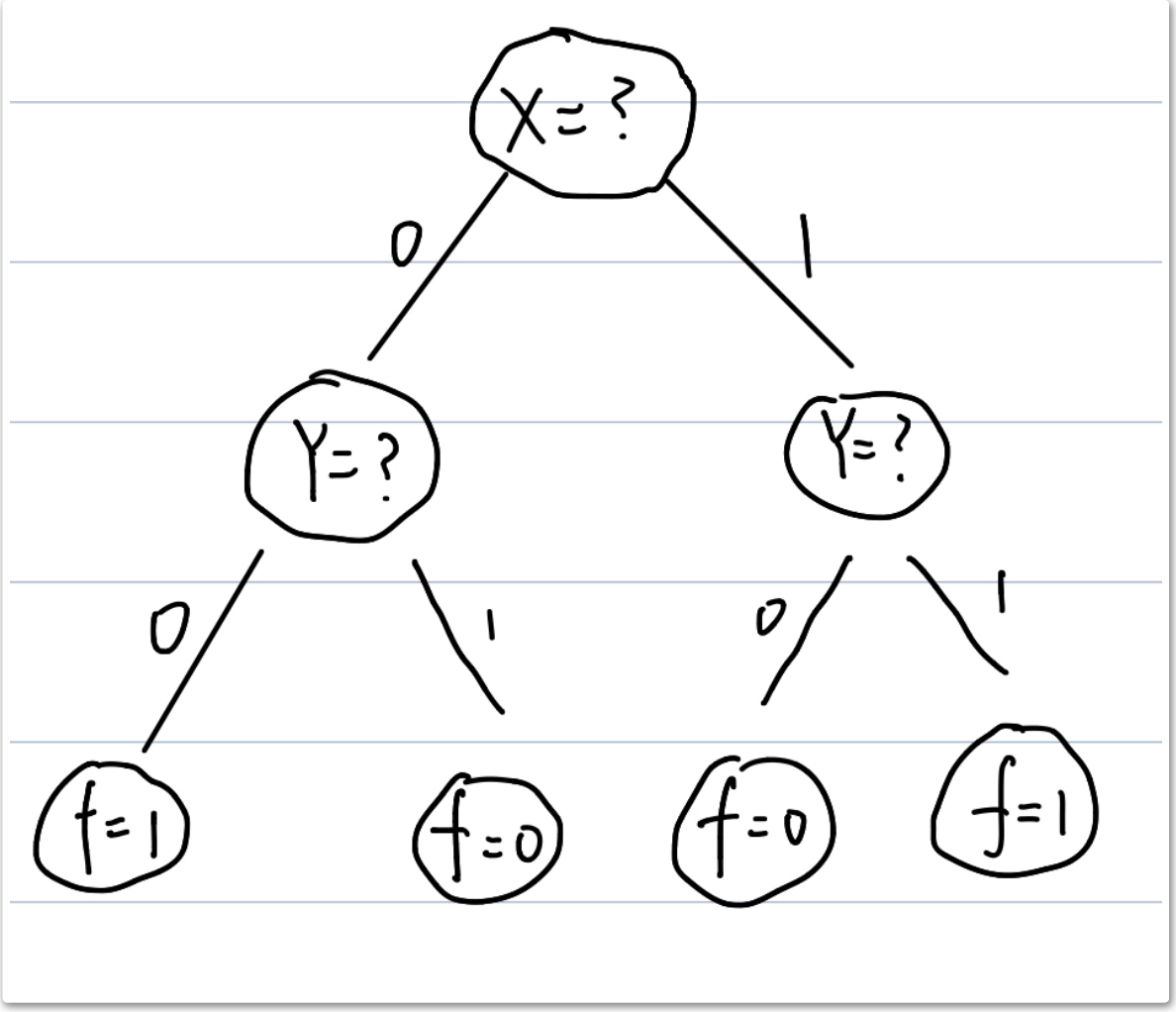
机器学习第三次作业

181860107 王梓涛

1.Decision Tree

（1）是可实现的。

构造的决策树如下图所示：



（2）

1.根节点的信息熵定义为

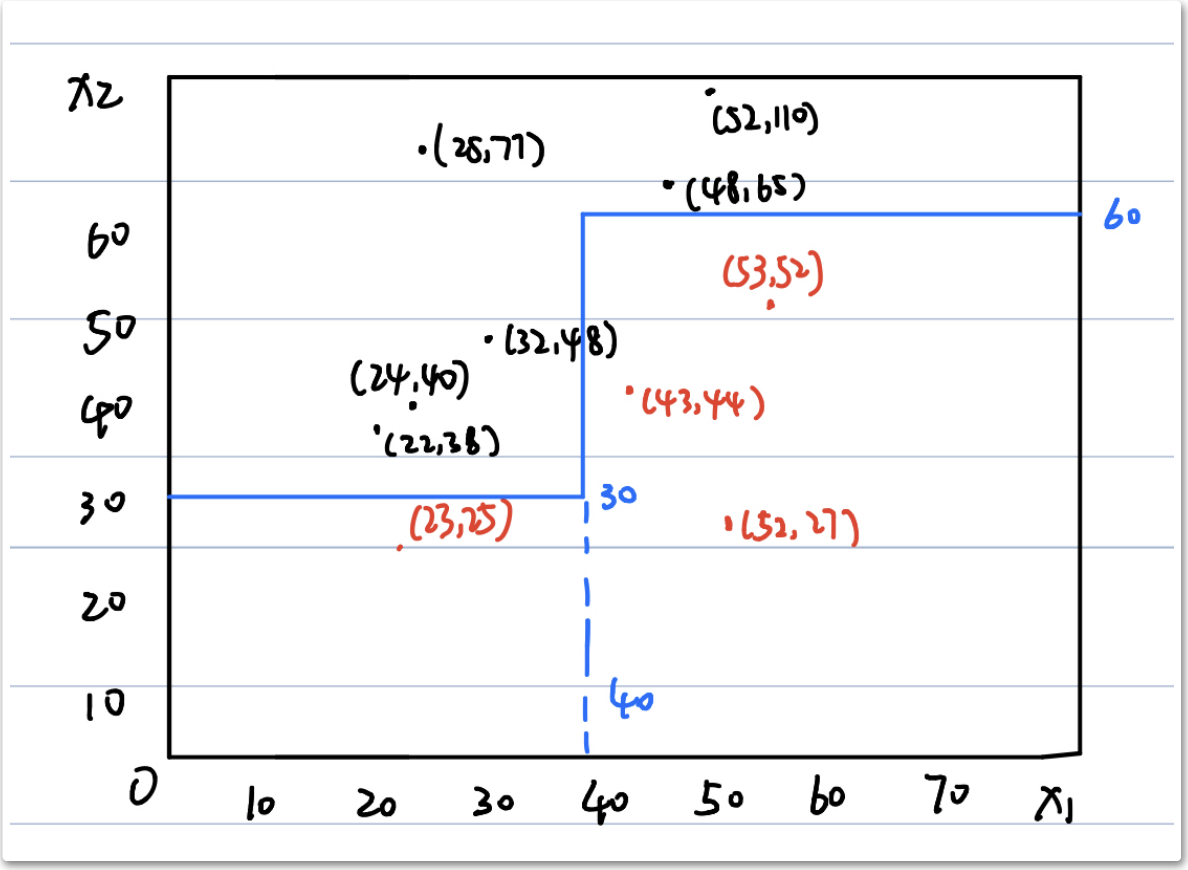
$$Ent(D) = -\Sigma_{k=1}^{|\gamma|} p_k log_2 p_k$$

显然只有两类样本x1和x2，一类所占比例为0.6(正例)，一类为0.4(反例)。

所以根节点的信息熵为：

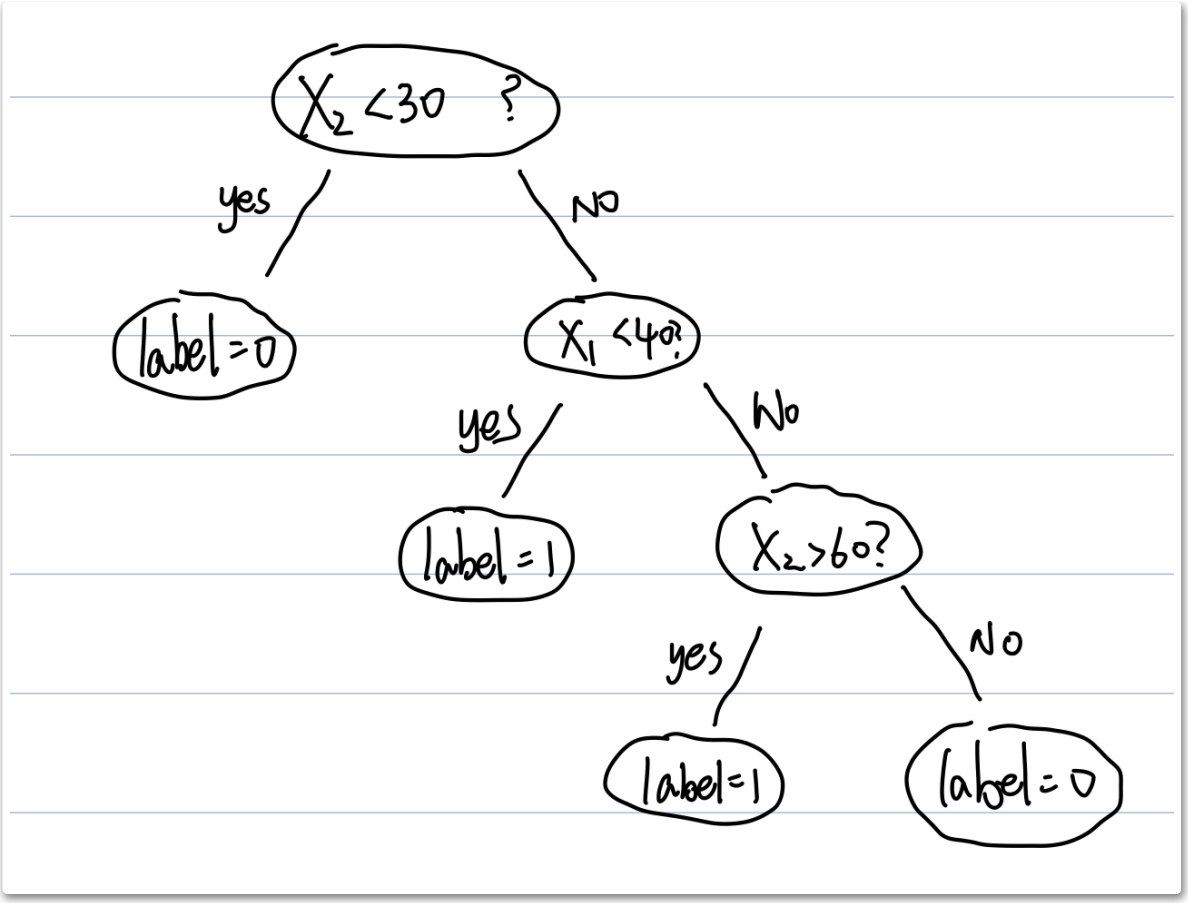
$$Ent(D) = -\Sigma_{k=1}^{2} p_k log_2 p_k = -(0.6log_2 0.6 + 0.4log_2 0.4) = 0.97095$$

