基于 PyTorch 和 MNIST 数据集的手写数字识别项目

刘正浩 唐晨烨

1. 简介

神经网络是一种重要的机器学习算法，而卷积神经网络是一类重要的神经网络。卷积神经网络长期以来就是图像识别领域的核心算法之一，并在学习数据充足时有稳定的表现。PyTorch 是一个基于 Torch 的开源 Python 机器学习库，具有简洁、高效、快速的优点。利用 PyTorch 可以很方便地搭建一个神经网络。所以我们将利用 PyTorch 搭建卷积神经网络。

手写数字识别在计算机视觉领域占有非常重要的地位。是利用现有计算机技术及摄像头等设备，对日常生活中手写数字进行辨识的过程。在财务处理、金融管理、税务管理等领域应用广泛。由于人工手写数字笔体不一，使得计算机进行智能识别时准确率较低。例如在处理日常银行支票或识别邮政编码时，极其微小的错误将会造成巨大的损失。因此，手写数字识别最重要的任务是提高识别准确率。

MNIST 数据集是 Yann LeCun 等人创建的著名的手写数字数据集，包含六万个训练数据和一万个测试数据，这些数据都以字节的形式保存。我们利用 MNIST 数据集进行网络的训练和测试，并得到了比较好的效果。

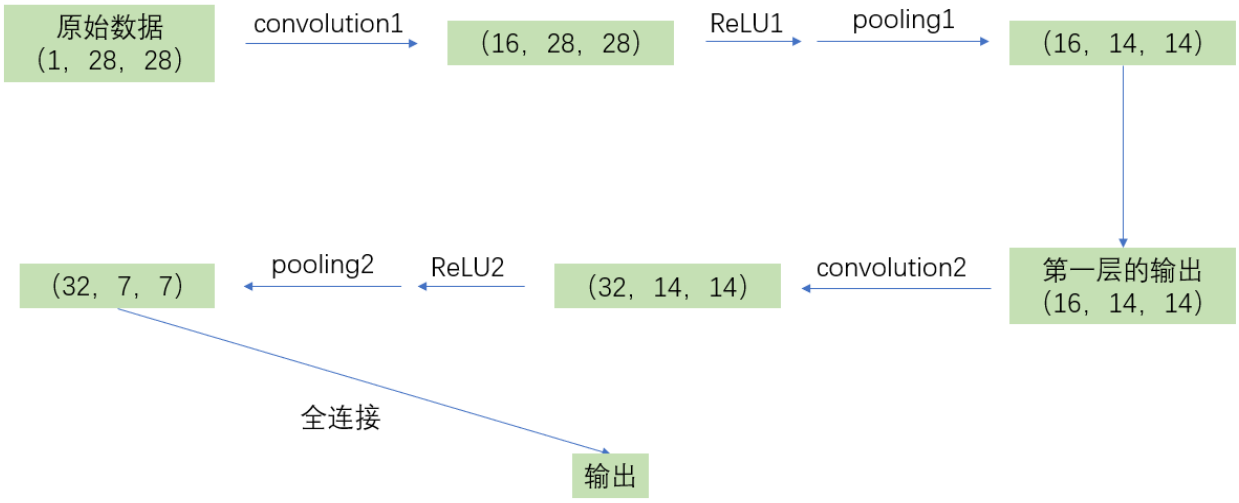
2. 超参数、网络结构、优化器与损失函数

a. 超参数

我们把 batch size 设置成 50，学习率为默认的 0.001。

b. 网络结构

我们使用 PyTorch 搭建了一个简单的卷积神经网络，它包含两个卷积层、两个池化层和一个全连接层，所有的过程在代码中都用 PyTorch 中的函数来实现。网络结构如下图所示。



数据集中的数据在导入的时候会转变为一个(1, 28, 28)的 Tensor，因为数据集中的图片用数组来表示，数组的边长为 28，通道数是 1，每一个数据的大小在 0 到 1 之间，表示像素点的灰度值；而 label 介于 0 到 0 之间，用一个长度为 10 的向量表示，其中 1 所在的位置就是标签数，例如标签 0 表示为[1,0,0,0,0,0,0,0,0,0]。

在第一个卷积层中，卷积核的数量为 16，大小为 5*5，步长为 1，padding 为 2，这样输出的

Tensor 长和宽仍为 28，高变为 16；最大池化的 kernel 为 2，池化后 Tensor 长和宽为 14，高为 16。

在第二个卷积层中，卷积核的数量为 32，大小、步长和 padding 都和第一个卷积层相同，这样输出 Tensor 的形状为长和宽都是 14，高为 32；接下来池化的 kernel 仍然是 2，输出的 Tensor 形状变为（32，7，7）。

