分数：	

深度学习

2020 - 2021 学年度第 二 学期

实践报告

任课教师：孔雨秋

题目：全连接网络识别手写数字集（MNIST）

院(系)：人工智能学院

班级：电智 1902

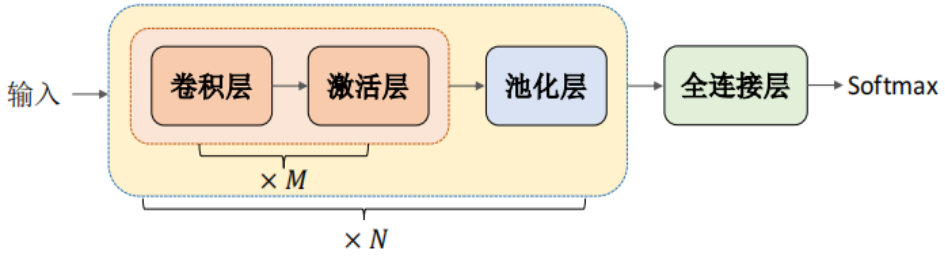
学号：201981498

姓名：李铭潍

任务：使用全连接网络识别手写数字集（MNIST）

1. 理论基础

1. 卷积神经网络



实验中搭建的网络模型为卷积神经网络(CNN)，含有卷积(convolution)，激活(activation)和池化(pooling)操作。

(1) 卷积

卷积过程就是 kernel 所有权重与其在输入图像上对应元素乘积之和。

kernel_size 规定一共进行几轮卷积操作，feature map 会增加 n 个维度，通常认为是多抓取 n 个特征。；

stride 为步长,表示每卷积一次向右或向下前进几个单位;

padding 表示在边缘补偿几个值为 0 的像素点，保证图像边缘点也可以被充分卷积，同时保证 feature map 与原始图像大小不变。

(2) 激活

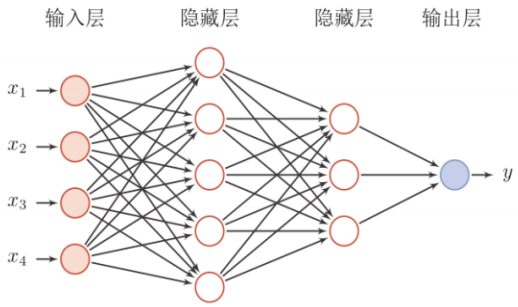
卷积之后，通常会加入偏置(bias)，并引入非线性激活函数(activation function)，激活函数为 sigmoid 函数或者 ReLU 函数，本实验使用的是 ReLU 函数 h(x)=max(0,x)。

(3) 池化

池化是一种降采样操作(subsampling)，主要目标是降低 feature maps 的特征空间，或者可以认为是降低 feature maps 的分辨率。因为 feature map 参数太多，而图像细节不利于高层特征的抽取。

本实验使用的 Pooling 操作是最大值池化(Max pooling):如上图所示，2 * 2 的 Max pooling 就是取 4 个像素点中最大值保留。M

2. 全连接神经网络



全连接网络每一层的所有单元与上一层完全连接。

通常，除了输入层和输出层的其他层，都被认为是隐含层。

每一层等于上一层的所有输出加权重相乘相加再偏置后得出的结果，再通过一 层激活函数(本实验为 ReLU 函数)，传输到下一层。

当传播到最后一层时，根据 softmax 函数获得每一类别可能的概率，得到模型输出的结果，根据结果计算交叉熵损失，进行反向传播和优化， 优化权重和偏置。

3. 训练过程

按照多轮数，多批次地进行训练，不断优化其中的权重参数，以提高准确率。

首先读取一个批次的图片，根据现有模型输出结果，计算 loss 损失;

然后根据 loss 损失对梯度进行反向传播，优化器对参数进行优化，以期在下一轮训练中提高准确率。

4. 测试过程

读取测试数据图片和标签，将测试图片传入模型中得到训练结果，再将其与图片标签进行比较是否一致，将一致的数量除以测试集总数即可得到正确率，正确率可以表征模型训练的好坏。