2.这道题的样本x1，x2分类有点特殊，用一个横坐标为x1，纵坐标为x2的散点图来进行划分说明：



从中可以看到，以蓝线为界，蓝线下方的全是反例，蓝线上方的全是正例。因此可以以蓝线作为分裂规则：

x2值小于30的为负例，大于30的再看x1，若x1小于40则为正例，否则再看x2，若x2小于60则为负例，否则为正例。由此得到的决策树：



最后经检验易得分类误差为0。

2.Neural Network

(1)重新构造一个无隐层的线性神经网络，连接输入x1和输出y，他们的新权值是：

$$w1 * w5 + w2 * w6$$

再连接输入x2和输出y，他们的新权值是：

$$w3 * w5 + w4 * w6$$

此时新的无隐层的线性神经网络的激活函数功能与原先一样。

(2)是的。证明如下：

在线性的神经网络中，每一层都可以看作从输入通过一个矩阵乘法得到一个该层的输出。因此，整个神经网络从输入到输出可以通过不断的进行矩阵相乘（类似于一个矩阵链）得到，同时这些矩阵相乘得到的就是新神经网络的权值。因此一个有隐层的线性神经网络可以表示成一个无隐层的神经网络。

(3)与输入层x1和x2相连的两个神经元采用线性函数作为激活函数，与输出层Y相连的神经元采用对数几率函数作为激活函数。

3.Neural Network in Practice

卷积神经网络(CNN)简介：

卷积神经网络（Convolutional Neural Network, CNN）是一种前馈神经网络，它的人工神经元可以响应一部分覆盖范围内的周围单元，对于大型图像处理有出色表现。CNN通常包含以下几种层：卷积层，线性整流层，池化层，全连接层。其中每层的卷积层由若干个卷积单元构成，卷积运算的目的是提取输入的不同特征。线性整流层的激活函数采用ReLU函数。池化层用于在得到卷积层的大维度特征后，将特征切成几个区域，取其中最大值或者平均值，以得到新的小维度特征。全连接层用来将局部特征结合转化为全局特征，用来计算最后的得分。

(1)

本次实验用python3完成了3个不同网络结构的CNN，包括一个LeNet5，一个AlexNet，还有一个普通的CNN，分别命名为文件LeNet5.py，AlexNet.py和CNN_main.py。同时对AlexNet采用了3种优化算法进行分析比较，分别是Adam，SGD和SGD+momentum。下面以CNN_main.py文件为例介绍实验代码和实验过程：

实现需要下载实验的数据集，可以使用torchvisio中的datasets进行下载，非常的方便，其中参数root指定了存放路径，transform指定了下载数据集时作何种变换，train表示是否为训练集。

在transform中的两个参数0.1307和0.3081分别为MNIST提供的均值和标准差。

```
    Transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize(mean=[0.1307,],std=[0.3081,])])
    #下载训练集，测试集
    trainSet =
datasets.MNIST(root="./data",transform=Transform,train=True,download=True)
    testSet = datasets.MNIST(root="./data",transform=Transform,train=False)
```

下载完之后需要装载数据集，其中batch_size设置了每批装载的数据图片数，shuffle为true表示随机装载：

```
#装载训练集，测试集，一个batch数据集64张图
    trainLoader = torch.utils.data.DataLoader(dataset=trainSet,batch_size =
64,shuffle = True,num_workers=2)
    testLoader = torch.utils.data.DataLoader(dataset=testSet,batch_size =
64,shuffle = True,num_workers=2)
```

接下来就是构建神经网络，这里构建了一个包含两个卷积层，一个池化层，若干个线性整流层和一个

全连接层，其中卷积层使用nn.Conv2d来构建，池化层由nn.MaxPool2d来构建，线性整流层由nn.ReLU来构建，全连接层用nn.Linear来构建。还使用了一个nn.Dropout函数来防止模型过拟合。再使用一个forward函数定义前向传播，经过卷积，池化处理后，使用x.view进行扁平化处理再通过全连接层进行分类。

```
class normalCNN(nn.Module):
    #网络结构：卷积层，激活函数，池化层，全连接层
    def __init__(self):
        super(normalCNN,self).__init__()
        self.conv1 = nn.Conv2d(1,64,kernel_size=3,stride=1,padding=1)
        self.conv2 = nn.Conv2d(64,128,kernel_size=3,stride=1,padding=1)
        self.pool = nn.MaxPool2d(stride=2,kernel_size=2)
        self.fc1 = nn.Linear(14*14*128,1024)
        self.fc2 = nn.Linear(1024, 10)
        self.drop = nn.Dropout(p=0.5)
        self.relu = nn.ReLU()
    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool(x)
        x = x.view(-1,14*14*128)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.drop(x)
        x = self.fc2(x)
        return x
```

之后调用normolCNN对象，使用交叉熵计算Loss，优化算法选取Adam算法。

```
#损失函数使用交叉熵
cost = nn.CrossEntropyLoss()
#优化计算方式选择：
optimizer = torch.optim.Adam(model.parameters(),lr=1e-3,betas=(0.9,0.99))
epochSize = 20
```

然后就可以开始训练测试了，epoch次数选取20次，每一次epoch中进行训练和测试：

```
for epoch in range(epochSize):
    runLoss=0.0
    trainCorrect=0
    print("Epoch {}/{}".format(epoch,epochSize-1))
    print("----------")
    #training:
    for trainData in trainLoader:
        ……
    #testing:
    testCorrect=0
    for testData in testLoader:
        ……
```

在每批64张图的训练中，先提取出inputs特征和labels标签，然后使用模型训练出outputs，再舍弃掉数据大于1的部分，然后初始化优化器梯度，使用交叉熵损失函数计算损失，再反向传播梯度，最后更新参数。

```
        inputs,labels = trainData
        inputs,labels = inputs.to(device),labels.to(device)
        outputs = model(inputs)
        _,pred = torch.max(outputs.data, 1)
        optimizer.zero_grad()
        loss = cost(outputs, labels)
        loss.backward()
        optimizer.step()
        runLoss += loss.item()
        trainCorrect += torch.sum(pred == labels.data)
```

在使用测试集测试数据就更简单了：

```
            inputs,labels = testData
            inputs,labels = inputs.to(device),labels.to(device)
            outputs = model(inputs)
            loss = cost(outputs, labels)
            testLoss += loss.item()
            _,pred = torch.max(outputs.data, 1)
            testCorrect += torch.sum(pred == labels.data)
```

最后打印每个epoch是训练测试结果：

```
print("Train Accuracy is {:.4f}%,Test Accuracy is {:.4f}%,Loss is
{:.4f}".format(trainCorrect*100/trainDataLen,testCorrect*100/testDataLen,runLoss
/trainDataLen))
```

还添加了一个将训练，测试损失率和测试准确率通过折线图的形式表现出来的功能：

```
    plt.figure(1)
    x1=range(0,epochSize)
    x2=range(0,epochSize)
    x3=range(0,epochSize)
    y1=accuracy_list
    y2=runLoss_list
    y3=testLoss_list
    plt.plot(x1,y1,'o-')
    plt.title('Test accuracy in epoches')
    plt.ylabel('Test accuracy')
    plt.figure(2)
    plt.subplot(2, 1, 1)
    plt.plot(x2, y2, '.-')
    plt.title('Train and Test Loss in epoches')
    plt.ylabel('Train loss')
    plt.subplot(2, 1, 2)
    plt.plot(x3, y3, '.-')
    plt.xlabel('epoches')
    plt.ylabel('Test loss')
    plt.show()
```

还可以使用gpu进行加速，如果没有配置cuda环境则仍使用cpu:

```
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    print(device)
```

(2)