在展平的过程中，我们将第二个池化层输出的所有 Tensor 展平成(batch_size, 32*7*7)的 Tensor，并进行全连接，通过十个数的最大值来判断一个图片上是哪个数字。

c. 优化器

在众多优化器中，Adam 一般来说是收敛速度最快的优化器。它综合了 Momentum 的更新方向策略和 RMSProp 的计算衰减系数策略，因此很好地结合了这两种优化器的优点，Momentum 帮助摆脱震荡，从而加速收敛；同时 RMSProp 可以有效避免降速太快而且没有办法收敛到全局最优情况的发生。相较于传统的 SGD，Adam 可以为不同的参数设置独立的自适应学习率。

d. 损失函数

在损失函数方面，我们使用了比较常见的交叉熵损失函数。这是分类问题（例如本文中的手写数字识别）最常见的设置。

3. 代码实现

a. MNIST 数据集导入

因为 torchvision 中的一个 package（*torchvision.datasets*）包含了 MNIST 数据集的处理，即 *torchvision.datasets.MNIST*()函数，这就为数据的处理提供了方便。

数据集文件以.pt 格式存储，在使用时我们*torchvision.transforms.ToTensor*()函数来读取数据，并转换成 Tensor 形式。Tensor 形式的数据能够被 PyTorch 中的函数进行运算操作。

```
# 加载训练数据
train_data = torchvision.datasets.MNIST(
    root='./data/',
    train=True,
    transform=torchvision.transforms.ToTensor(),
    download=DOWNLOAD_MNIST,
)

# 分批训练，一次输入五十个样本，每个样本有一个通道，大小为 28*28 (50, 1, 28, 28)
train_loader = Data.DataLoader(
    dataset=train_data,
    batch_size=BATCH_SIZE,
    shuffle=True
)

# 加载测试数据，为节约时间，只提取前两千个进行测试
test_data = torchvision.datasets.MNIST(root='./data/',train=False)

# 把测试数据规格从 (2000, 28, 28) 变成 (2000, 1, 28, 28)，取值在(0,1)范围内
test_x = torch.unsqueeze(test_data.test_data, dim=1).type(torch.FloatTensor)[:20
00] / 255.
```

```
# 测试数据对应的标签，因为是是个数字所以为 0~9
test_y = test_data.test_labels[:2000]
```

b. 搭建 CNN 网络

在网络中使用的卷积、激活、池化等操作都可以使用 PyTorch 中已经定义好的函数，所以搭建网络比较方便，直接调用这些函数就可以了。这也是我们选用 PyTorch 的初衷：让整个过程比较简单、好理解。

在定义前向传播时需要注意，由于我们并没有在网络本身中加入展平这一操作，所以在全连接之前我们需要将第二个池化层输出的张量展平成为一个二维张量，这样才能进行后面的全连接。

```
# 定义 CNN 类
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # 第一个卷积层
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ), # 卷积
            nn.ReLU(),  # 激活，这里使用 ReLU 作为激活函数
            nn.MaxPool2d(kernel_size=2),  # 最大池化
        )
        # 第二个卷积层
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),  # 卷积
            nn.ReLU(),  # 激活
            nn.MaxPool2d(2),  # 最大池化
        )
        # 全连接层
        self.out = nn.Linear(32 * 7 * 7, 10)

    # 定义前向传播
    def forward(self, x):
        x = self.conv1(x)  # 第一层卷积
        x = self.conv2(x)  # 第二层卷积
        x = x.view(x.size(0), -1)
        output = self.out(x)  # 全连接
        return output, x
```

```
'''实例化 CNN 并打印网络结构'''
cnn = CNN() # 实例化
print(cnn) # 打印出网络结构，注：不是必需步骤，可以去掉
```

c. 定义优化器和损失函数

我们使用的 Adam 优化器和交叉熵损失函数都在 torch 中有定义，只要调用两个函数 *torch.optim.Adam*()和*torch.nn.CrossEntropyLoss*()即可。需要注意的是，*CrossEntropyLoss*()已经包含了 softmax 处理，因此不需要单独进行 softmax：

