

**深 度 学 习 实 验**

**实 验 报 告**

**实验名称：多层感知机的实现**

**姓 名：李岚琦**

**学 号：2111078**

**专 业：智能科学与技术**

**人工智能学院**

**2023年9月**

1. **问题简述**

使用多层感知机来对MNIST数据集进行图像分类。

1. **实验目的**
   1. 熟悉多层感知机的原理以及其代码实现
   2. 通过改变超参数，观察对多层感知机的训练和预测效果的影响
   3. 通过改变网络架构，来观察对网络效果的影响
2. **编译环境**

语言：python 版本： 3.7.16

编译器： pycharm 版本：2023

深度学习框架 ： pytorch 版本： 1.10.1

工具包 ： d2l

1. **基础代码与实验步骤**

**创建了两个文件： dataset.py用于进行数据集的加载和小批量导入dataloader， main.py用于构建网络并对数据集进行训练。**

**要把数据集放在和.py文件一样的名称为data的文件夹中，或者先运行dataset.py，可以直接下载数据集到相应位置。**

**首先是dataset.py：**

**代码：**

import torch  
import torchvision  
from torch.utils import data  
from torchvision import transforms  
from d2l import torch as d2l  
  
d2l.use\_svg\_display()  
  
# 通过ToTensor实例将图像数据从PIL类型变换成32位浮点数格式，  
# 并除以255使得所有像素的数值均在0～1之间  
trans = transforms.ToTensor()  
mnist\_train = torchvision.datasets.FashionMNIST(  
 root="../data", train=True, transform=trans, download=True)  
mnist\_test = torchvision.datasets.FashionMNIST(  
 root="../data", train=False, transform=trans, download=True)  
  
print(len(mnist\_train))  
print(len(mnist\_test))  
  
def get\_fashion\_mnist\_labels(labels): #@save  
 *"""返回Fashion-MNIST数据集的文本标签"""* text\_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',  
 'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']  
 return [text\_labels[int(i)] for i in labels]  
train\_dataloader = data.DataLoader(mnist\_train, batch\_size=256)  
test\_dataloader = data.DataLoader(mnist\_test, batch\_size=256)

**通过下载，将minist数据集转化为minist\_train和minist\_test两个dataset文件，并调用data.DataLoader将其送入数据迭代器。此时的batch\_size设置为256。**

**导入数据完毕后，主体文件是main.py ：**

**各部分代码与其解释：**

1. **导入所需的包和数据集，并且设置超参数：**
2. import torch  
   from torch import nn  
   from d2l import torch as d2l  
   import dataset as ppd  
     
   device = torch.device('cuda')  
   train\_dataset = ppd.mnist\_train  
   test\_dataset = ppd.mnist\_test  
   train\_dataloader = ppd.train\_dataloader  
   test\_dataloader = ppd.test\_dataloader  
   num\_inputs = 784  
   num\_outputs = 10  
   num\_hiddens1 = 256  
   num\_hiddens2 = 128

**关于为什么要把dataset缩写为pdd : 本来想着是prepared\_data的缩写，结果也成了拼多多的缩写。**

**这里设置两个隐藏层，第一个隐藏层的神经元个数是256，第二个隐藏层的神经元个数的128。输入大小是784，因为一会要把图片数据给展平了，输出是10因为有十个类别标签。**

1. **初始化各个层的参数：**

w1 = nn.Parameter(torch.randn(num\_inputs,num\_hiddens1,requires\_grad=True)\*0.01)  
b1 = nn.Parameter(torch.zeros(num\_hiddens1,requires\_grad=True))  
w2 = nn.Parameter(torch.randn(num\_hiddens1,num\_hiddens2,requires\_grad=True)\*0.01)  
b2 = nn.Parameter(torch.zeros(num\_hiddens2,requires\_grad=True))  
w3 = nn.Parameter(torch.randn(num\_hiddens2,num\_outputs,requires\_grad=True)\*0.01)  
b3 = nn.Parameter(torch.zeros(num\_outputs))  
params = [w1,b1,w2,b2 ,w3 ,b3]

**w1和b1是输入到第一个隐藏层的权重和偏置，w2和b2是第一个隐藏层到第二个隐藏层的权重和偏置，w3和b3是第二个隐藏层到输出的权重和偏置。**

1. **定义网络，relu激活函数和交叉熵损失函数**

def relu(x):  
 re = torch.zeros\_like(x)  
 return torch.max(re, x)  
def net(x):  
 x= x.reshape(-1,num\_inputs)  
 hidden1 = relu(x@w1 + b1)  
 hidden2 = relu(hidden1@w2 + b2)  
 out = hidden2@w3 + b3  
 return out  
loss = nn.CrossEntropyLoss(reduction='none')