5. 加载数据

Data_Loader 从存储空间中读取图片和标签，对图片进行预处理(统一尺寸，随机翻转，旋转角度，转换

为张量)，并传到模型中进行训练和测试。卷积神经网络：四层卷积层、两层池化层：

2. 程序实现

CNN. py　　卷积神经网络：四层卷积层、两层池化层

```
import torch
import torchvision as tv
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
import argparse
from matplotlib import pyplot as plt
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter('/home/dut_levi/CODE/python 基础/mnist')

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(torch.cuda.is_available())
# 定义是否使用 GPU

# 定义网络结构
class CNN(nn.Module):
    """卷积神经网络：四层卷积、两层池化层"""
    def __init__(self):
        super(CNN, self).__init__()  # 四层卷积、两层池化层卷积后使用批标准化处理加快收敛速度
        self.layer1 = nn.Sequential(                    # input_size=(1*28*28)
            nn.Conv2d(1, 16, kernel_size=3),    #第一层 input_channel = 1, output_channel = 16, 卷积核 = 3
            nn.BatchNorm2d(16),                 # output_channel = 16 防止梯度消失或者爆炸
        # inplace 为 True，将会改变输入的数据 ，否则不会改变原输入，只会产生新的输出
            nn.ReLU(inplace=True)
          )

        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=3), #第二层 input_channel = 16, output_channel = 32, 卷积核 = 3
            nn.BatchNorm2d(32),                 # output_channel = 32 防止梯度消失或者爆炸
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)    # 池化 kernel_size=2, stride=2
```

```
        )

        self.layer3 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3), # 第三层 input_channel = 32, output_channel = 64, 卷积核 = 3
            nn.BatchNorm2d(64),                    # output_channel = 64 防止梯度消失或者爆炸
            nn.ReLU(inplace=True)
        )

        self.layer4 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3)#第四层 input_channel = 64, output_channel = 128, 卷积核 = 3
            nn.BatchNorm2d(128),              # output_channel = 128 防止梯度消失或者爆炸
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)     # 第二次池化 kernel_size=2, stride=2
        )

        self.fc = nn.Sequential(
            nn.Linear(128 * 4 * 4, 1024),          # 全连接层：in_features = 128 * 4 * 4, out_features =
1024
            nn.ReLU(inplace=True),
            nn.Linear(1024, 128),                  # 全连接层：in_features = 1024, out_features = 128
            nn.ReLU(inplace=True),
            nn.Linear(128, 10)                     # 全连接层：in_features = 128, out_features = 10
        )
    # 定义前向传播过程，输入为 x
    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
    # nn.Linear()的输入输出都是维度为一的值，所以要把多维度的 tensor 展平成一维
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

parser = argparse.ArgumentParser()
parser.add_argument('--outf', default='./model/', help='folder to output images and model
checkpoints')  # 模型保存路径
parser.add_argument('--net', default='./model/net.pth', help="path to netG (to continue training)")
# 模型加载路径
opt = parser.parse_args()

# 超参数设置
EPOCH = 5  # 遍历数据集次数
BATCH_SIZE = 64  # 批处理尺寸(batch_size)
LR = 0.001  # 学习率

# 定义数据预处理方式
transform = transforms.ToTensor()
```

```
# 定义训练数据集
trainset = tv.datasets.MNIST(
        root='./data/',
        train=True,
        download=True,
        transform=transform)

# 定义训练批处理数据
trainloader = torch.utils.data.DataLoader(
