Lab1Part2

By 黄子豪 18302010034

代码基本架构

本次实验基于pytorch实现了卷积神经网络（cnn），能够完成图片识别分类问题。现在就最基本的功能实现讲解代码基本架构，不包括后续优化步骤：CNN.py使用pytorch搭建了一个*卷积层——ReLU层——池化层——卷积层——ReLU层——池化层——全连接线性层*的cnn神经网络，而Classifi.py则调用这个网络完成具体任务。

CNN

```
import torch.nn as nn

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(  # input shape (1,28,28)
            nn.Conv2d(in_channels=1,  # input height
                      out_channels=16,  # n_filter
                      kernel_size=5,  # filter size
                      stride=1,  # filter step
                      padding=2  # con2d出来的图片大小不变
                      ),  # output shape (16,28,28)
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)  # 2x2采样，output shape (16,14,14)

        )
        self.conv2 = nn.Sequential(nn.Conv2d(16, 32, 5, 1, 2),  # output shape
(32,7,7)
                                   nn.ReLU(),
                                   nn.MaxPool2d(2))
```

```
        self.out = nn.Linear(32 * 7 * 7, 12)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)  # flat (batch_size, 32*7*7)
        output = self.out(x)
        return output
```

在cnn类构造方法中就进行网络的初始化，第一套conv设置为*卷积层——ReLU层——池化层*的结构，并且根据输入向量设置好层的各项参数，第二套conv只在卷积层参数稍有不同，输出化为（32,7,7）；最终网络输出还需要过一层线性全连接层，以化成一个12维的向量。

Classifi

事实上本次lab实现cnn网络结构并不难，重点在于数据的处理，因此详细分析Classfi结构。

设置超参数：

```
# Hyper Parameters
EPOCH = 15
BATCH_SIZE = 5
LR = 0.001
```

将提前准备好的训练集和测试集加载进来，变成易操作的tensor数据;

```
train_loader = data.DataLoader(dataset=dset.ImageFolder('train',
transform=torchvision.transforms.Compose([
    torchvision.transforms.Grayscale(1),  # 单通道
    torchvision.transforms.ToTensor(),  # 将图片数据转成tensor格式
])), batch_size=BATCH_SIZE, shuffle=True)

test_x = data.DataLoader(dataset=dset.ImageFolder('test',
transform=torchvision.transforms.Compose([
    torchvision.transforms.Grayscale(1),  # 单通道
    torchvision.transforms.ToTensor(),  # 将图片数据转成tensor格式
])))
```

构建网络并定义优化器、损失函数（这里暂时不讲解后续优化工作）：

```
cnn = CNN()
# optimizer
optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)
#optimizer = torch.optim.SGD(cnn.parameters(), lr = 0.01,
momentum=0.9,weight_decay=1e-5)#L2正则化

# loss_fun
loss_func = nn.CrossEntropyLoss()
```

循环训练网络，每跑完一个epoch就对测试集进行一次测试，并打印出准确率和损失误差：

```
# training loop
for epoch in range(EPOCH):
    for i, (x, y) in enumerate(train_loader):
        batch_x = Variable(x)
        batch_y = Variable(y)
        # 输入训练数据
        output = cnn(batch_x)
        # 计算误差
        loss = loss_func(output, batch_y)
        # 清空上一次梯度
        optimizer.zero_grad()
        # 误差反向传递
        loss.backward()
        # 优化器参数更新
        optimizer.step()

    print("Epoch " + str(epoch) + "/10")
    count = 0
    totalloss = 0
    for i, (x, y) in enumerate(test_x):
        batch_x = Variable(x)
        batch_y = Variable(y)
        output = cnn(batch_x)
        # 计算误差
        totalloss += loss_func(output, batch_y).item()
        index = torch.max(output, 1)[1].data.numpy().squeeze()
        if index == y.item():
            count += 1

    print("average loss: " + str(totalloss / len(test_x.dataset)))
    print("correctness: " + str(count / len(test_x.dataset)))
```

