華中農業大學
HUAZHONG AGRICULTURAL UNIVERSITY

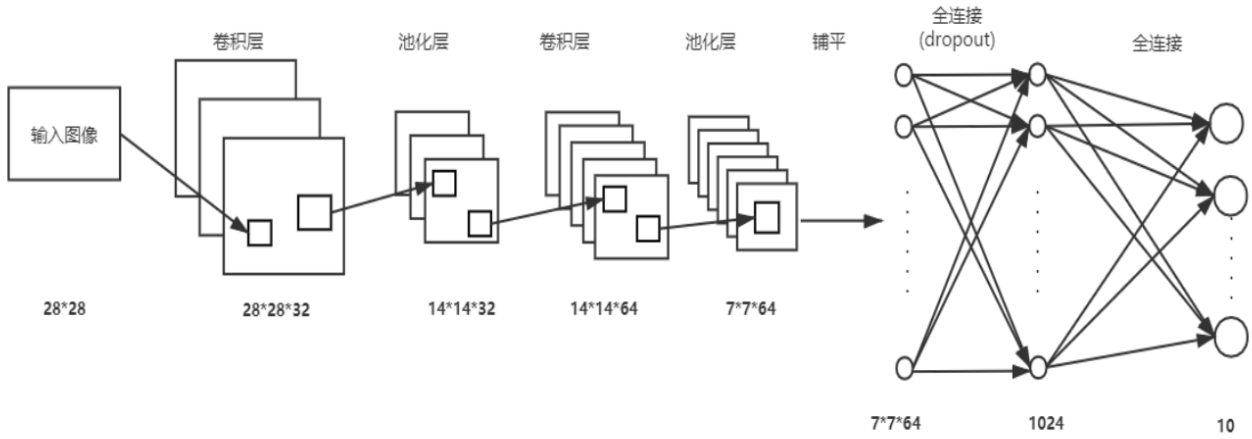
神经网络实验报告

实验二-Pytorch 实现 CNN

院系	信息学院
专业班级	大数据 2001
姓名	李俊松
学号	2020301221205
指导老师	胡学海

实验要求

用 $python$ 的 $Pytorch$ 模块实现卷积神经网络。网络结构为一个输入层、两个卷积层、一个全连接层、一个输出层



实验数据

$mnist$ 手写体数字识别数据集。

通过$pytorch$内置的包直接载入

```
trainData = torchvision.datasets.MNIST(
    path, train=True, transform=transform, download=True
)

testData = torchvision.datasets.MNIST(path, train=False, transform=transform)
```

实验流程

$Pytorch$环境配置

1. 终端中输入 $nvidia-smi$, 查看 $CUDA$ 版本
2. 进入 $Pytorch$ 官网 https://pytorch.org/get-started/locally/找到对应 $CUDA$ 版本的 $Pytorch$（ps: 一般来说 $Pytorch$ 官网会直接提供 $conda$ 下载指令）
3. 终端中使用 $conda$ 创建新环境并下载 $Pytorch$
4. 进入 $Python$ 检查是否安装成功

```
import torch
#成功import 则安装成功
torch.cuda.is_available()
#输出True，说明服务器GPU可用
```

5. 本实验为了更好可视化训练和测试结果，需使用 $pip\ install\ tqdm$ 安装 $tqdm$ 库

网络结构

使用2个卷积+池化板块,在$flatten$之后做一个MLP,其中套用$drop\ out$做正则化,使用$RELU$激活函数,最后经过$Softmax$运算后得到10分类的概率

参数如下

```
(model): Sequential(
  (0): Conv2d(1, 32, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (3): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (4): ReLU()
  (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (6): Flatten(start_dim=1, end_dim=-1)
  (7): Linear(in_features=3136, out_features=1024, bias=True)
  (8): ReLU()
  (9): Dropout(p=0.30000000000000004, inplace=False)
  (10): Linear(in_features=1024, out_features=10, bias=True)
  (11): Softmax(dim=1)
)
```

$tqdm$

$tqdm$ 是一个快速，可扩展的Python进度条，可以在 Python 长循环中添加一个进度提示信息，用户只需要封装任意的迭代器 $tqdm(iterator)$。