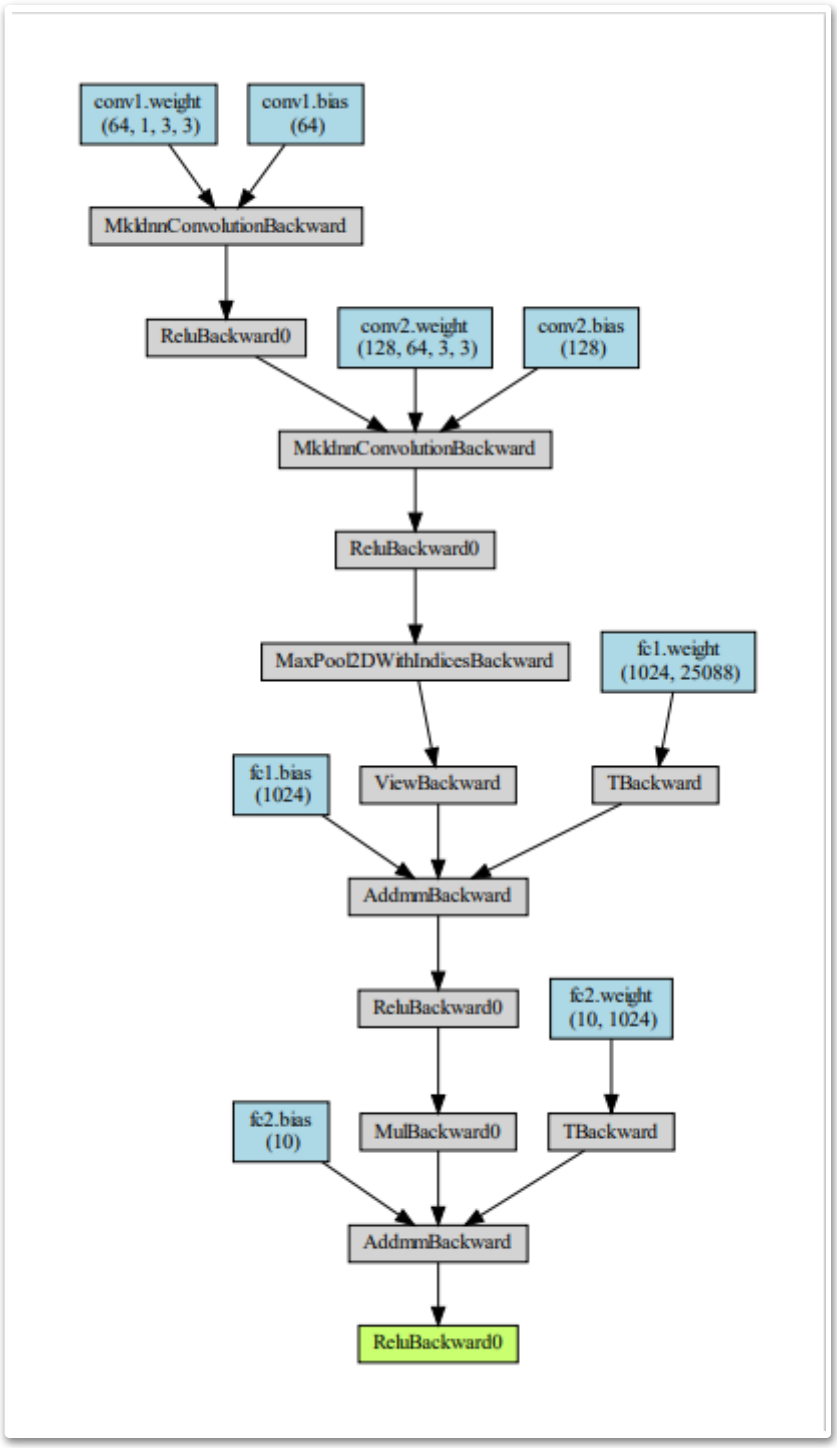
一共使用了3种CNN模型，分别是普通的CNN模型，AlexNet模型和LeNet5模型。

普通的CNN模型：

模型代码已经在(1)中给出，绘制模型网络结构可以使用以下代码：

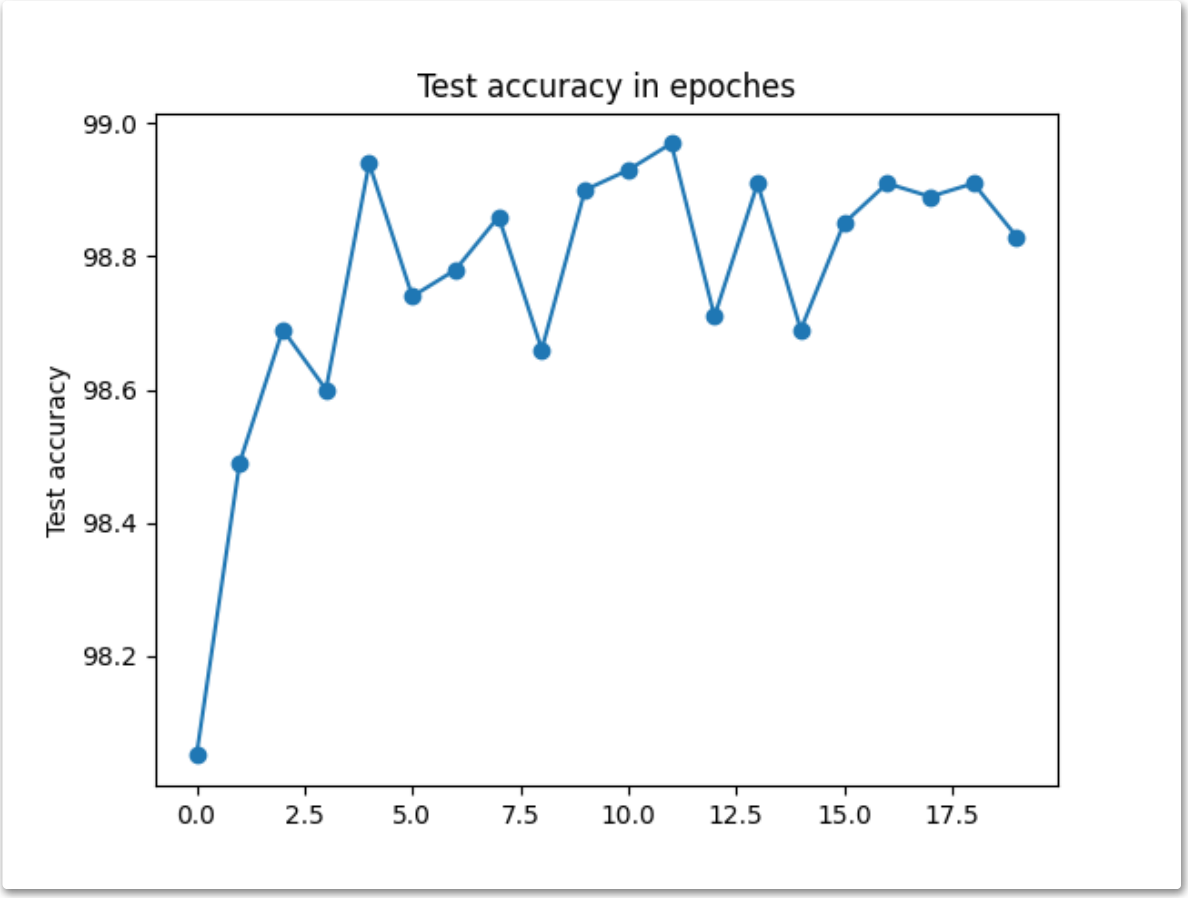
```
g=make_dot(model(torch.rand(64,1,28,28)),params=dict(model.named_parameters()))g
.view()
```

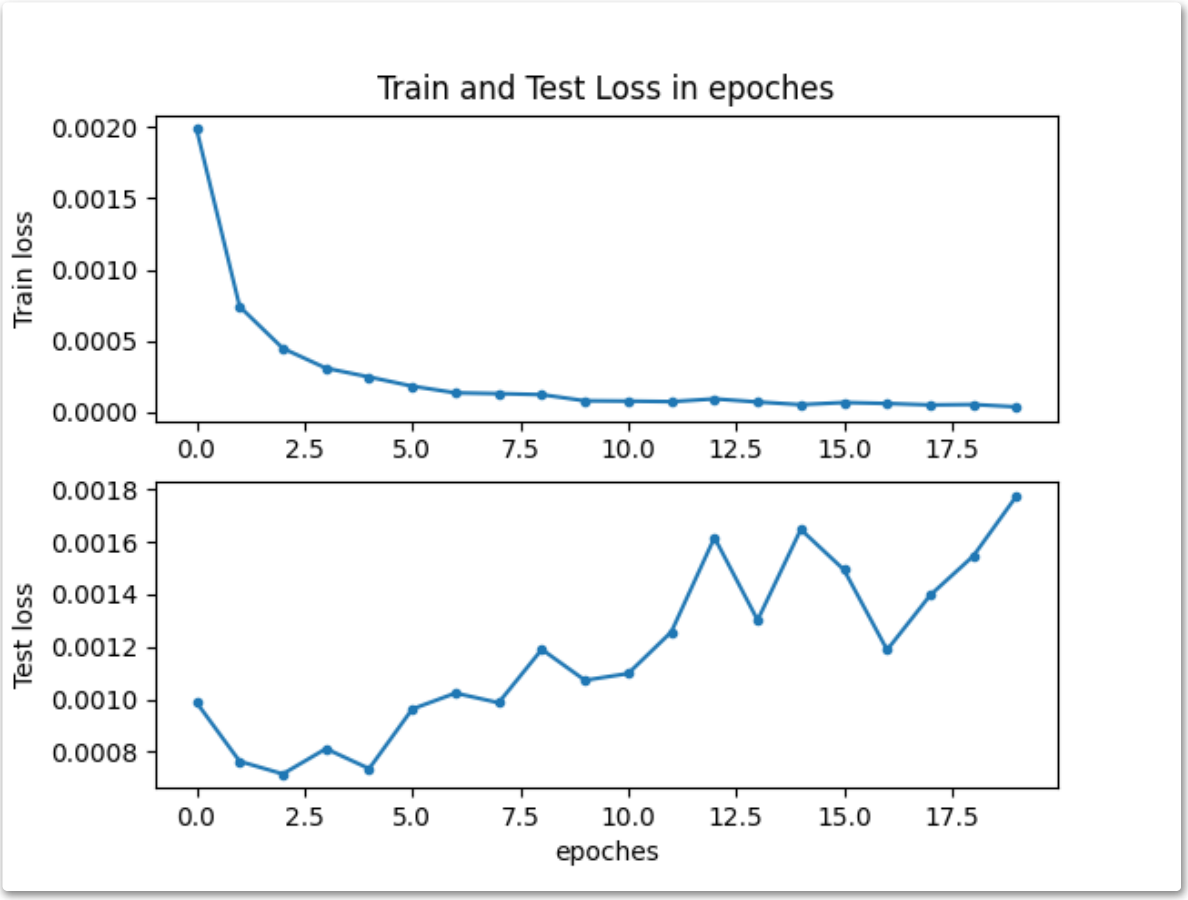
绘制结果（图片过大，建议查看附件的normalCNNStruct.pdf文件）：