画出准确率和损失函数的图像：

```
# draw
plt.xlabel("epoch")
plt.ylabel("average loss")
plt.title('average loss')
plt.legend()
plt.plot(pltx, pltloss)
plt.show()

plt.xlabel("epoch")
plt.ylabel("accuracy")
plt.title('accuracy')
plt.plot(pltx, pltaccuracy)
plt.show()
```

初步测试最终训练准确率可达约98.5%，画出准确率和损失函数的图像如下：

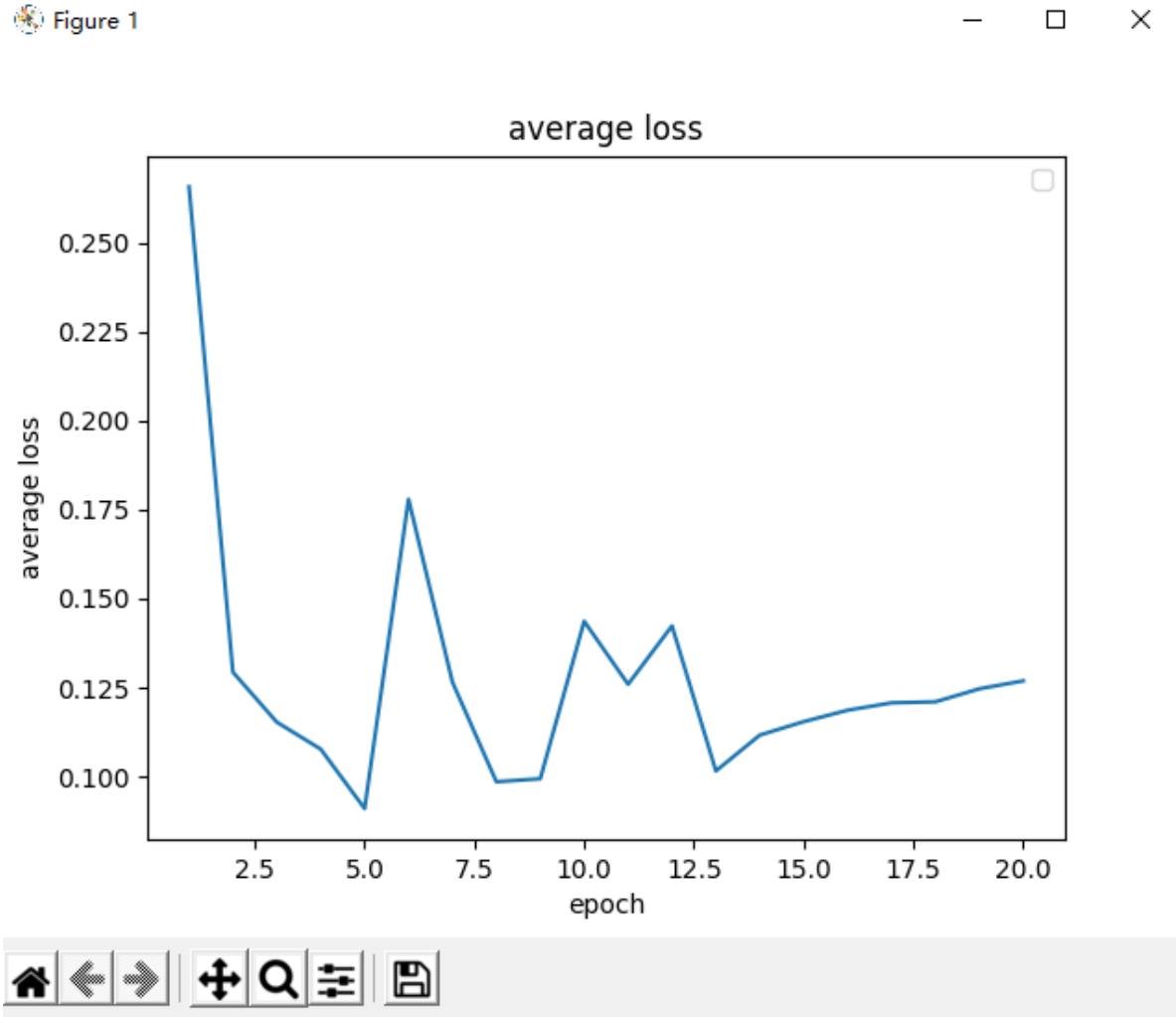


Figure 1
average loss
0.250
0.225
0.200
0.175
0.150
0.125
0.100
average loss
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
epoch