$$loss(x,class) = -\log\frac{\exp(x[class])}{\sum_j \exp(x[j])} = -x[class] + \log(\sum_j \exp(x[j]))$$

```
'''定义用于优化神经网络的函数'''
# 优化函数，用于优化参数，使用 Adam 作为优化器
optimizer = torch.optim.Adam(
    cnn.parameters(),  # 待优化参数的 dict
    lr=LR  # 学习率
)
# 损失函数，这里选用交叉熵损失函数（自带 softmax）
loss_func = nn.CrossEntropyLoss()
```

d. 训练网络

```
# 训练网络
for epoch in range(EPOCH):
    timeStart = time.time()
    for step, (b_x, b_y) in enumerate(train_loader):  # 加载数据

        output = cnn(b_x)[0]
        loss = loss_func(output, b_y)  # 计算损失
        optimizer.zero_grad()  # 梯度归零
        loss.backward()  # 反向传播
        optimizer.step()  # 应用梯度

        if step % 50 == 0:  # 每 50 步打印一次训练效果
            test_output, last_layer = cnn(test_x)
            pred_y = torch.max(test_output, 1)[1].data.numpy()
            accuracy = float((pred_y == test_y.data.numpy()).astype(int).sum())
                       / float(test_y.size(0))  # 计算精确度
            print('Epoch: ', epoch, '| train loss: %.4f' % loss.data.numpy(),
                  '| test accuracy: %.3f' % accuracy)
    timeEnd = time.time()
    print('run time:',timeEnd-timeStart)
```

e. 测试网络

在测试网络的过程中，我们先生成一个随机数，利用这个随机数来确定选择测试数据的区间。这样做的好处是：每一次测试所提取的测试数据都是不同的，这样才能反映出网络识别数字的真实水平。

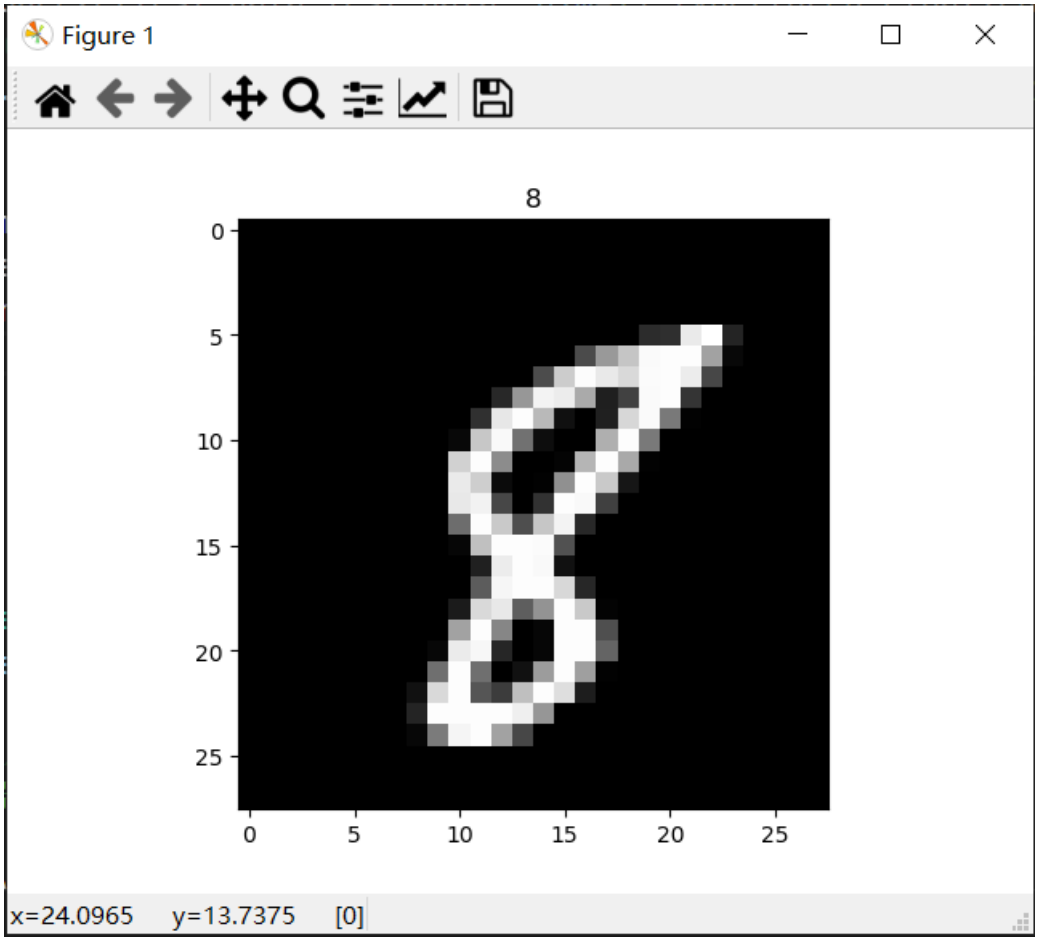
```
# 测试网络
random_num = random.randint(1, 1000)  # 生成随机数，用于确定测试数据的区间
test_output, _ = cnn(test_x[random_num: random_num + 10])
pred_y = torch.max(test_output, 1)[1].data.numpy()
print(pred_y, 'prediction number')
print(test_y[random_num: random_num + 10].numpy(), 'real number')
```