安装很简单,只需要执行以下命令即可

```
pip install tqdm
```

tqdm参数说明

```
class tqdm(object):
  """
  Decorate an iterable object, returning an iterator which acts exactly
  like the original iterable, but prints a dynamically updating
  progressbar every time a value is requested.
  """

    def __init__(self, iterable=None, desc=None, total=None, leave=False,
        file=sys.stderr, ncols=None, mininterval=0.1,
        maxinterval=10.0, miniters=None, ascii=None,
        disable=False, unit='it', unit_scale=False,
        dynamic_ncols=False, smoothing=0.3, nested=False,
        bar_format=None, initial=0, gui=False):
```

- iterable: 可迭代的对象, 在手动更新时不需要进行设置
- desc: 字符串, 左边进度条描述文字
- total: 总的项目数

- leave: bool值, 迭代完成后是否保留进度条
- file: 输出指向位置, 默认是终端, 一般不需要设置
- ncols: 调整进度条宽度, 默认是根据环境自动调节长度, 如果设置为0, 就没有进度条, 只有输出的信息
- unit: 描述处理项目的文字, 默认是$'it'$, 例如: $100it/s$, 处理照片的话设置为$'img'$,则为 $100img/s$
- unit_scale: 自动根据国际标准进行项目处理速度单位的换算, 例如 $100000it/s >> 100kit/s$

本次代码的实操如下

```
processBar = tqdm(trainDataLoader, unit='step')
...//代码主体
processBar.set_description(
    "[%d/%d] Loss: %.8f, Acc: %.8f" % (
        epoch, Epoches, loss.item(), accuracy.item()
    )
)
```

原始参数

参数名称	$Batch\ Size$	$Epoch$	$learning\ rate$	$P_{drop\ out}$
初始值	100	10	10^{-4}	0.7