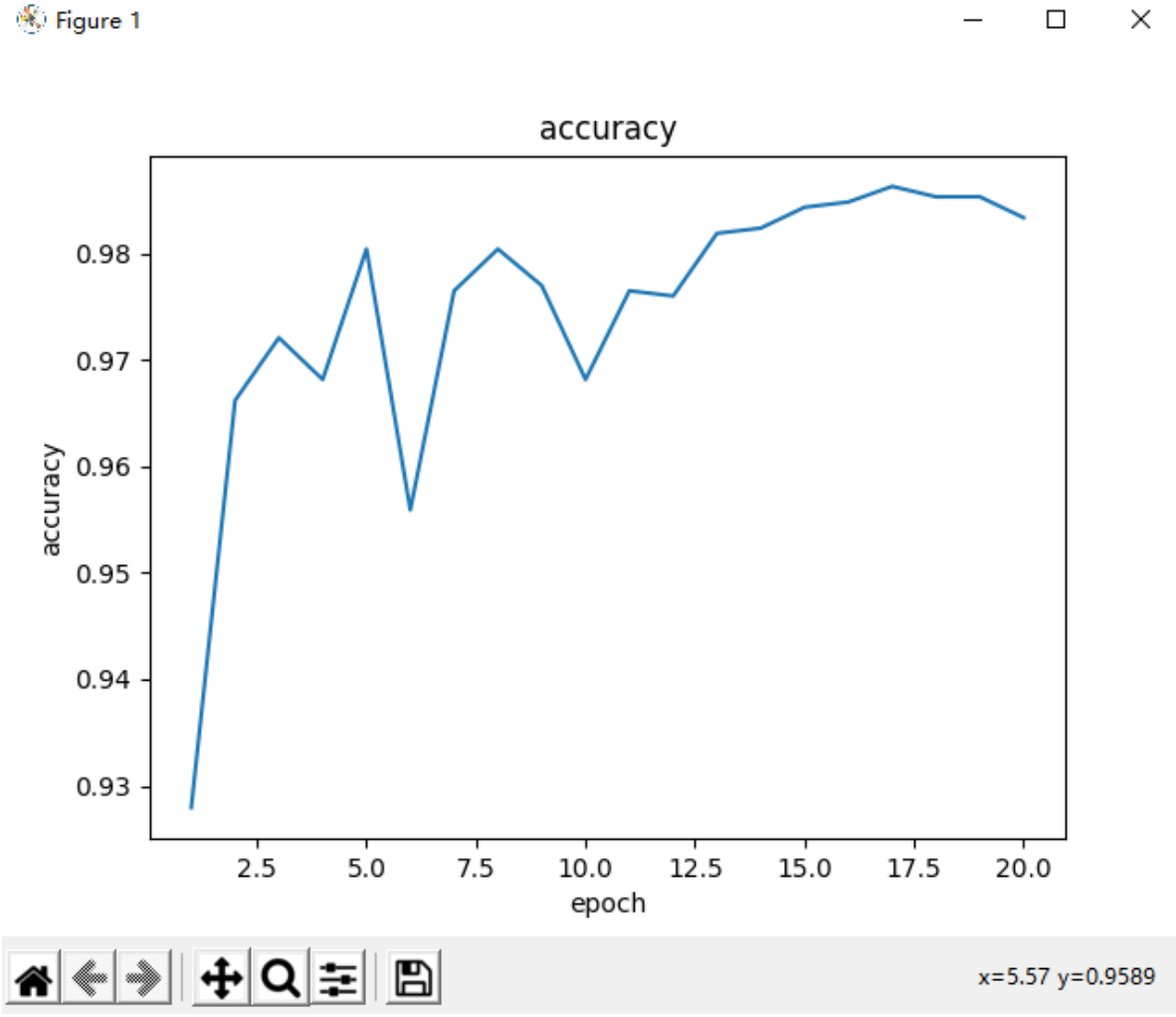


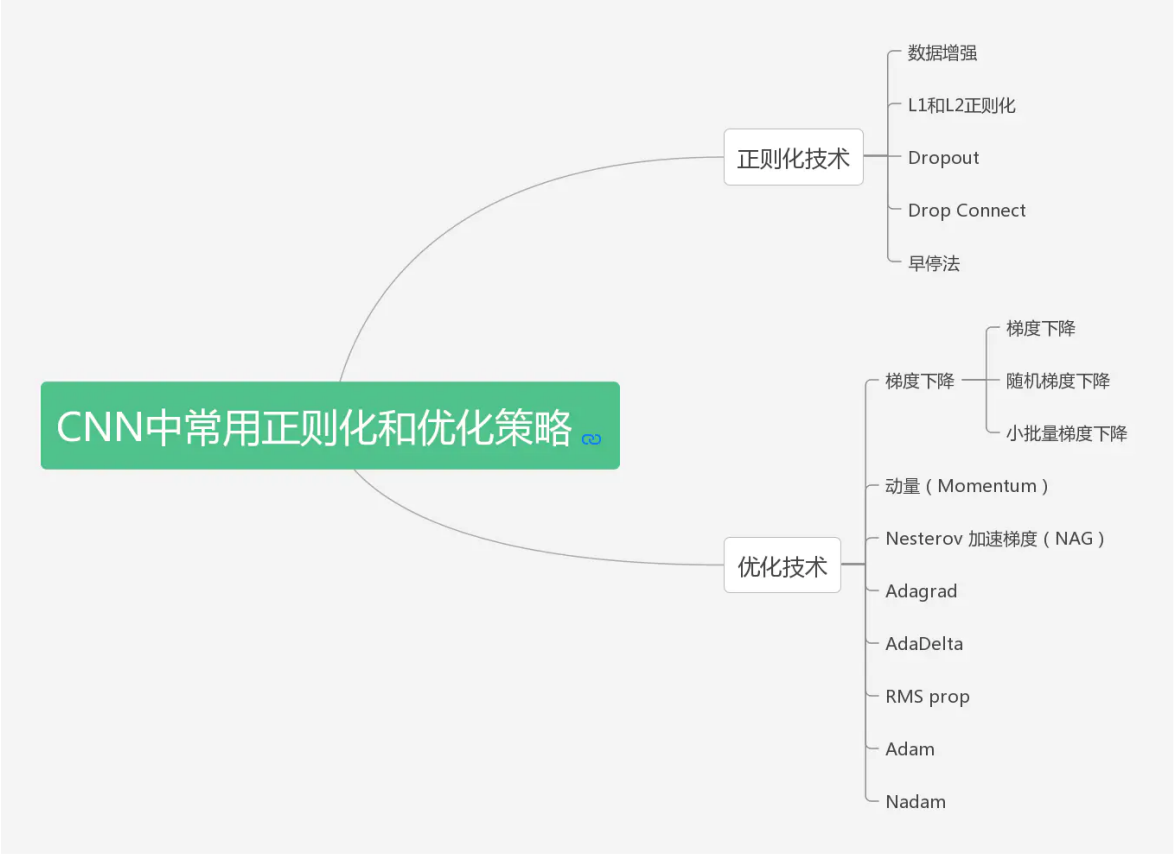
Figure 1
accuracy
0.98
0.97
0.96
0.95
0.94
0.93
accuracy
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
epoch
x=5.57 y=0.9589

可以看到当训练20个epoch时其实已经出现了轻微过拟合的现象，所以最终我把epoch定为15。

对网络的改进和论证

基于上述基本代码，我采用了*L1/L2正则化*、*Dropout*等方法改进网络，并且设计了实验对比使用各个方法的效果，并分析论证产生对应结果的原因。

总的来说我几乎使用了下图中所有改进策略：



（图片来源：https://www.jianshu.com/p/b410aeef9bce?utm_campaign）

L2正则化

当设置优化器时给它设置参数weight_decay即可：

```
#使用L2正则化
optimizer = torch.optim.Adam(cnn.parameters(), lr=LR, weight_decay=0.01)
```

epoch	采用L2正则化准确率	不采用L2正则化准确率
8	0.9789	0.9818
10	0.9853	0.9784
12	0.9637	0.9754
14	0.9799	0.9794