f. 展示一个图像（非必需步骤）

在导入测试数据集之后，可以利用下面的代码随机展示一个图像：

```
# 展示一个图像
random_num = random.randint(1, 60000)  # 生成随机数，确定展示图像的编号
plt.imshow(train_data.train_data[random_num].numpy(), cmap='gray')
plt.title('%i' % train_data.train_labels[random_num])
plt.show()
```

由于我们使用$random.randint()$函数生成的随机数来确定所选数据的编号，因此代码每一次运行时展示出的图像都是不一样的。这是某一次运行展示出的图像：



完整的代码已经上传 Github：https://github.com/sdescat0301/pytorch_mnist

4. 展示结果

```
Epoch:  0 | train loss: 2.3266 | test accuracy: 0.100
Epoch:  0 | train loss: 0.4716 | test accuracy: 0.847
Epoch:  0 | train loss: 0.3628 | test accuracy: 0.891
Epoch:  0 | train loss: 0.2837 | test accuracy: 0.907
Epoch:  0 | train loss: 0.0917 | test accuracy: 0.931
Epoch:  0 | train loss: 0.0692 | test accuracy: 0.943
Epoch:  0 | train loss: 0.1151 | test accuracy: 0.945
Epoch:  0 | train loss: 0.0769 | test accuracy: 0.950
Epoch:  0 | train loss: 0.0598 | test accuracy: 0.956
Epoch:  0 | train loss: 0.0874 | test accuracy: 0.963
Epoch:  0 | train loss: 0.0632 | test accuracy: 0.959
Epoch:  0 | train loss: 0.0234 | test accuracy: 0.964
Epoch:  0 | train loss: 0.0588 | test accuracy: 0.969
Epoch:  0 | train loss: 0.1401 | test accuracy: 0.970
Epoch:  0 | train loss: 0.0411 | test accuracy: 0.972
Epoch:  0 | train loss: 0.2309 | test accuracy: 0.968
Epoch:  0 | train loss: 0.0420 | test accuracy: 0.966
Epoch:  0 | train loss: 0.1319 | test accuracy: 0.976
Epoch:  0 | train loss: 0.0242 | test accuracy: 0.977
Epoch:  0 | train loss: 0.1427 | test accuracy: 0.976
Epoch:  0 | train loss: 0.0409 | test accuracy: 0.977
Epoch:  0 | train loss: 0.0158 | test accuracy: 0.978
Epoch:  0 | train loss: 0.0440 | test accuracy: 0.978
Epoch:  0 | train loss: 0.1043 | test accuracy: 0.972
[1 5 9 7 3 4 9 6 6 5] prediction number
[1 5 9 7 3 4 9 6 6 5] real number
```

```
Epoch:  0 | train loss: 2.3068 | test accuracy: 0.174
Epoch:  0 | train loss: 0.6202 | test accuracy: 0.823
Epoch:  0 | train loss: 0.4900 | test accuracy: 0.894
Epoch:  0 | train loss: 0.0716 | test accuracy: 0.922
Epoch:  0 | train loss: 0.4648 | test accuracy: 0.931
Epoch:  0 | train loss: 0.0510 | test accuracy: 0.947
Epoch:  0 | train loss: 0.2330 | test accuracy: 0.953
Epoch:  0 | train loss: 0.4641 | test accuracy: 0.949
Epoch:  0 | train loss: 0.0590 | test accuracy: 0.965
Epoch:  0 | train loss: 0.0623 | test accuracy: 0.962
Epoch:  0 | train loss: 0.0186 | test accuracy: 0.957
Epoch:  0 | train loss: 0.0507 | test accuracy: 0.973
Epoch:  0 | train loss: 0.0985 | test accuracy: 0.971
Epoch:  0 | train loss: 0.0573 | test accuracy: 0.974
Epoch:  0 | train loss: 0.0215 | test accuracy: 0.965
Epoch:  0 | train loss: 0.0970 | test accuracy: 0.968
Epoch:  0 | train loss: 0.2300 | test accuracy: 0.977
Epoch:  0 | train loss: 0.0762 | test accuracy: 0.978
Epoch:  0 | train loss: 0.0773 | test accuracy: 0.979
Epoch:  0 | train loss: 0.0493 | test accuracy: 0.976
Epoch:  0 | train loss: 0.3605 | test accuracy: 0.976
Epoch:  0 | train loss: 0.1187 | test accuracy: 0.973
Epoch:  0 | train loss: 0.2787 | test accuracy: 0.978
Epoch:  0 | train loss: 0.0290 | test accuracy: 0.981
[8 2 1 2 9 7 5 9 2 6] prediction number
[8 2 1 2 9 7 5 9 2 6] real number
```