按照上面的参数跑一边模型之后,发现其实初始参数的值并不差,结果如下

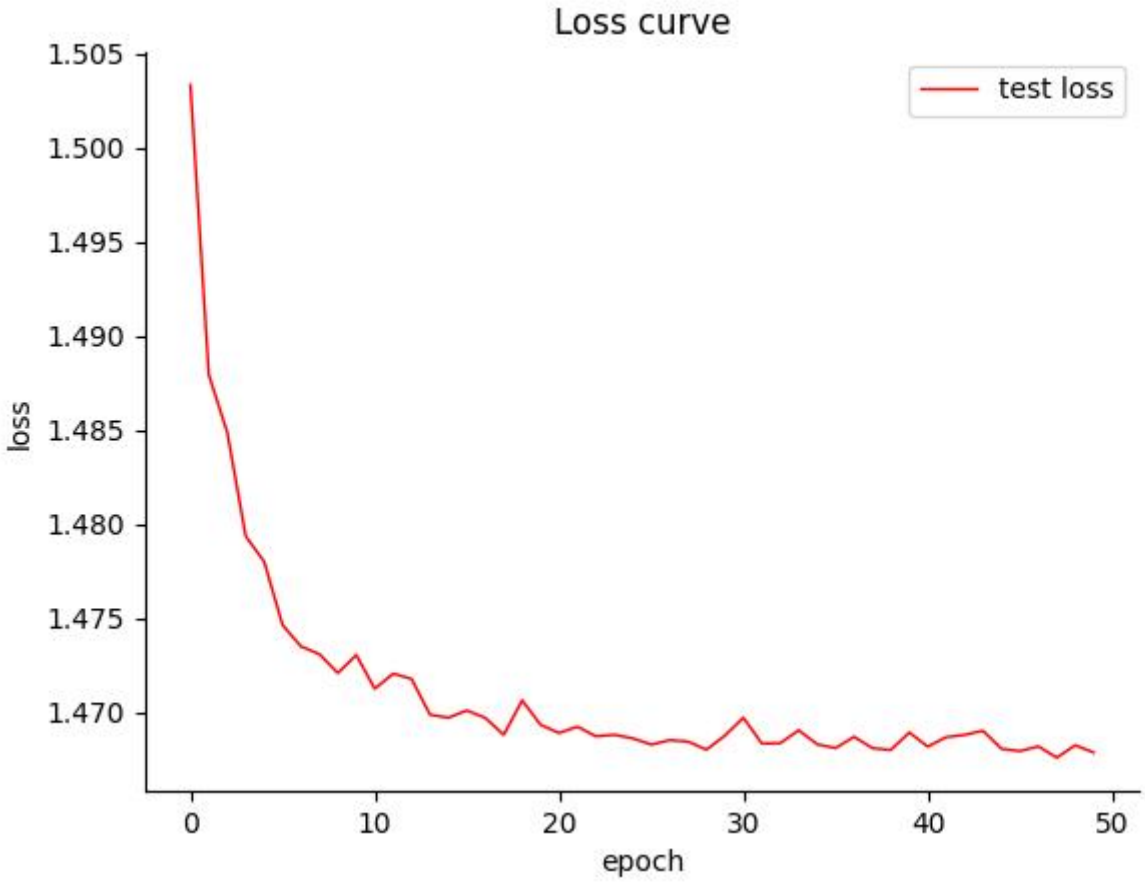
```
[1/10] Loss: 1.52360702, Acc: 0.97000003, Test Loss: 1.51450741, Test Acc: 0.95480001: 100%|██████████| 600/600 [01:09<00:00,  8.67step/s]
[2/10] Loss: 1.49443698, Acc: 0.94999999, Test Loss: 1.49167311, Test Acc: 0.97359997: 100%|██████████| 600/600 [01:11<00:00,  8.41step/s]
[3/10] Loss: 1.48700953, Acc: 0.98000002, Test Loss: 1.48639584, Test Acc: 0.97759998: 100%|██████████| 600/600 [01:08<00:00,  8.75step/s]
[4/10] Loss: 1.48936021, Acc: 0.97000003, Test Loss: 1.47975206, Test Acc: 0.98339999: 100%|██████████| 600/600 [01:10<00:00,  8.57step/s]
[5/10] Loss: 1.48094010, Acc: 0.99000001, Test Loss: 1.47876537, Test Acc: 0.98390001: 100%|██████████| 600/600 [01:10<00:00,  8.54step/s]
[6/10] Loss: 1.48112321, Acc: 1.00000000, Test Loss: 1.47542310, Test Acc: 0.98699999: 100%|██████████| 600/600 [01:09<00:00,  8.64step/s]
[7/10] Loss: 1.48685014, Acc: 0.94999999, Test Loss: 1.47620726, Test Acc: 0.98650002: 100%|██████████| 600/600 [01:09<00:00,  8.62step/s]
[8/10] Loss: 1.48016393, Acc: 0.98000002, Test Loss: 1.47398257, Test Acc: 0.98769999: 100%|██████████| 600/600 [01:10<00:00,  8.55step/s]
[9/10] Loss: 1.47419918, Acc: 0.99000001, Test Loss: 1.47237408, Test Acc: 0.99000001: 100%|██████████| 600/600 [01:10<00:00,  8.57step/s]
[10/10] Loss: 1.47290969, Acc: 0.99000001, Test Loss: 1.47186947, Test Acc: 0.99019998: 100%|██████████| 600/600 [01:09<00:00,  8.63step/s]
1.4718694686889648
0.9901999831199646
```