        trainset,
        batch_size=BATCH_SIZE,
        shuffle=True,
)

# 定义测试数据集
testset = tv.datasets.MNIST(
        root='./data/',
        train=False,
        download=True,
        transform=transform)

# 定义测试批处理数据
testloader = torch.utils.data.DataLoader(
        testset,
        batch_size=BATCH_SIZE,
        shuffle=False,
)

# 定义损失函数 loss function 和优化方式（采用 SGD）
net = CNN().to(device)
criterion = nn.CrossEntropyLoss()     # 指定交叉熵损失函数，通常用于多分类问题上
optimizer = optim.SGD(net.parameters(), lr=LR, momentum=0.9)   # 指定优化器
losses = []
accs = []
# 训练
if __name__ == "__main__":

        for epoch in range(EPOCH):
                sum_loss = 0.0
                # 数据读取
                for i, data in enumerate(trainloader):
                        inputs, labels = data
                        inputs, labels = inputs, labels                    # input 设置转化为 CUDA 数据
                        inputs = inputs.to(device)
                        # 梯度清零
                        optimizer.zero_grad()
                        # forward + backward
```

```
            outputs = net(inputs).to(device)                    # output 设置转化为 CUDA 数据
            loss = criterion(outputs, labels.to(device))        # labels 设置转化为 CUDA 数据
            loss.backward()
            optimizer.step()
            # 每训练 100 个 batch 打印一次平均 loss
            sum_loss += loss.item()
            if i % 100 == 99:
                print('[%d, %d] loss: %.03f'
                      % (epoch + 1, i + 1, sum_loss / 100))
                sum_loss = 0.0

        # 每跑完一次 epoch 测试一下准确率
        with torch.no_grad():
            correct = 0
            total = 0
            for data in testloader:
                images, labels = data
                images, labels = images.to(device), labels.to(device)
                outputs = net(images)
                # 取得分最高的那个类
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum()
            print('第%d 个 epoch 的识别准确率为：%d%%' % (epoch + 1, (100 * correct /
total)))
    torch.save(net.state_dict(), 'net.pth')                     # 保存模型

dataiter = iter(trainloader)                                    # 迭代器 获取随机的图片
images, labels = dataiter.next()
for i in range(5):
    if predicted[i] == labels[i]:
        plt.imshow(testset.data[i].numpy(), cmap='gray')        # 画出分类正确的图片
        plt.title('%i' % testset.targets[i])                    # 显示图片标签
        writer.add_scalar('LOSS', loss, i)                      # 使用 tensorboard 显示 loss 标量
        writer.add_scalar('acc', correct, epoch)                # 使用 tensorboard 显示 accuracy 准确率

    else:
        plt.imshow(testset.data[i].numpy(), cmap='gray')        # 画出分类错误的图片
        plt.title('%i' % predicted[i])                          # 显示预测标签
        writer.add_scalar('LOSS', loss, i)                      # 使用 tensorboard 显示 loss 标量
        writer.add_scalar('acc', correct, epoch)                # 使用 tensorboard 显示 accuracy 准确率

    plt.show()
writer.add_graph(net, images)                                   # 使用 tensorboard 显示网络结构和图片
writer.close()
```