上面两图是训练、测试各运行一次，也就是程序运行一次之后输出的结果。通过分析我们可以看到，随着训练的进行，识别数字的准确率也越来越高，在测试网络的过程中，程序对数字的识别已经相当准确。但训练的损失还有较大的波动，网络还不够稳定。所以我们构建的神经网络还需要进一步的优化，才能达到更好的效果。事实上，许多使用著名卷积神经网络 LeNet-5 或 AlexNet 的手写数字识别项目都有着稳定的表现。

下图是程序打印出的 CNN 结构：

```
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (out): Linear(in_features=1568, out_features=10, bias=True)
)
```

5. 提出一个问题

我们在讨论如何优化这个网络时提出了一个疑问:如果在网络中第二个池化层的后面再增加一个卷积层，训练过程会不会发生变化？如果发生了变化，网络本身是变得更稳定还是更不稳定？或者

说，训练相同的批次达到的效果和所用的时间会发生什么变化？

为了验证我们的想法，我们在原来代码的基础上增加了一个卷积层：

```
# 第三个卷积层
self.conv3 = nn.Sequential(
    nn.Conv2d(32, 64, 5, 1, 2),
    nn.ReLU()
)
```

这里没有加入池化层的原因是，第二个池化层输出的张量大小为（32，7，7），无法把池化的 kernel 定为 2.

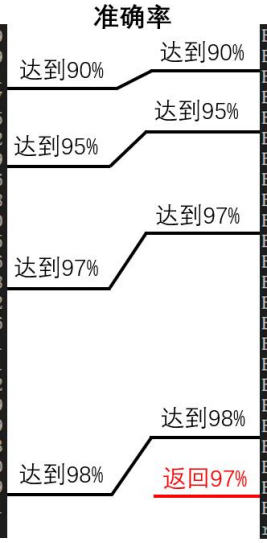
同时，由于网络结构改变了，前向传播的过程也要相应地变化，加入第三个卷积层：

```
# 定义前向传播
def forward(self, x):
    x = self.conv1(x)  # 第一层卷积
    x = self.conv2(x)  # 第二层卷积
    x = self.conv3(x)  # 第三层卷积
    x = x.view(x.size(0), -1)
    output = self.out(x)  # 全连接
    return output, x
```

我们分别测试了若干次改变前与改变后的网络，在分析的过程中我们发现这两组对照组的差别具有普遍性。我们选出了两组具有代表性的训练过程进行对比：

双层

```
Epoch:  0 | train loss: 2.3056 | test accuracy: 0.149
Epoch:  0 | train loss: 0.5086 | test accuracy: 0.819
Epoch:  0 | train loss: 0.3985 | test accuracy: 0.891
Epoch:  0 | train loss: 0.1235 | test accuracy: 0.907
Epoch:  0 | train loss: 0.1048 | test accuracy: 0.935
Epoch:  0 | train loss: 0.0995 | test accuracy: 0.942
Epoch:  0 | train loss: 0.0843 | test accuracy: 0.949
Epoch:  0 | train loss: 0.0689 | test accuracy: 0.956
Epoch:  0 | train loss: 0.0649 | test accuracy: 0.958
Epoch:  0 | train loss: 0.4816 | test accuracy: 0.960
Epoch:  0 | train loss: 0.0849 | test accuracy: 0.965
Epoch:  0 | train loss: 0.0786 | test accuracy: 0.966
Epoch:  0 | train loss: 0.0364 | test accuracy: 0.968
Epoch:  0 | train loss: 0.0127 | test accuracy: 0.972
Epoch:  0 | train loss: 0.1451 | test accuracy: 0.966
Epoch:  0 | train loss: 0.0518 | test accuracy: 0.971
Epoch:  0 | train loss: 0.2065 | test accuracy: 0.971
Epoch:  0 | train loss: 0.0222 | test accuracy: 0.972
Epoch:  0 | train loss: 0.1235 | test accuracy: 0.979
Epoch:  0 | train loss: 0.0385 | test accuracy: 0.979
Epoch:  0 | train loss: 0.0888 | test accuracy: 0.973
Epoch:  0 | train loss: 0.1073 | test accuracy: 0.970
Epoch:  0 | train loss: 0.1972 | test accuracy: 0.969
Epoch:  0 | train loss: 0.0639 | test accuracy: 0.981
run time: 145.05201315879822
```