1. **设置迭代次数和学习率，设置优化器并进行训练，并且可视化**

num\_epochs = 10  
lr = 0.1  
updater = torch.optim.SGD(params,lr=lr)  
d2l.train\_ch3(net,train\_dataloader,test\_dataloader,loss,num\_epochs,updater)

**这里迭代次数设置为10，学习率为0.1，优化器是随机梯度下降法，并且调用d2l.train\_ch3()函数来进行训练**

**同时，因为pycharm编译器原因，直接运行该代码，train\_ch3()函数将不能正常绘制图片，所以还需要更改一下train\_ch3()函数的基本代码：**

**找到torch.py中的train\_ch3()函数的定义，再找到其中Animator类的add函数的定义，在倒数第二和第三行增加代码：**

plt.draw()  
plt.pause(0.001)

**，然后回到main.py，在最后部分增加代码：**

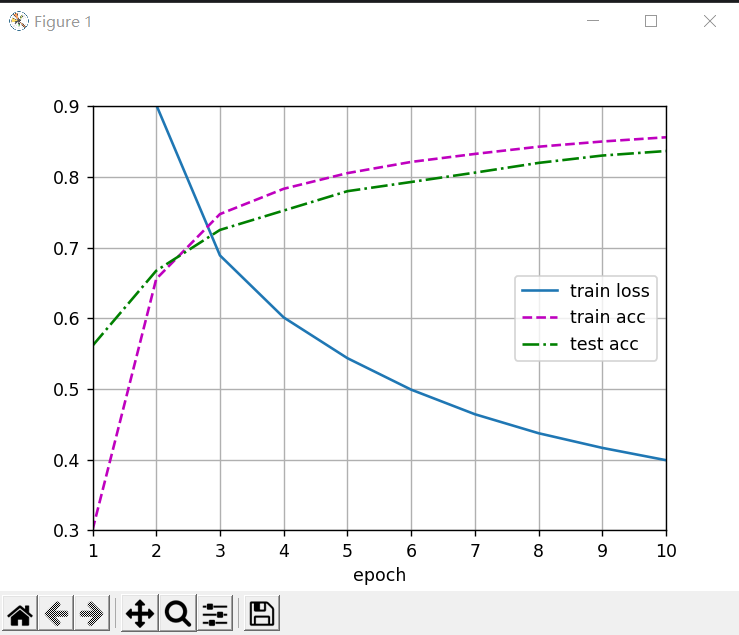
d2l.plt.show()