3. 实验

3.1 数据库介绍

导入 MNIST，内分 progress 与 raw，progress 分测试集与训练集两部分

MNIST 数据集中的每张图片由 28x28 个像素点构成,每个像素点用一个灰度值表示.在这里，我们将 28 x 28 的像素展开为一个一维的行向量,这些行向量就是图片数组里的行(每行 784 个值,每行代表一张图片).

3.2 定量实验结果展示

1.改变学习率得到数据：

(1) LR = 0.01

/home/dut_levi/.conda/envs/mm/bin/python3.8 /home/dut_levi/CODE/python 基础/mnist/CNN.py

True

[1, 100] loss: 0.613
[1, 200] loss: 0.119
[1, 300] loss: 0.083
[1, 400] loss: 0.067
[1, 500] loss: 0.064
[1, 600] loss: 0.054
[1, 700] loss: 0.050
[1, 800] loss: 0.050
[1, 900] loss: 0.051
第 1 个 epoch 的识别准确率为：98%
[2, 100] loss: 0.036
[2, 200] loss: 0.037
[2, 300] loss: 0.026
[2, 400] loss: 0.036
[2, 500] loss: 0.029
[2, 600] loss: 0.035
[2, 700] loss: 0.020
[2, 800] loss: 0.036
[2, 900] loss: 0.041
第 2 个 epoch 的识别准确率为：98%
[3, 100] loss: 0.017
[3, 200] loss: 0.022
[3, 300] loss: 0.018
[3, 400] loss: 0.021
[3, 500] loss: 0.023
[3, 600] loss: 0.026
[3, 700] loss: 0.026
[3, 800] loss: 0.021
[3, 900] loss: 0.022
第 3 个 epoch 的识别准确率为：99%
[4, 100] loss: 0.015
[4, 200] loss: 0.016
[4, 300] loss: 0.017
[4, 400] loss: 0.016
[4, 500] loss: 0.016
[4, 600] loss: 0.020
[4, 700] loss: 0.017
[4, 800] loss: 0.020
[4, 900] loss: 0.017
第 4 个 epoch 的识别准确率为：99%
[5, 100] loss: 0.011
[5, 200] loss: 0.006
[5, 300] loss: 0.009
[5, 400] loss: 0.012
[5, 500] loss: 0.013
[5, 600] loss: 0.009
[5, 700] loss: 0.014
[5, 800] loss: 0.022
[5, 900] loss: 0.015
第 5 个 epoch 的识别准确率为：99%

(2) LR = 0.001

/home/dut_levi/.conda/envs/mm/bin/python3.8 /home/dut_levi/CODE/python 基础/mnist/CNN.py

True

[1, 100] loss: 1.830
[1, 200] loss: 0.611
[1, 300] loss: 0.275
[1, 400] loss: 0.176
[2, 100] loss: 0.074
[2, 200] loss: 0.074
[2, 300] loss: 0.069
[2 , 400] loss: 0.071

[1, 500] loss: 0.148
[1, 600] loss: 0.127
[1, 700] loss: 0.106
[1, 800] loss: 0.103
[1, 900] loss: 0.092
第 1 个 epoch 的识别准确率为：97%
[2, 500] loss: 0.063
[2, 600] loss: 0.060
[2, 700] loss: 0.060
[2, 800] loss: 0.060
[2, 900] loss: 0.055
第 2 个 epoch 的识别准确率为：98%
[3, 100] loss: 0.045
[3, 200] loss: 0.046
[3, 300] loss: 0.045
[3, 400] loss: 0.047
[3, 500] loss: 0.048
[3, 600] loss: 0.045
[3, 700] loss: 0.046
[3, 800] loss: 0.044
[3, 900] loss: 0.044
第 3 个 epoch 的识别准确率为：98%
[4, 100] loss: 0.040
[4, 200] loss: 0.037
[4, 300] loss: 0.037
[4, 400] loss: 0.035
[4, 500] loss: 0.038
[4, 600] loss: 0.032
[4, 700] loss: 0.036
[4, 800] loss: 0.035
[4, 900] loss: 0.032
第 4 个 epoch 的识别准确率为：98%
[5, 100] loss: 0.031
[5, 200] loss: 0.025
[5, 300] loss: 0.033
[5, 400] loss: 0.024
[5, 500] loss: 0.030
[5, 600] loss: 0.032
[5, 700] loss: 0.032
[5, 800] loss: 0.031
[5, 900] loss: 0.027
第 5 个 epoch 的识别准确率为：99%
(3) LR = 0.0001
/home/dut_levi/.conda/envs/mm/bin/python3.8 /home/dut_levi/CODE/python 基础/mnist/CNN.py
True
[1, 100] loss: 2.270
[1, 200] loss: 2.177
[1, 300] loss: 2.063
[1, 400] loss: 1.928
[1, 500] loss: 1.771
[1, 600] loss: 1.579
[1, 700] loss: 1.387
[1, 800] loss: 1.192
[1, 900] loss: 1.007
第 1 个 epoch 的识别准确率为：88%
[2, 100] loss: 0.810
[2, 200] loss: 0.707
[2, 300] loss: 0.614
[2, 400] loss: 0.539
[2, 500] loss: 0.473
[2, 600] loss: 0.432
[2, 700] loss: 0.397
[2, 800] loss: 0.377
[2, 900] loss: 0.344
第 2 个 epoch 的识别准确率为：94%
[3, 100] loss: 0.311
[3, 200] loss: 0.286
[3, 300] loss: 0.276
[3, 400] loss: 0.259
[3, 500] loss: 0.249
[3, 600] loss: 0.246
[3, 700] loss: 0.227
[3, 800] loss: 0.227
[3, 900] loss: 0.204
[4, 100] loss: 0.200
[4, 200] loss: 0.196
[4, 300] loss: 0.182
[4, 400] loss: 0.189
[4, 500] loss: 0.179
[4, 600] loss: 0.178
[4, 700] loss: 0.167
[4, 800] loss: 0.164
[4, 900] loss: 0.164