模型训练测试结果：

normalCNN+Adam:





每轮epoch输出的具体内容，包含训练正确率，测试正确率，训练损失，测试损失详见附件./image/NormalCNN+Adam.txt

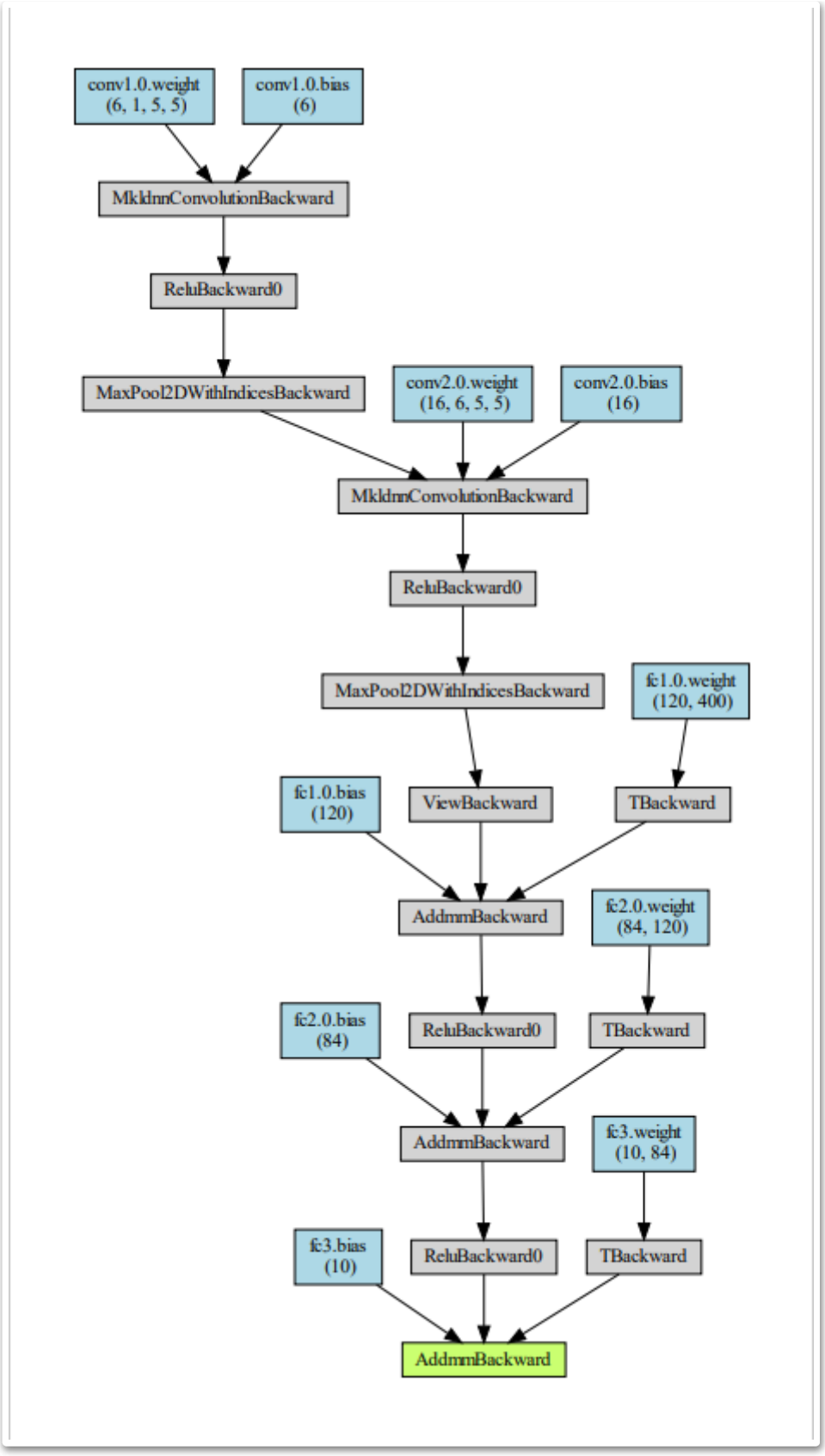
LeNet5模型：

在1998年提出的一个简单的CNN结构，有着2个卷积层和3个全连接层，期间还穿插着2个池化层。

网络结构代码如下：

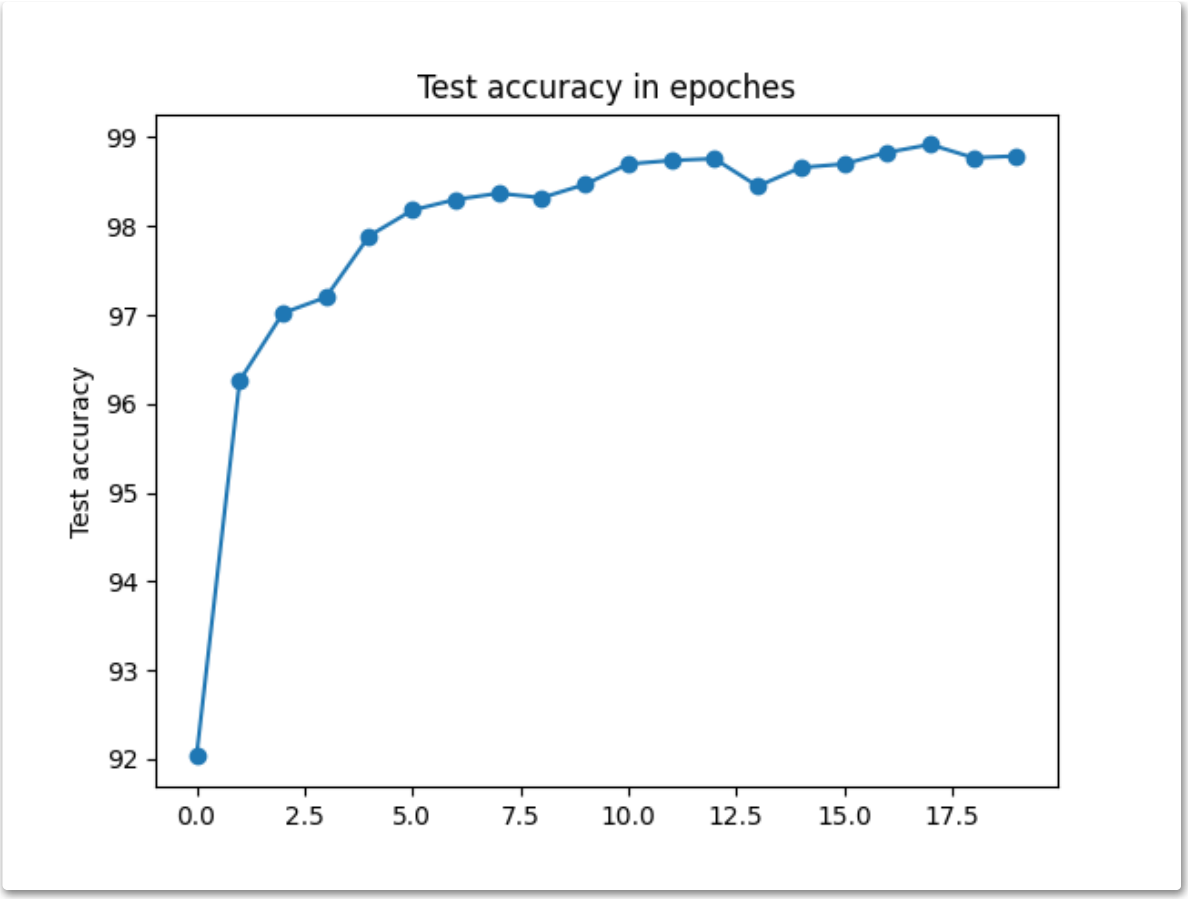
```
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5,self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(1,6,5,1,2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(6,16,5),
            nn.ReLU(),
            nn.MaxPool2d(2,2)
        )
        self.fc1 = nn.Sequential(
            nn.Linear(16*5*5,120),
            nn.ReLU()
        )
        self.fc2 = nn.Sequential(
            nn.Linear(120,84),
            nn.ReLU()
        )
        self.fc3 = nn.Linear(84,10)

    def forward(self,x):
        x=self.conv1(x)
        x=self.conv2(x)
        x=x.view(x.size()[0],-1)
        x=self.fc1(x)
        x=self.fc2(x)
        x=self.fc3(x)
        return x
```