理论上L2正则化可以在backward时对模型的权重W参数进行惩罚，从而可以有效防止过拟合提高正确率，在我的对比试验中也可以看出采用L2正则化的平均正确率、最高正确率都比不采用高，但是也由于这种变化使得模型更不稳定，收敛得更慢，需要更多的epoch才能达到最大值，这在合理范围之内。

L1正则化

在每次计算损失时并不是简单的调用CrossEntropyLoss()，而是加上额外的正则项再进行backpropagate：

```
# 使用L1正则化
reg_loss = 0
for papam in cnn.parameters():
    reg_loss += torch.sum(torch.abs(papam))
classify_loss = loss_func(output, batch_y)
loss = classify_loss + 0.01 * reg_loss
```

epoch	采用L1正则化准确率	不采用L1正则化准确率
8	0.9368	0.9818
10	0.9343	0.9784
12	0.9402	0.9754
14	0.9299	0.9794

L1正则化思路是在每次更新权重时都加上一项$\sum W^2$，但是也许是因为本次十二分类任务过于简单，样本量也不是很大，因此网络结构简单参数不多，并未产生过拟合的现象，所以使用了L1正则化以后，准确率反而下降了，故这一方案最终并未被采用。

Dropout

构建cnn网络时在全连接层之前加入一个Dropout层：

```
nn.Dropout()#drop out
```

epoch	采用Dropout准确率	不采用Dropout准确率
8	0.9770	0.9818
10	0.9764	0.9784
12	0.9818	0.9754
14	0.9843	0.9794

在全连接层之前Dropout层，即在训练过程中以一定概率P（默认是0.5，这里我采用0.5），将隐含层节点的输出值清0，而用反向传播算法更新权值时，不再更新与该节点相连的权值。这种方法也会降低训练的收敛速度，但是在我的网络模型中确实能够提升正确率，因此最终我采用了这一正则化方法，并且设置Epoch数量为20。

早停法

只需要在每一个epoch训练完成以后，将这一次的准确率（或者损失）和上一个epoch的准确率（或者损失），当发现准确率下降时马上停止训练，也能达到防治过拟合的效果：

```
if count / len(test_x.dataset) <= lastaccuracy:
    break
else:
    lastaccuracy = count / len(test_x.dataset)
```

由于采用早停法，一部分测试集已经被用于检测是否过拟合，所以需要重新分割数据集，测试数据和测试指标与之前有所不同：

	采用早停法	不采用早停法
训练所需epoch数	8	15
准确率	0.9818	0.9811

早停法在训练中计算模型在验证集上的表现，当模型在验证集上的表现开始下降的时候，停止训练，这样就能避免继续训练导致过拟合的问题。在本次lab中由于数据集完全给出，因此采用早停法时确实可以带来优化，所以我在开发时保留了早停法，当然当面试时预测数据序列就变得不可用了。

Batch Normalization

构建cnn网络时不是简单的卷积层-ReLU层-卷积层的形式，而是在卷积层之后加入BatchNorm2d层：

```
nn.BatchNorm2d(16),#batch normalization
```

epoch	采用Batch Normalization准确率	不采用Batch Normalization准确率
8	0.9818	0.9818
10	0.9784	0.9784
12	0.9752	0.9754
14	0.9794	0.9794

BN进行数据归一化，引入了两个伸缩变量，最大的优点应该是加快训练速度，还能解决梯度消失和梯度爆炸的问题，但是可能是由于问题过于简单，在我的模型中加入BN完全不影响准确率，也不能使收敛所需的epoch数减少，但是最终还是保留了这一优化。