第 3 个 epoch 的识别准确率为：95%　　第 4 个 epoch 的识别准确率为：96%

[5, 100] loss: 0.152
[5, 200] loss: 0.153
[5, 300] loss: 0.155
[5, 400] loss: 0.141
[5, 500] loss: 0.145
[5, 600] loss: 0.141
[5, 700] loss: 0.136
[5, 800] loss: 0.135
[5, 900] loss: 0.139

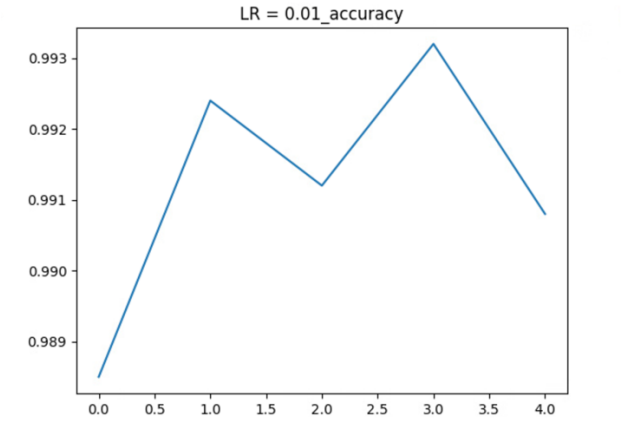
第 5 个 epoch 的识别准确率为：96%

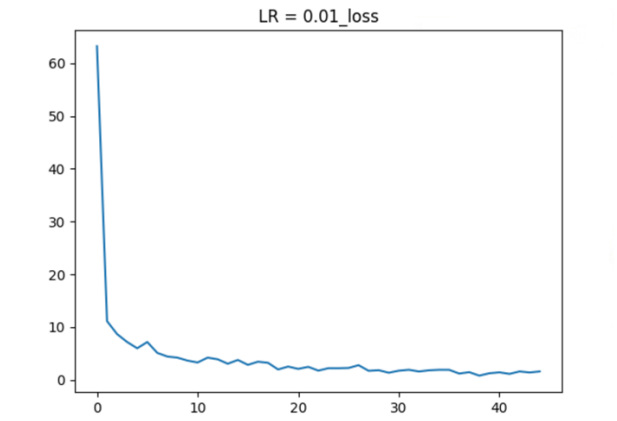
2. 改变批处理尺寸得到数据：略

3.3 定性实验结果展示

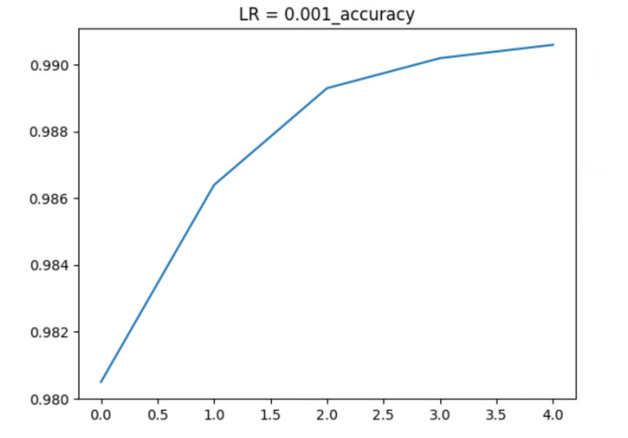
1. EPOCH = 5 Batchsize = 64 改变 LR(学习率)得到结果如下：