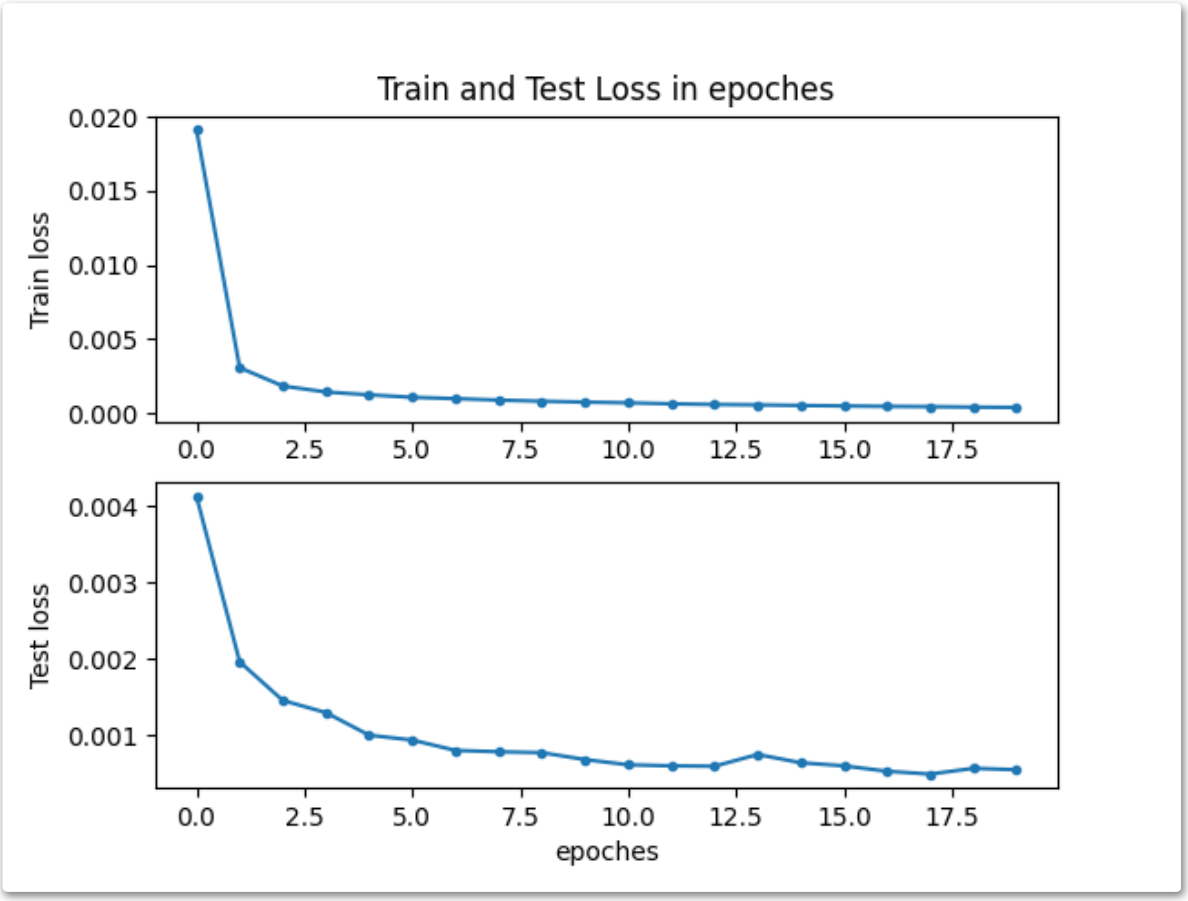
模型网络结构（图片过大，建议查看附件的LeNet5Struct.pdf文件）：



模型训练结果：

LeNet5+SGD+momentum:





每一步的具体结果见附件./image/LeNet5+SGD+momentum.txt.

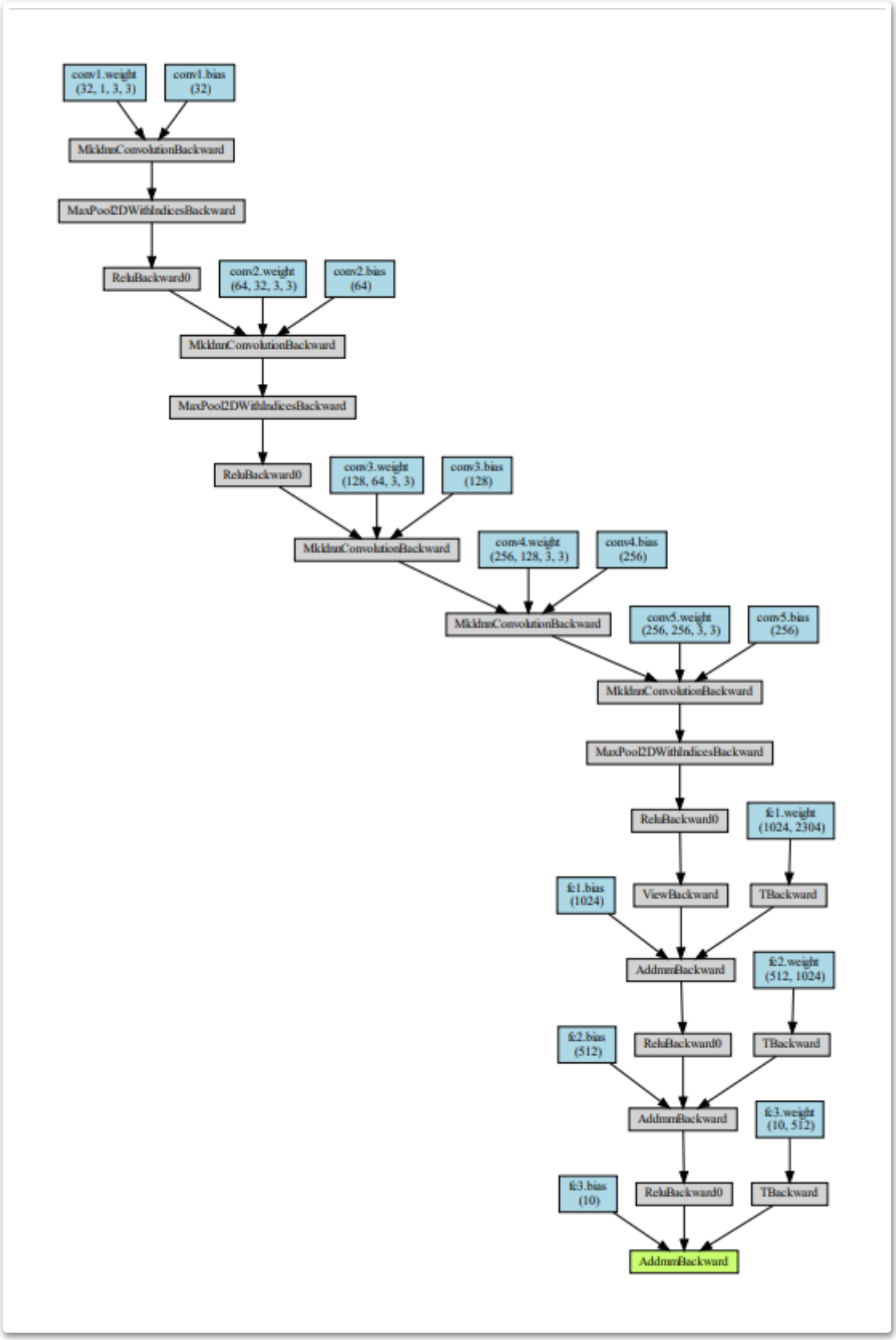
AlexNet模型:

AlexNet模型是由Alex在2012年提出的一种CNN网络结构，共有8层，其中5个卷积层和3个全连接层，还有3个池化层。

网络模型结构代码如下：

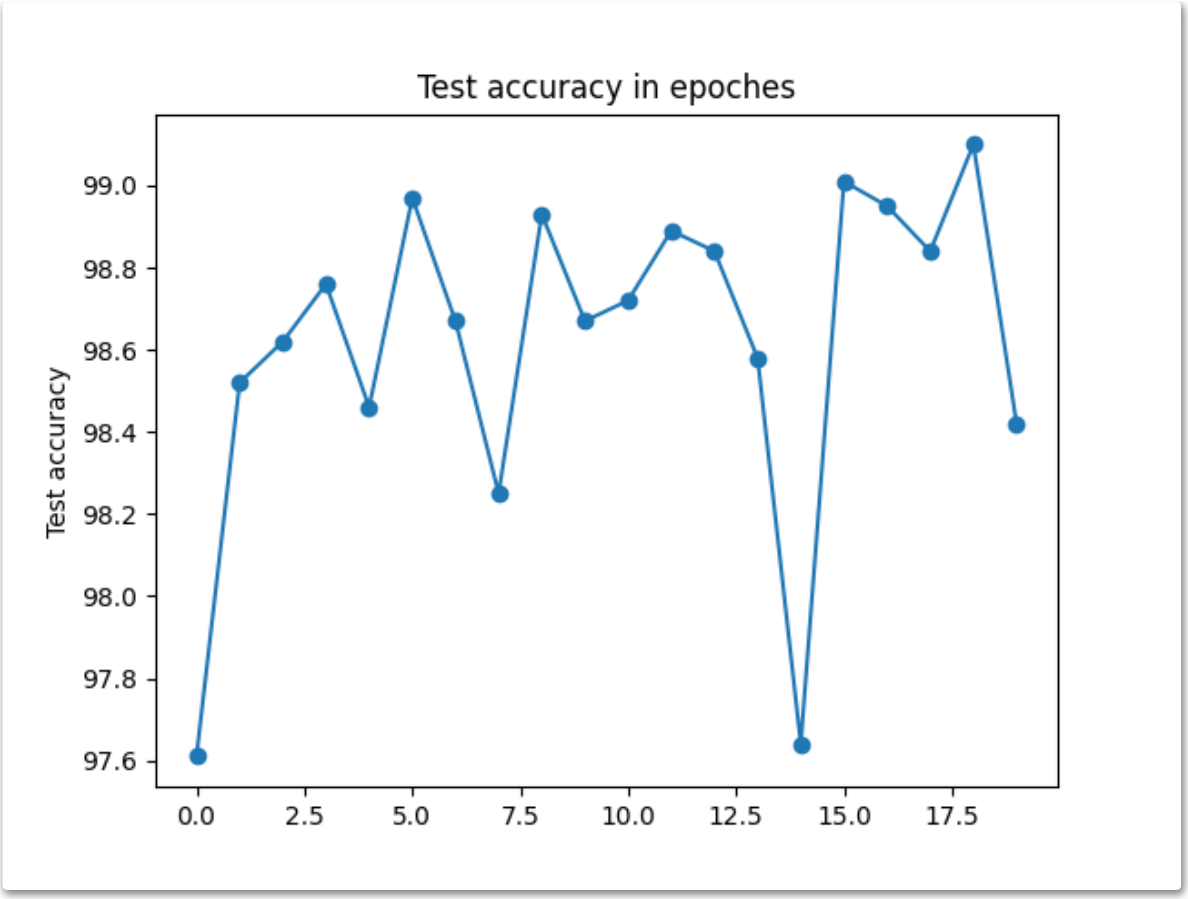
```
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet,self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.relu = nn.ReLU()

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2,stride=2)



        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.conv5 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)



        self.fc1 = nn.Linear(256*3*3, 1024)
        self.fc2 = nn.Linear(1024, 512)
        self.fc3 = nn.Linear(512, 10)

    def forward(self,x):
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = self.relu(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = self.pool3(x)
        x = self.relu(x)
        x = x.view(-1, 256 * 3 * 3)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x
```