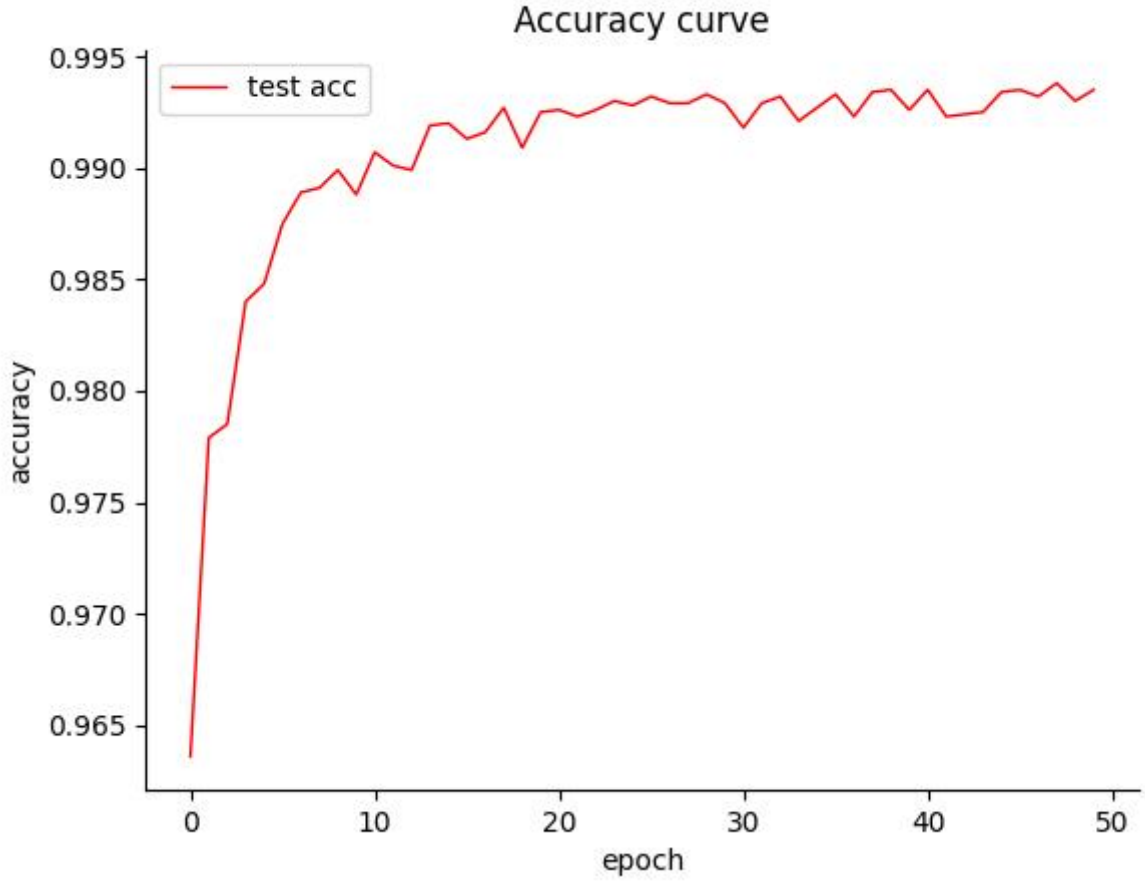
参数寻优

其实一般情况下,当我们选取一个比较大的$Batch-Size$的时候,说明它每次步数比较大,按照极端情况来看,假设我的$Batch-Size=N$,其中N为样本总数,那么很显然我们一次就会抓取完所有的样本,那就应该适当的放大$epoch$

在这里是因为我们仅仅做的是一个参数寻优,只是比较它们哪一种组合的效果最好,因此我们选择定住某一个参数之后再寻找其他参数的最优值

为了降低训练时间,我们先确定$epoch$的最优值,设定最优$epoch$的参数范围$[1,50]$,用for循环重复运行函数主体,将得到的$loss$和$accuracy$保存下来,并据此画出其随$epoch$的曲线图


Loss curve
test loss
1.505
1.500
1.495
1.490
1.485
1.480
1.475
1.470
loss
0
10
20
30
40
50
epoch


Accuracy curve
test acc
0.995
0.990
0.985
0.980
0.975
0.970
0.965
accuracy
0
10
20
30
40
50
epoch

观察完上图之后我们发现$Accuracy$在23左右开始趋于收敛,但由于是在测试集上的结果,准确率会稍有上下浮动.