三层

```
Epoch:  0 | train loss: 2.3033 | test accuracy: 0.098
Epoch:  0 | train loss: 0.3707 | test accuracy: 0.868
Epoch:  0 | train loss: 0.2182 | test accuracy: 0.909
Epoch:  0 | train loss: 0.2444 | test accuracy: 0.927
Epoch:  0 | train loss: 0.1314 | test accuracy: 0.949
Epoch:  0 | train loss: 0.0832 | test accuracy: 0.953
Epoch:  0 | train loss: 0.1383 | test accuracy: 0.967
Epoch:  0 | train loss: 0.0935 | test accuracy: 0.968
Epoch:  0 | train loss: 0.1306 | test accuracy: 0.966
Epoch:  0 | train loss: 0.0107 | test accuracy: 0.968
Epoch:  0 | train loss: 0.1002 | test accuracy: 0.970
Epoch:  0 | train loss: 0.1525 | test accuracy: 0.967
Epoch:  0 | train loss: 0.0750 | test accuracy: 0.974
Epoch:  0 | train loss: 0.1315 | test accuracy: 0.971
Epoch:  0 | train loss: 0.0155 | test accuracy: 0.970
Epoch:  0 | train loss: 0.0438 | test accuracy: 0.977
Epoch:  0 | train loss: 0.0198 | test accuracy: 0.979
Epoch:  0 | train loss: 0.1390 | test accuracy: 0.977
Epoch:  0 | train loss: 0.1108 | test accuracy: 0.970
Epoch:  0 | train loss: 0.0593 | test accuracy: 0.977
Epoch:  0 | train loss: 0.0090 | test accuracy: 0.984
Epoch:  0 | train loss: 0.1636 | test accuracy: 0.980
Epoch:  0 | train loss: 0.0474 | test accuracy: 0.986
Epoch:  0 | train loss: 0.2363 | test accuracy: 0.976
run time: 215.6926567554474
```

总用时145秒

总用时216秒

可以看出，在增加了一层卷积层之后，训练网络使两个网络首次达到相同准确度所需要的批次数减少了；同时训练一次所需要的时间却增加了，而且在三卷积层网络的训练过程中损失一直在较大范围内波动。

我们通过分析得出结论：增加卷积层可以使首次达到相同识别准确率所需要的训练批次减少，但训练的总时间会增加，而且损失会在更大的范围内波动。所以增加卷积层并不是优化网络、减小损失的有效方法。

6. 总结

在这学期研讨课的学习中，我们初步了解了人工智能是什么，认识到了之前对于“人工智能”这一研究领域的认识的偏差。在听老师讲解的过程中，我们也了解了人工智能发展的三次浪潮，并且了解了一些经典的人工智能算法。

在这次搭建神经网络的过程中，我们仔细回顾了上课内容中的卷积神经网络部分之后，发现其实只用在上课时学到的知识是远远不够实际应用来搭建一个网络的，所以我们为了实验成功查阅了大量的资料，这中间也培养了我们的信息检索能力。

参考资料

[1]PyTorch 官网 https://pytorch.org/

[2]PyTorch 中文文档 https://pytorch-cn.readthedocs.io/zh/latest/

[3]MNIST 网站 http://yann.lecun.com/exdb/mnist/

[4]尹晓伟,王真真,孟庆林,陈书旺.基于改进的 LeNet-5 手写数字识别研究[J].信息通信,2019(03):17-18.

[5]莫烦 Python:CNN https://www.bilibili.com/video/av15997678?zw

[6]机器学习：各种优化器 Optimizer 的总结与比较
https://blog.csdn.net/weixin_40170902/article/details/80092628

[7]MNIST 手写数字识别（一）
https://blog.csdn.net/sinat_34328764/article/details/83832487