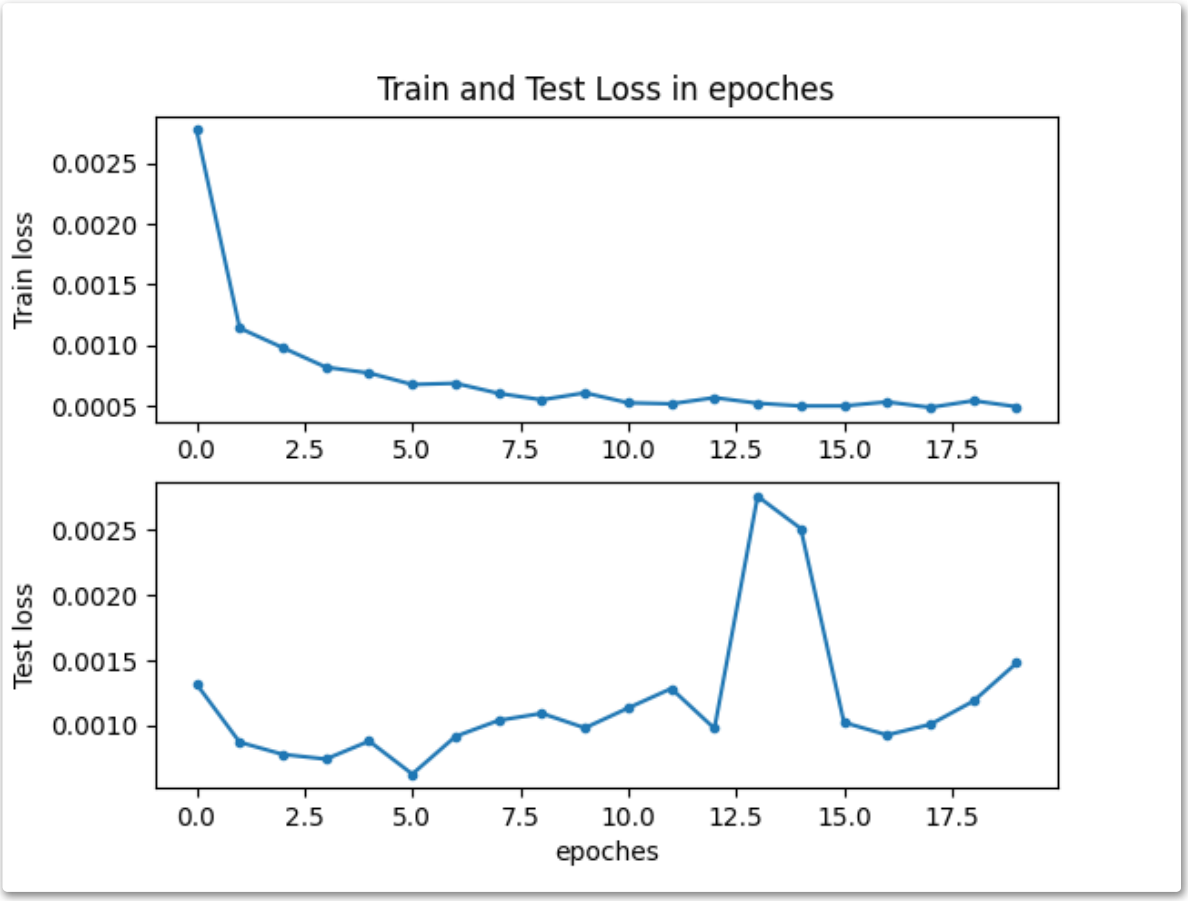
模型网络结构（图片过大，建议查看附件的AlexNetStruct.pdf文件）：



模型训练结果：

AlexNet+Adam:





每一步的具体结果见附件./image/AlexNet+Adam.txt.

3种不同的网络结构训练测试结果对比：

Accuracy(%)	normalCNN	AlexNet	LeNet5
epoch=0	98.0500	97.6100	92.0300
epoch=1	98.4900	98.5200	96.2600
epoch=2	98.6900	98.6200	97.0200
epoch=3	98.6000	98.7600	97.2000
epoch=4	98.9400	98.4600	97.8900
epoch=9	98.9000	98.6700	98.4700
epoch=14	98.6900	97.6400	98.6600
epoch=19	98.8300	98.4200	98.7900

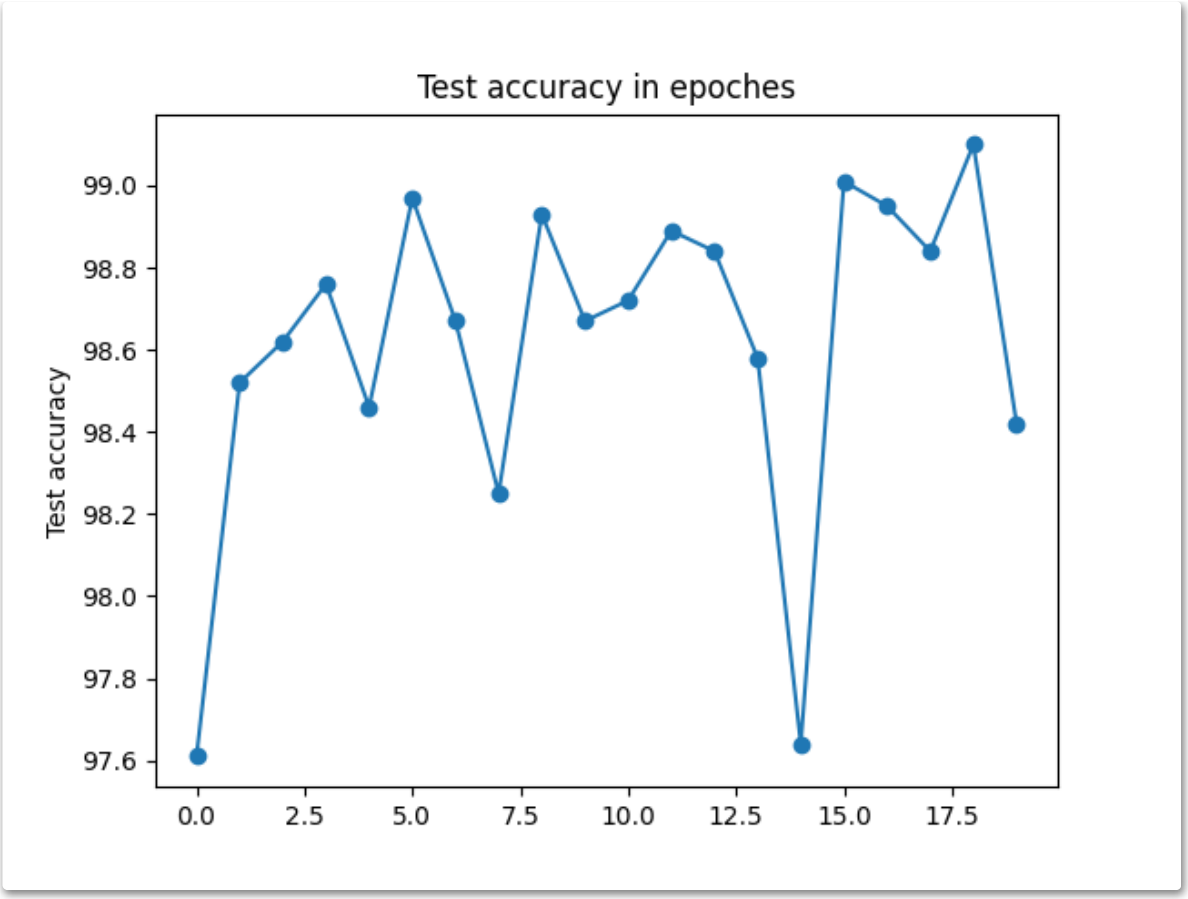
首先比较三个优化算法的loss曲线变化情况，可以看出AlexNet模型和normalCNN模型的trainLoss曲线早期一直下降，而同期的testLoss曲线趋于不变，可能是模型过拟合了，而后期trainLoss和testLoss都趋于不变，训练应该是快到瓶颈了。而LeNet模型早期trainLoss曲线和testLoss曲线一起下降，说明正在学习，后期下降速度变慢，趋于不变，是训练到达瓶颈。