不同的优化函数

尝试了随机梯度下降和小批量梯度下降等很多不同的优化函数，但是似乎并不能显著改善准确率（准确率几乎完全不变，因此没必要贴出对比表格了），只在下面展示代码：

```
# optimizer = torch.optim.Adam(cnn.parameters(), lr=LR, weight_decay=1e-5)#L2正则化
# optimizer = torch.optim.SGD(cnn.parameters(), lr = LR)#随机梯度下降
# optimizer = torch.optim.ASGD(params, lr=0.01, lambd=0.0001, alpha=0.75, t0=1000000.0, weight_decay=0)#平均随机梯度下降算法
# optimizer = torch.optim.Adagrad(params, lr=0.01, lr_decay=0, weight_decay=0)#AdaGrad算法
# optimizer = torch.optim.Adadelta(params, lr=1.0, rho=0.9, eps=1e-06, weight_decay=0)#自适应学习率调整 Adadelta算法
# optimizer = torch.optim.RMSprop(params, lr=0.01, alpha=0.99, eps=1e-08, weight_decay=0, momentum=0, centered=False)#RMSprop算法
# optimizer = torch.optim.Adamax(params, lr=0.002, betas=(0.9, 0.999), eps=1e-08, weight_decay=0)#Adamax算法（Adamd的无穷范数变种
# optimizer = torch.optim.SparseAdam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08)#SparseAdam算法
# optimizer = torch.optim.LBFGS(params, lr=1, max_iter=20, max_eval=None, tolerance_grad=1e-05, tolerance_change=1e-09, history_size=100, line_search_fn=None)#L-BFGS算法
# optimizer = torch.optim.Rprop(params, lr=0.01, etas=(0.5, 1.2), step_sizes=(1e-06, 50))#弹性反向传播算法
```

其他超参数的调整

与Lab1Part1相同，对模型的层数、隐层节点数、学习率、Epoch数、Batch size都进行了改变和效果的比较，过程十分繁琐，并且几乎与Lab1Part1的内容一样，面试时我可以提供Lab1Part1的文档作为调试记录，在此不再赘述，仅仅表明最终以下超参数效果最好：

```
# Hyper Parameters
EPOCH = 20
BATCH_SIZE = 5
LR = 0.001
```

对CNN网络的理解

Why CNN?

在Lab1Part1中其实我们已经手写了一个bp网络，并且实现了对这十二个汉字的分类，那么为什么要使用CNN再来一次呢?

参数数量太多。全连接的神经网络，在上一层所有节点和下一层所有节点之间，都有连接，导致参数数量会以每层节点数量的积的形式增长，这对于图片分类来说，实在过于庞大，因为每张图片都会有几百个像素点甚至更多的输入。CNN通过局部连接的卷积、同一个局部使用相同的参数、pooling解决了这一点。

没有利用像素之间的位置信息。在全连接神经网络中似乎同一层的每个节点都是对等的——因为他们无差别的连接着上一层和下一层的所有节点，但是对于图像来讲通常一个像素点与它周围的像素点关系紧密，CNN通过局部连接的卷积形式解决了这一点。

网络层数限制。全连接神经网络由于梯度下降法的限制，层数很难很多，因为存在梯度消失的问题，而CNN则通过多重手段例如ReLU解决了这一问题。

What is CNN?

卷积层、ReLU层（其实是过一个激活函数）、Pooling层，这样的三个结构构成一个更大的“层”，重复这个层若干次，最后再经过一个全连接层，就是基本的卷积神经网络结构。

卷积层使用"Window"的方法使得上一层庞大的节点数只和下一层的部分节点相连，从而提取出这一小块区域的特征；ReLU层其实是一个激活函数，这个函数的形式能够解决梯度消失的问题；Pooling层进一步对上次得到的Feature Map进行采样，从而特征更加浓缩，参数量更少。

虽然具体表达形式不同，但是其实训练原理还是一样的，还是不断重复forward和bp的过程，那么CNN为什么能达到效果也就是明了的了：还是在不断重复犯错误-纠正-减小误差的过程，以使得找到一个合适的参数网络使得不同的图片输入映射到对应的取值空间。