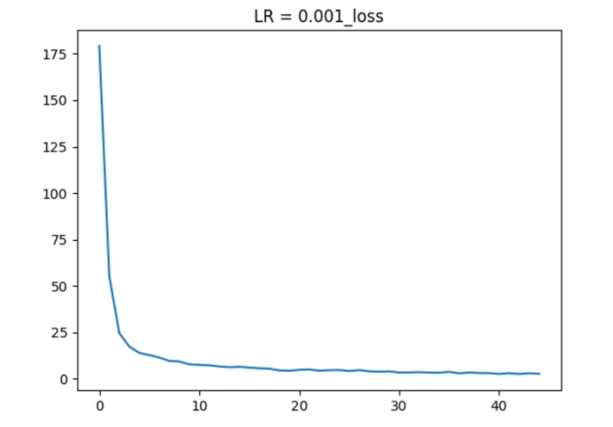
(1) Learning Rate = 0.01



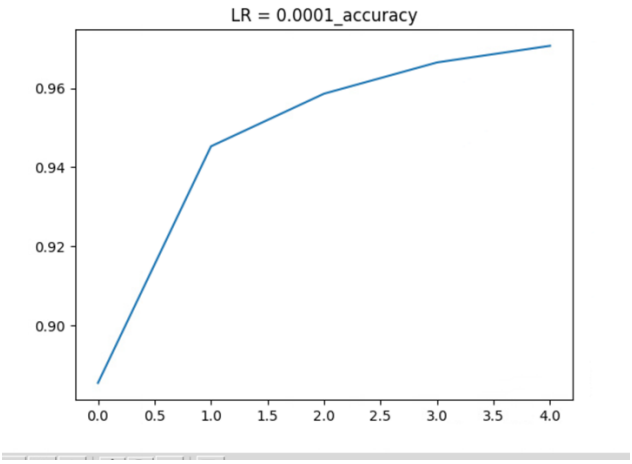


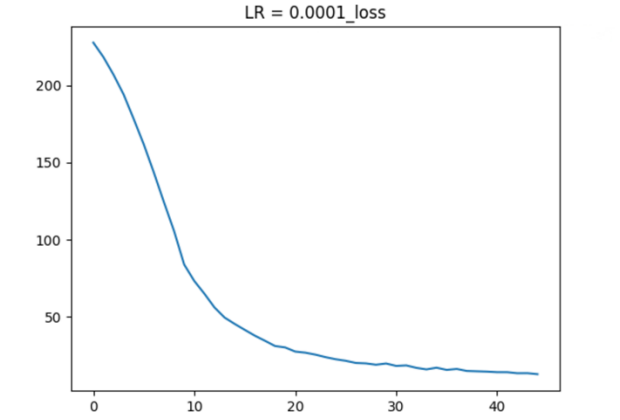
(2) Learning Rate = 0.001





(3) Learning Rate = 0.0001

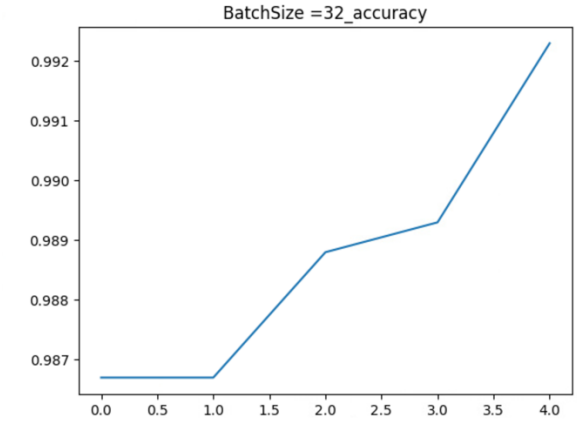


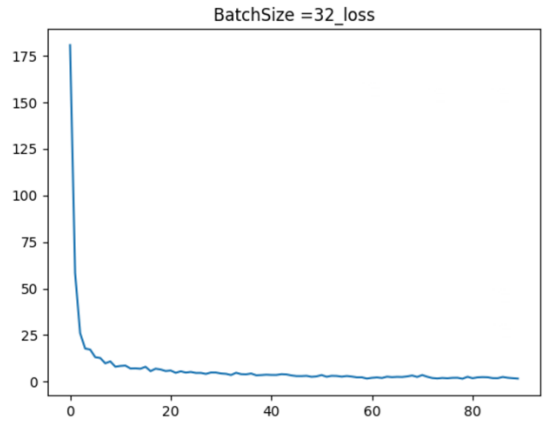


不同学习率每一轮训练后测试集的正确率　　模型输出结果和分类结果的交叉熵损失

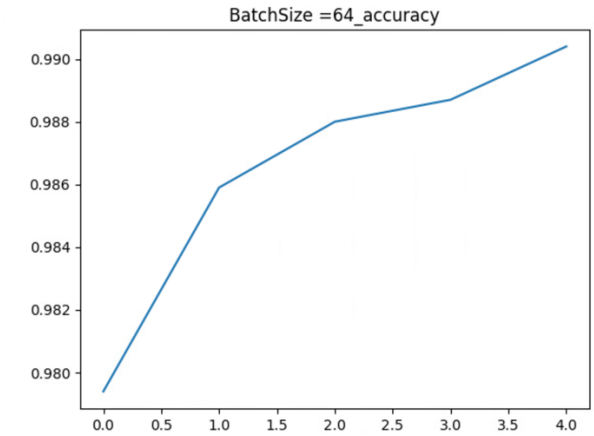
2. EPOCH = 5 LearningRate = 0.001 改变 BatchSize(批处理尺寸)得到结果如下：