我认为我们要寻找的最优参数既要收敛,又不能太大,从上图来看,可能在后面还有比我所定义的"最优参数"跑出来的$Accuracy$还要高的$epoch$,但总体来看提升并不明显,因为我的$Accuracy$已经达到99.5%的区间了,而加大$epoch$提升的那一点准确率却是用巨大的时间成本换来的,因此我从保存下来的epoch的值中选取了$epoch_{best}=26$

以$epoch-Accuracy$为例附上画出曲线的代码,其中$test_acc.txt$是之前跑数据时保存下来的结果,要绘制$loss$的曲线图只需要更改文件名(之前也要保存$test_loss.txt$)和纵坐标的值就可以了

```python
import numpy as np
from matplotlib import pyplot as plt


def data_read(dir_path):
    with open(dir_path, "r") as f:
        raw_data = f.read()
        data = raw_data[1:-1].split(", ")  # [-1:1]是为了去除文件中的前后中括号"[]"
    return np.asfarray(data, float)

if __name__ == "__main__":
    test_loss_path = r"D:\sukkart\my projects\neural_network\exp2\test_acc.txt"

    y_test_acc = data_read(test_loss_path)
    x_test_acc = range(len(y_test_acc))

    plt.figure()

    # 去除顶部和右边框框
    ax = plt.axes()
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)

    plt.xlabel('epoch')  # x轴标签
    plt.ylabel('accuracy')  # y轴标签

    # 以x_test_loss为横坐标，y_test_loss为纵坐标，曲线宽度为1，实线，增加标签，训练损失，
    # 默认颜色，如果想更改颜色，可以增加参数color='red',这是红色。
    plt.plot(x_test_acc, y_test_acc, color="red", linewidth=1, label="test acc")
    plt.legend()
    plt.title('Accuracy curve')
    plt.savefig('epoch_test_acc.jpg')
    plt.show()

```

在确定了$epoch$的最优值之后,我们来寻找$Batch-Size,learning-rate,P_{drop\ out}$的最优值

给定三个参数的备选列表,使用一个三重for循环来找寻它们的最优值

```python
Batch_size_list = [16, 32, 64, 128, 256,512]
learning_rate_list = [1e-3, 1e-4, 1e-5, 1e-6]
keep_prob_rate_list = [0.5, 0.6, 0.7, 0.8]
```

```
for Batch_size in Batch_size_list:
    for learning_rate in learning_rate_list:
        for keep_prob_rate in keep_prob_rate_list:
```

在本机上测试没有语法错误之后选择丢到itc的服务器上跑。另外，为了更好地比较结果，将参考代码的保存4位小数改位保存8位小数

下面附上参数寻优及保存结果部分代码

```
params = ['Batch_size', 'learning_rate', 'keep_prob_rate', 'loss', 'accuracy']//列名
with open('params.csv', 'w', newline='') as csv_file:
    writer = csv.writer(csv_file)
    writer.writerow(['Batch_size', 'learning_rate', 'keep_prob_rate', 'loss',
'accuracy'])
    writer.writerow(params)
for Batch_size in Batch_size_list:
    for learning_rate in learning_rate_list:
        for keep_prob_rate in keep_prob_rate_list:
            params = []//每次保存五个值,在循环开始时清空
            params.append(Batch_size)
            params.append(learning_rate)
            params.append(keep_prob_rate)
            ...//程序主体
            print(test_loss)//测试集上的loss值
            print(test_acc)//测试集上的准确率
            params.append(test_loss)
            params.append(test_acc)
            with open('params.csv', 'a', newline='') as csv_file:
                writer = csv.writer(csv_file)
                writer.writerow(params)
            print(params)
```