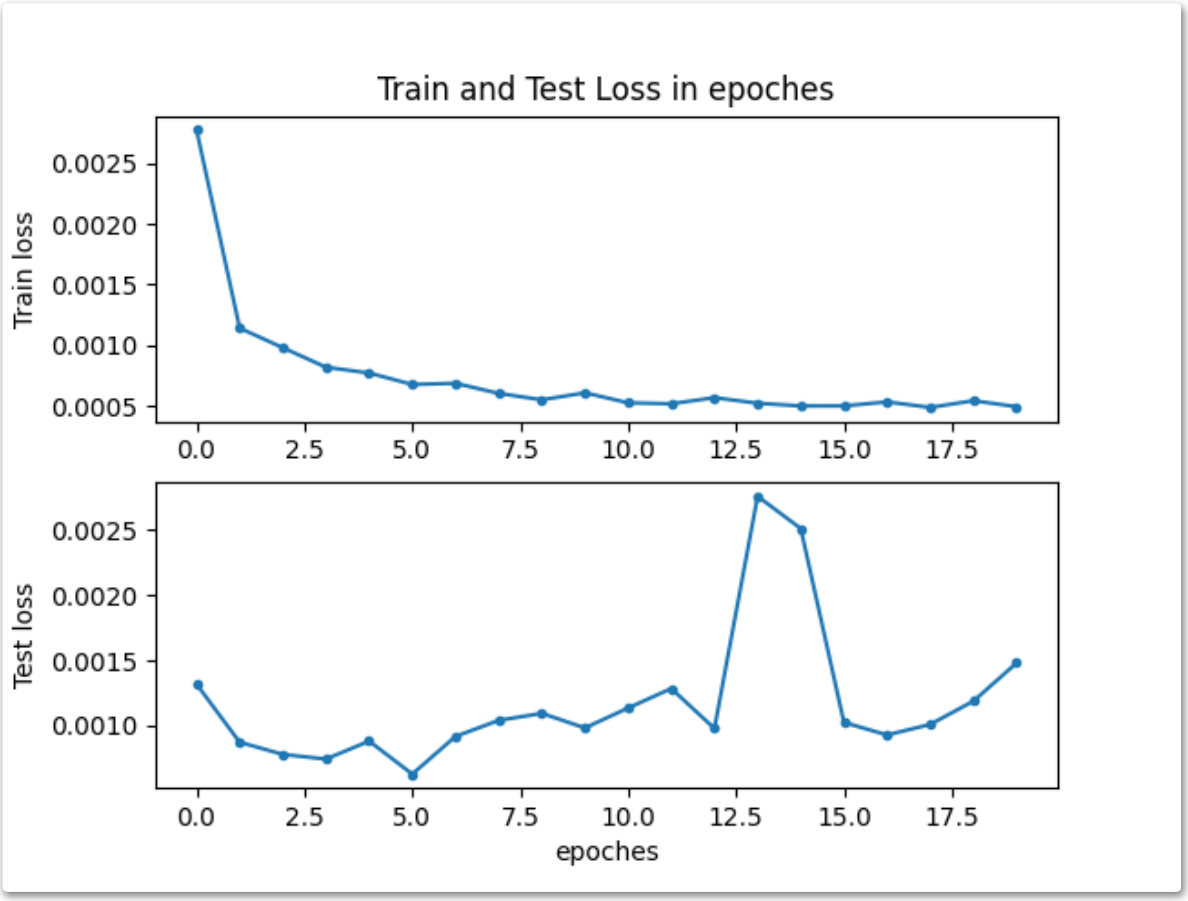
由表可知，模型normalCNN和AlexNet可以较快的拥有较高的测试正确率（97.5%以上），在epoch=0就能实现了，而LeNet5模型需要更多次epoch才能达到97.5%以上的正确率，在epoch=4才能实现，但三个模型最后正确率相差不大，都在98.5%左右波动，normalCNN和AlexNet表现相较于LeNet5更好。

(3)

对AlexNet模型采用3种不同的优化算法来训练，查看他们的训练测试结果，3种优化算法分别是：Adam，SGD，SGD+momentum。

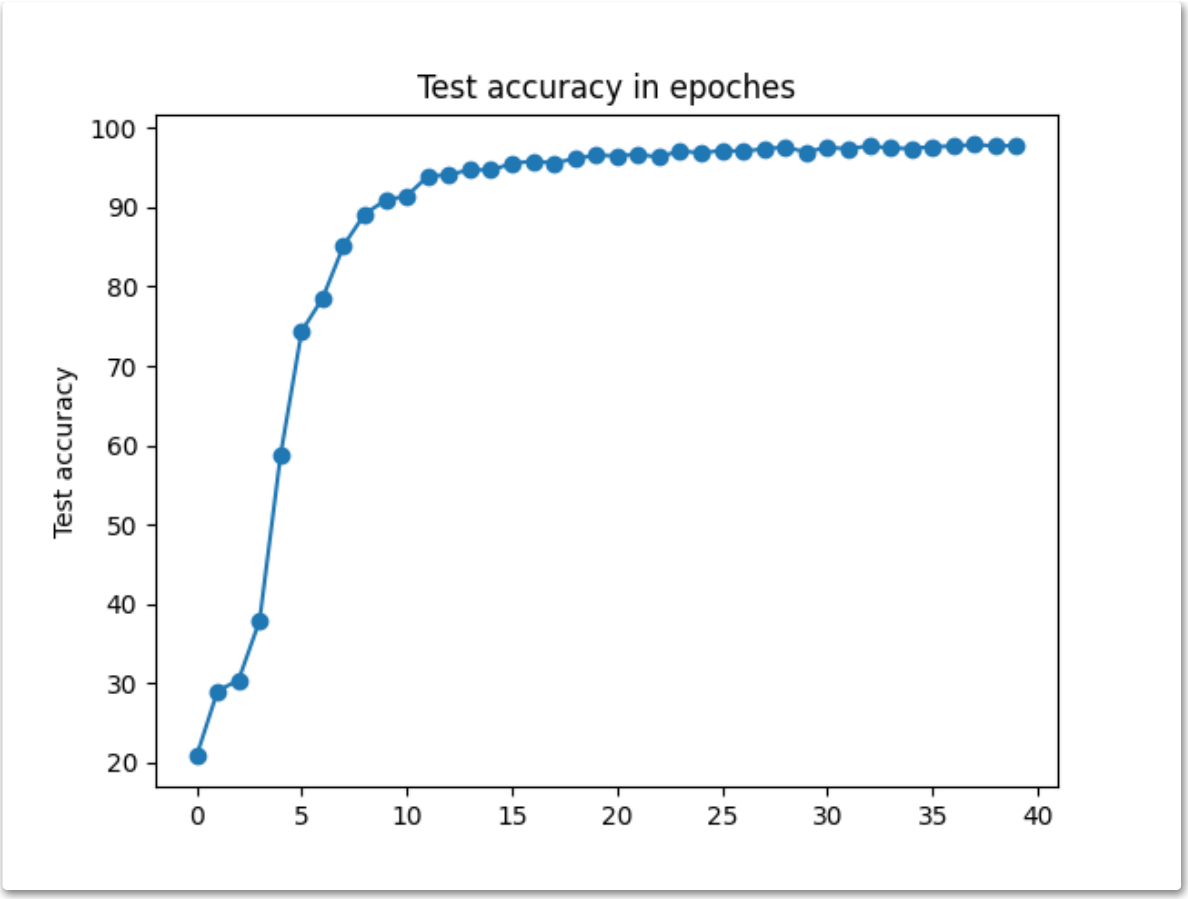
AlexNet+Adam:

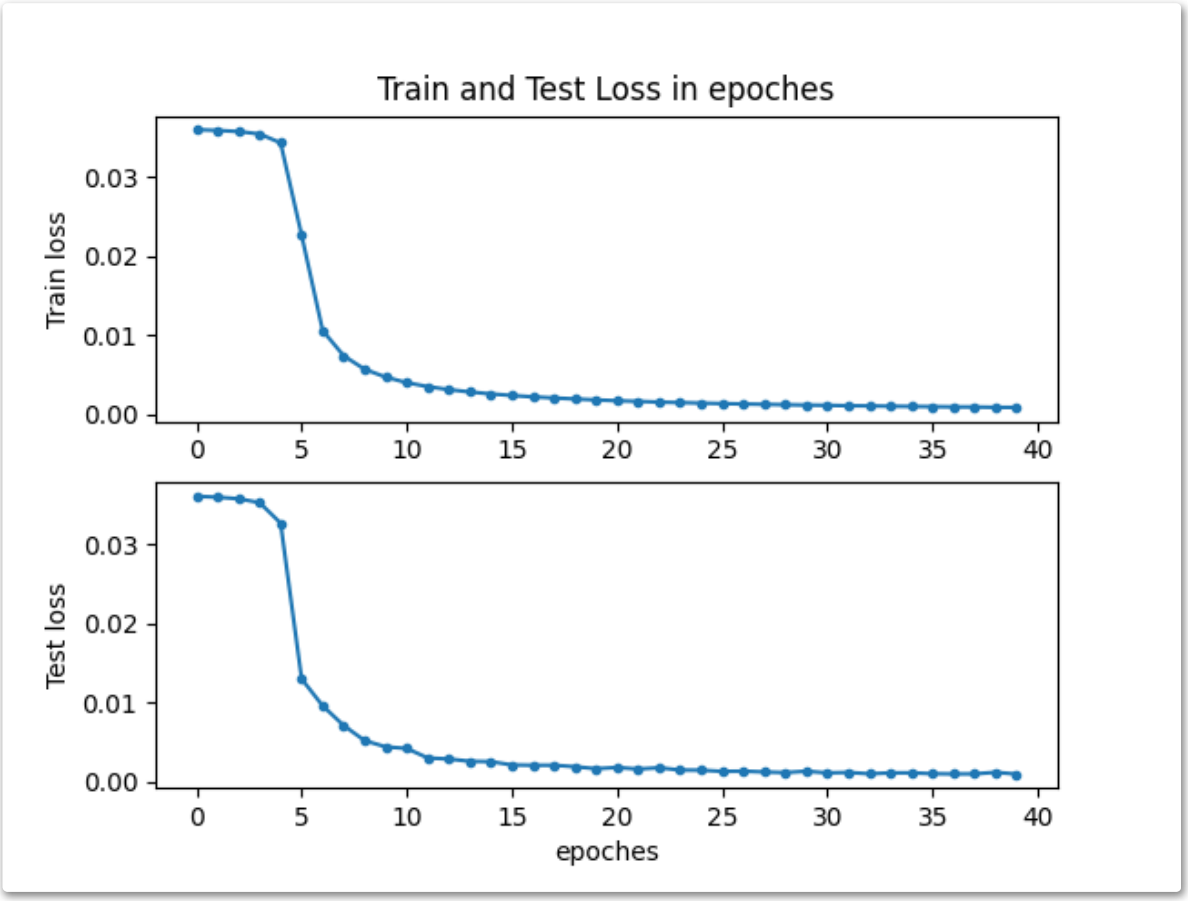




每一步的具体结果见附件./image/AlexNet+Adam.txt.