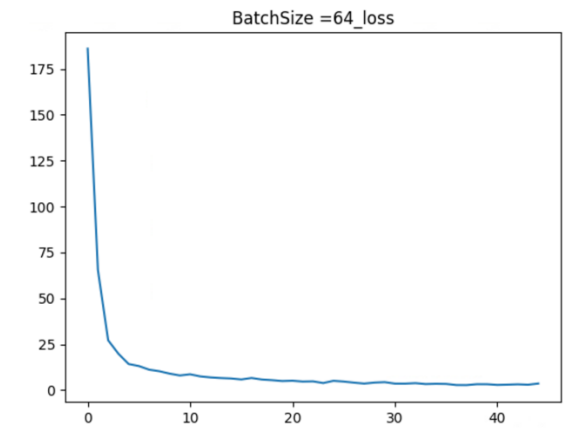
(1) BatchSize = 32



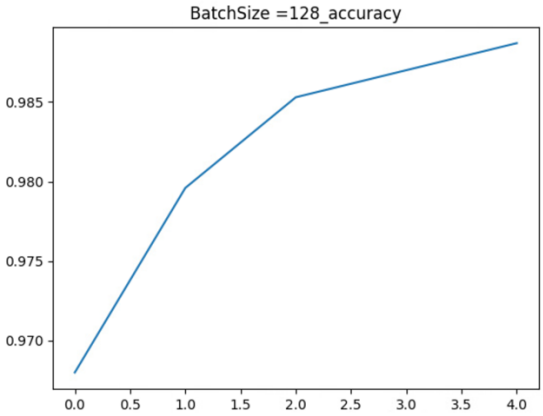


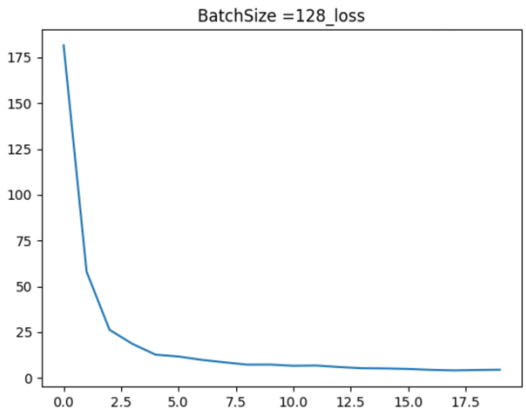
(2) BatchSize = 64





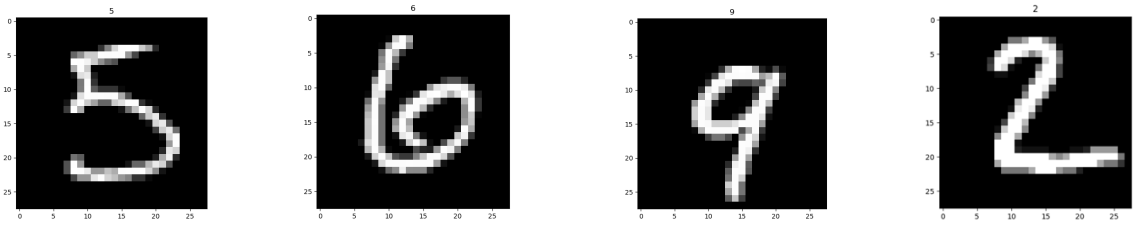
(3) BatchSize = 128



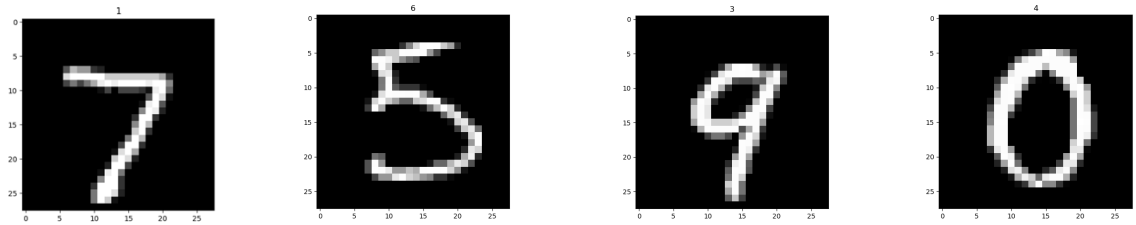


不同 BatchSize 每一轮训练后测试集的正确率　　模型输出结果和分类结果的交叉熵损失

如图是正确图片：



如图是错误图片：



3.4 分析实验结果

1.更改学习率得到模型准确率对比如下：
EPOCH = 5 Batchsize = 64 Change LR：

学习率	Epoch	1	2	3	4	5
0.01	Accuracy	98%	99%	99%	99%	99%
0.001	Accuracy	97%	98%	98%	99%	99%
0.0001	Accuracy	88%	94%	95%	96%	96%

固定学习率时，当到达收敛状态时，会在最优值附近一个较大的区域内摆动；
而当随着迭代轮次的增加而减小学习率，会使得在收敛时，在最优值附近一个更小的区域内摆动。

2.更改学习率得到模型准确率对比如下：