保存出来的csv文件有96条数据,我们对其排序后取最高值,下面时是排序后的部分结果

Batch_size	learning_rate	keep_prob_rate	loss	accuracy
16	0.0001	0.8	1.467636943	0.993599951
32	0.0001	0.5	1.467550159	0.992212415
16	0.0001	0.6	1.469101191	0.992199957
16	0.0001	0.7	1.468709588	0.992199957
16	0.0001	0.5	1.469033718	0.991799951
32	0.0001	0.6	1.468098044	0.991413713
32	0.0001	0.7	1.4681952	0.991413713
16	1.00E-05	0.7	1.472124696	0.99059999
32	0.0001	0.8	1.468806624	0.990515172
16	1.00E-05	0.6	1.472540617	0.989899993
16	1.00E-05	0.8	1.472167134	0.989600003
16	1.00E-05	0.5	1.472776651	0.98939997
64	0.0001	0.8	1.4680022	0.988753974
64	0.0001	0.6	1.468378186	0.988455415
64	0.0001	0.7	1.468384743	0.988057315
64	0.0001	0.5	1.468611717	0.987758756
32	1.00E-05	0.7	1.4738096	0.987120569
32	1.00E-05	0.6	1.474129319	0.986421704
32	1.00E-05	0.5	1.474841952	0.986122191
32	1.00E-05	0.8	1.475778461	0.985523164
128	0.0001	0.7	1.468733668	0.98170495
128	0.0001	0.5	1.468526602	0.981606066
128	0.0001	0.6	1.469035864	0.981507123
64	1.00E-05	0.7	1.47725141	0.981190324
64	1.00E-05	0.6	1.477315545	0.980991244
64	1.00E-05	0.8	1.477591276	0.980792224
64	1.00E-05	0.5	1.478433013	0.980493665
128	0.0001	0.8	1.469897389	0.980320454
64	0.001	0.5	1.477380872	0.979000807

基于上述方法,寻得最优参数

参数名称	$Batch\ Size$	$Epoch$	$learning\ rate$	$P_{drop\ out}$
参数值	16	26	10^{-4}	0.8

问题记录

1. 由于先入为主的思想,一开始并没有想到要画$loss-epoch$和$accuracy-epoch$曲线,因为$epoch$的初始值为10,我想当然的就想在$[10,20,30,40,50]$里面找到一个最优解,后面仔细一想发现显然不对,于是将范围改为了$[1,50]$
2. 知道范围之后,需要保存最后的值来画曲线,并不需要保存中间训练集训练过程的值,我一开始因为想着保存中间结果却又不知以何种格式保存花了不少时间

3. 确定完$epoch$之后,发现跑一次模型需要$5-6min$左右,由于剩下的三个参数有96种组合方式,那么寻优的时间将达到$9-10h$,一开始放在itc上跑总是会掉线,后来学习$nohup$指令可以后台挂起运行代码了,之后中间有一次过了一晚上确实跑出结果了,但由于那时保存文件使用的$'w'$覆盖参数而非$'a'$追加参数导致结果并没有保存下来,后面再次使用$nohup$指令的时候每次过一段时间登录itc还是会掉线,因此最后想到了分段跑保存结果,每次选择30种组合放进球跑,这样子模型运行一次的时间在$3h$左右,即使itc掉线结果也已经保存,最后把四份结果拼起来得到最终的csv文件

实验心得

本次实验通过$Pytorch$实现了卷积神经网络的搭建与训练过程，通过改变神经网络的超参数，比较前后模型学习能力，与理论知识相结合，有很大的收获。根据实验结果对比分析可以看出，随着epoch的增大，模型的拟合能力越来越强，但是，随着epoch增加，模型也会发生过拟合，如果增大训练层数也会不可避免会发生梯度消失或者梯度爆炸等问题。同时，我也得到了如下心得体会：

1. 卷积网络入门其实没有想象的那么难，但是越往高处走越来越需要我们去细心琢磨。
2. 在卷积神经网络中，有必要去深入了解一些经典的卷积神经网络。由浅入深，例如：$LeNet$、$AlexNet$、$VGG16$、$Inception$网络、$ResNet$网络等等。
3. 在了解这些网络的时候应该多考虑别人为什么要这么做，这样做又什么好处。
4. 在学习卷积神经网络的过程中，应该更多地去做一些实践，这样不仅仅能够更加深入的去理解网络的构成流程而且能产生更大的兴趣去学习去钻研。
5. 在学习卷积网络的过程中，应该适当的去修改一些经典网络，看看在修改过后有什么的不同，在从这些不同的地方吸取一些教训，然后在以后碰到类似的问题的时候能够有一些经验。
6. 其实在做深度学习实验中，写代码花不了多少时间，最花时间的是跑代码，特别是当某一次跑出结果之后发现代码有问题是很令人崩溃的，因此在编写代码的过程中尽量做到逻辑清晰，减少失误