领的部分，入门时非常痛苦。

此外，课程提供了云算力以运行代码，体验新奇。华为的平台让我深切体会到了神经网络对性能的要求。首先是价格。配置有GPU的已经不是课程代金券允许我使用的算力了，于是选择了4G计算加速平台，可见计算资源的重要性。其次是资源占用。一训练神经网络，内存就爆满，进程就被kill，是本地没有的新奇体验。