EPOCH = 5 LearningRate= 0.001 Change Batchsize：

BatchSize	Epoch	1	2	3	4	5
32	Accuracy	98%	99%	99%	99%	99%
64	Accuracy	97%	98%	98%	99%	99%
128	Accuracy	96%	97%	98%	99%	98%

Batch Size 设置合适时的优点：
1、通过并行化提高内存的利用率。就是尽量使 GPU 满载运行，提高训练速度。
2、单个 epoch 的迭代次数减少了，参数的调整也慢了，要达到相同的识别精度，需要更多的 epoch。
3、适当 Batch Size 使得梯度下降方向更加准确。
Batch Size 的增大能使网络的梯度更准确。

梯度的方差表示：
Var(g)=Var(1m∑mi=1g(xi,yi))=1m2Var(g(x1,y1)+g(x2,y2)+・・・+g(xm,ym))
由于样本是随机选取的，满足独立同分布，所以所有样本具有相同的方差 Var(g(xi,yi))
所以上式可以简化成 Var(g)=1mVar(g(xi,yi))
可以看出当 Batch Size 为 m 时，样本的方差减少 m 倍，梯度就更准确了。

3.总结与体会：
根据定量和定性的结果显示，通过 CNN 可以有效地提高模型判断图片数字类型的准确度。
改变神经网络超参数的同时，不断通过测试准确率的表现上体验炼丹的乐趣，调参 炼丹必经之路