**即可正常使用绘图功能。**

1. **进行训练与预测**

**运行代码，观察运行结果出现的损失函u数，训练结果和测试误差关于epoch的变化：**

**如下图，可以看到随着epoch的增大，误差损失逐渐减小，而训练准确率和测试准确率上升。但是三者的变化率（导数绝对值）逐渐减小并趋近于零。**

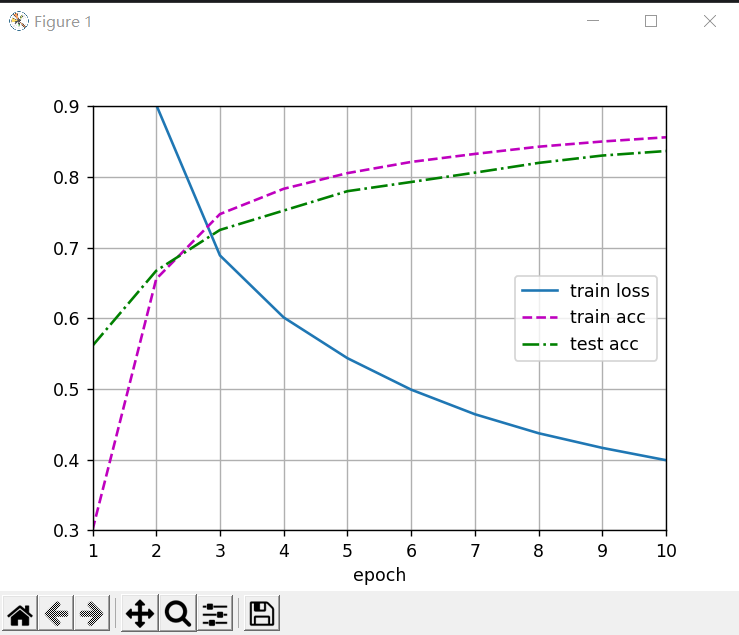
****



1. **实验的8个要求和7个附加题**

**要求1：画出训练过程中训练损失函数、训练误差、测试误差随迭代epoch的变化曲线：**

**答： （\*\*在本报告中，考虑到train acc 和 test acc实质上就是1减去训练误差和测试误差，所以之后直接对train acc 和 test acc进行讨论）**

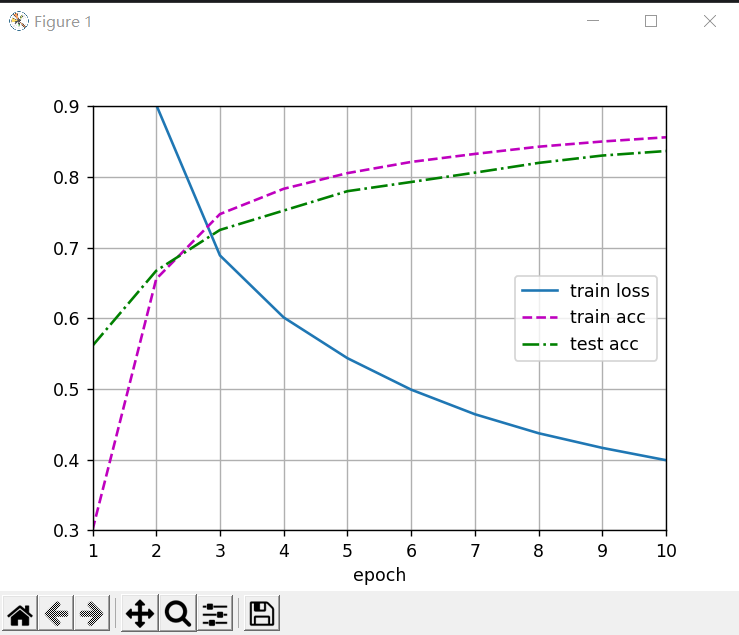
****

要求2：在其他参数保持不变的情况下，更改每个隐藏层的神经元数目，观察要求1中3个变化曲线

答： 原本的两个隐藏层神经元个数分别是256和128：

num\_hiddens1 = 256  
num\_hiddens2 = 128

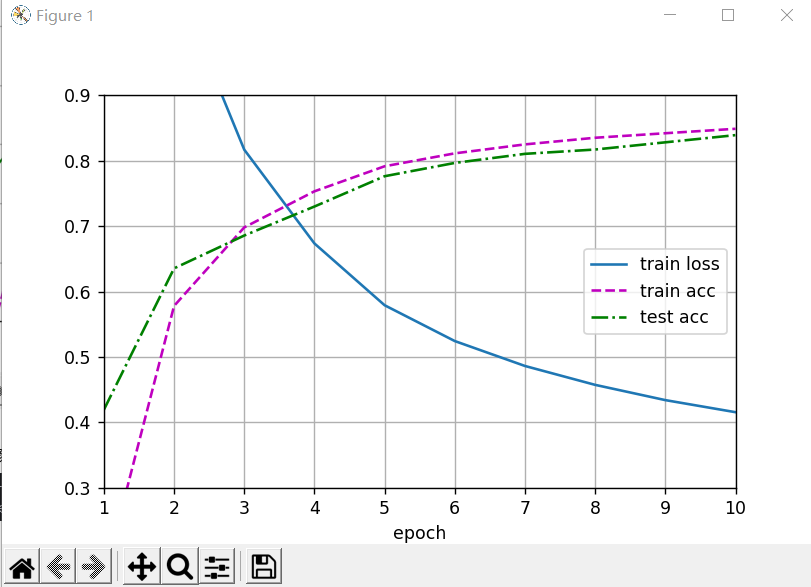
图像是：

****

将其个数修改为128和56：

num\_hiddens1 = 128  
num\_hiddens2 = 56

图像为：

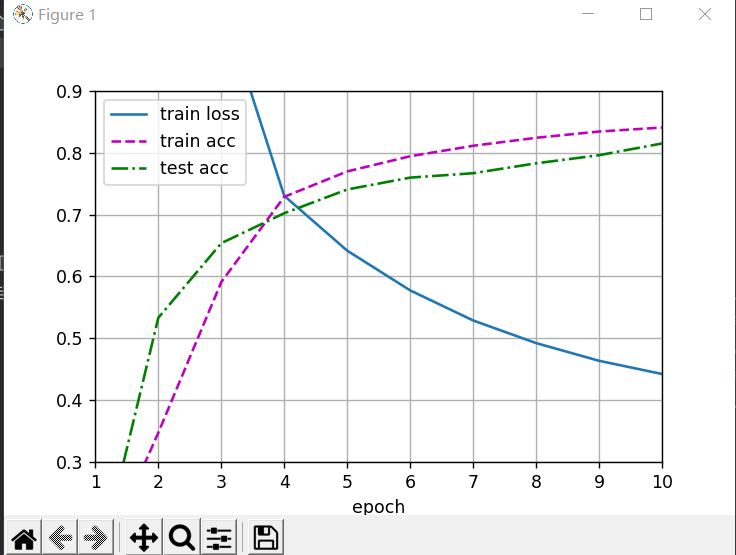