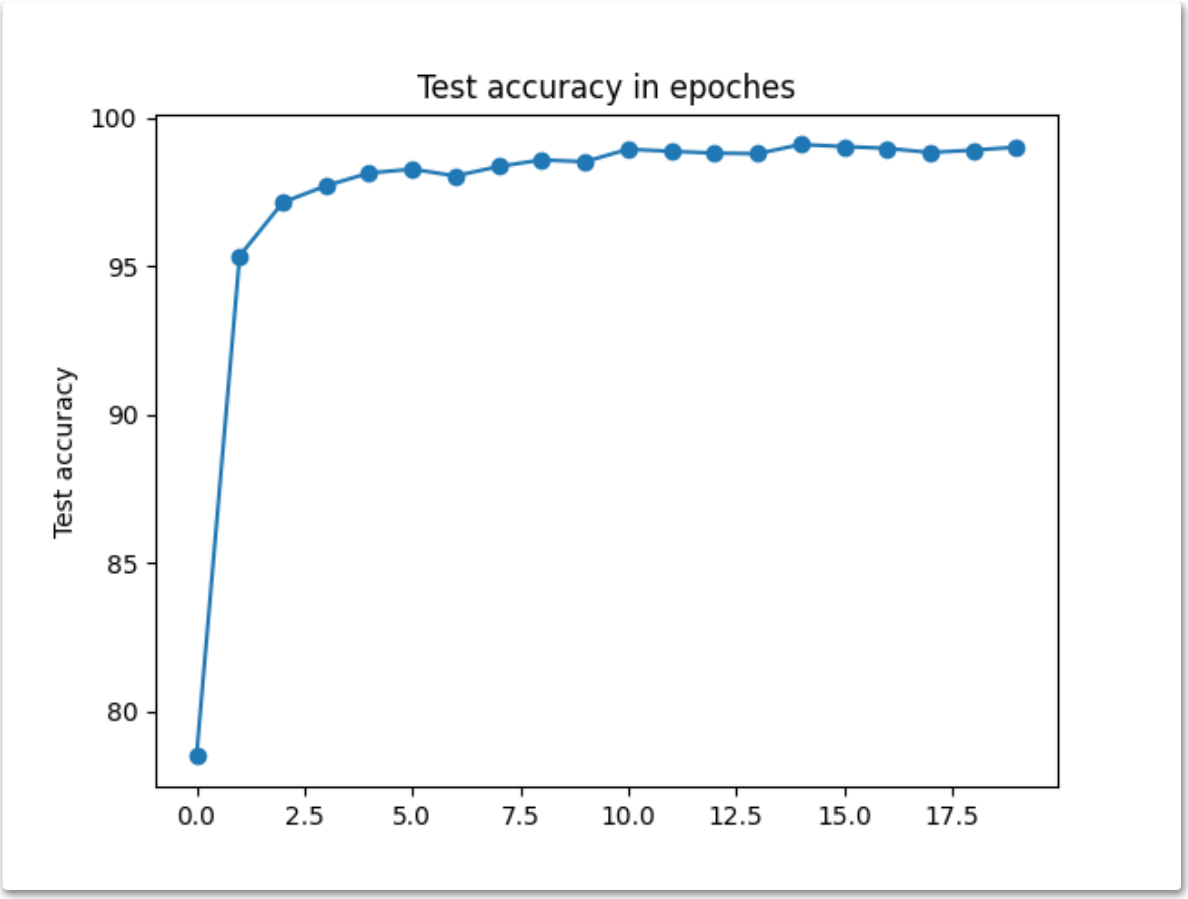
AlexNet+SGD(收敛太慢，加大epoch数，一共有40个epoch):

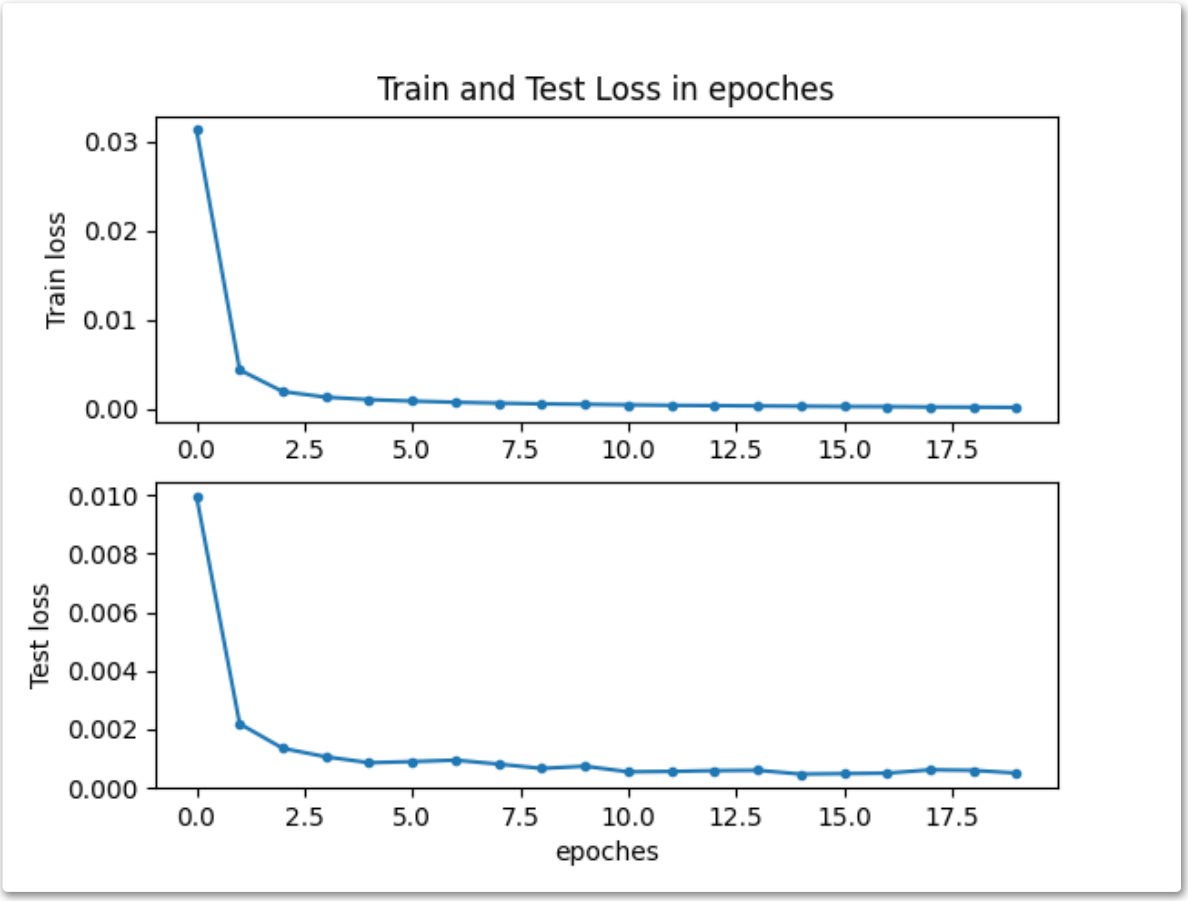




每一步的具体结果见附件./image/AlexNet+SGD.txt.

AlexNet+SGD+momentum:





每一步的具体结果见附件./image/AlexNet+SGD+momentum.txt.

Accuracy(%)	SGD	Adam	SGD+momentum
epoch=0	20.8000	97.6100	92.0300

Accuracy(%)	SGD	Adam	SGD+momentum
epoch=1	29.0200	98.5200	95.3400
epoch=2	30.4100	98.6200	97.1400
epoch=3	37.8900	98.7600	97.7000
epoch=4	58.8000	98.4600	98.1300
epoch=9	90.9100	98.6700	98.5100
epoch=14	94.6900	97.6400	99.0900
epoch=19	96.5700	98.4200	99.0000
epoch=24	96.8200		
epoch=29	96.9700		
epoch=34	97.4100		
epoch=39	97.8100		

首先比较三个优化算法的loss曲线变化情况，可以看出Adam算法的trainLoss曲线早期一直下降，而同期的testLoss曲线趋于不变，可能是模型过拟合了，而后期trainLoss和testLoss都趋于不变，训练应该是快到瓶颈了。而SGD算法和SGD+momentum算法都是早期trainLoss曲线和testLoss曲线一起下降，说明正在学习，后期下降速度变慢，趋于不变，是训练到达瓶颈，不过SGD算法要在更大的epoch时才这样。

根据上表的测试正确率来看，Adam算法最先到达一个较大的正确率（98.5%以上），其次是SGD+momentum算法，因为momentum有动量加速的效果，能够加速收敛的过程，最慢的是SGD算法且与其他两种算法有较大差别。在最后正确率情况上来看，由于测试epoch数相较于SGD算法来说还是太小，其最终结果在epoch=39时仍只有97.81%，不过理论上最后是可以到达和SGD+momentum收敛时的正确率差不多。而SGD+momentum最后的正确率是要高于Adam算法一些，不过相差不大，综合来说Adam算法更快达到一个高的正确率，但最后正确率不如SGD+momentum，SGD算法收敛速度较慢，最后正确率也较低（小epoch下）。

(4)

对于不同模型或者同一模型在不同优化算法下的training loss和validation loss(即testing loss)的图表结果详见(2)和(3)。通过分析这两种loss能够有效的判断当前模型的状态:

1.train loss不断下降，validation loss也不断下降，说明网络正在学习。

2.train loss不断下降，validation loss趋于不变，说明网络过拟合了。

3.train loss趋于不变，validation loss不断下降，说明数据集有问题。

4.train loss趋于不变，validation loss趋于不变，说明学习遇到瓶颈，到上限了。

5.train loss和validation loss都不断上升，说明网络结构设计不当。