可以观察到，epoch为1时（即第一轮训练后），图一的train acc 和test acc 均明显高于图二的，猜测是因为神经元数量的减少导致其学习的速度变慢；同时两者均在epoch为5左右趋于稳定，推测在本实验中，神经元数量对最终稳定速度和稳定acc值没有过大影响。

再进一步减少神经元，将神经元改为56和28：

num\_hiddens1 = 56  
num\_hiddens2 = 28

运行图像为：



发现初始（或者说前期）的训练和测试准确率都比原本有明显下降，和刚刚的猜想符合。因此可以大致认为，神经元数量越少，在初始阶段的学习越困难，即准确率越低。

要求3：在其他参数保持不变的情况下，更改隐藏层数，观察要求1中3个变化曲线

考虑增加一个隐藏层，变成三个隐藏层：

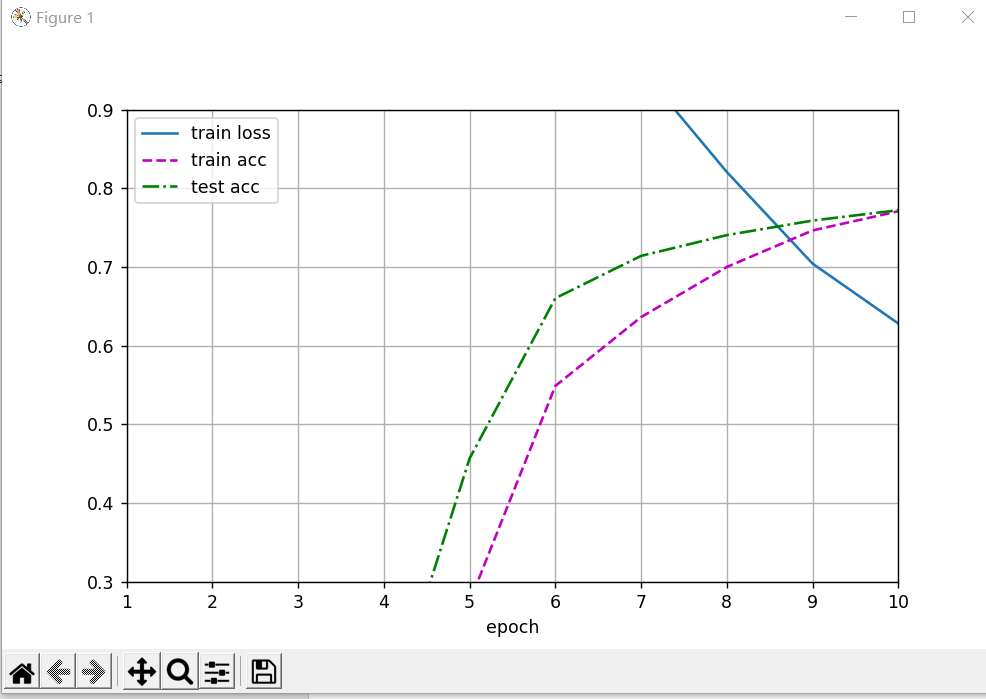
代码：

num\_hiddens1 = 256  
num\_hiddens2 = 128  
num\_hiddens3 = 56

w4 = nn.Parameter(torch.randn(num\_hiddens3,num\_outputs,requires\_grad=True)\*0.01)  
b4 = nn.Parameter(torch.randn(num\_outputs,requires\_grad=True))  
params = [w1,b1,w2,b2 ,w3 ,b3,w4,b4]

def net(x):  
 x= x.reshape(-1,num\_inputs)  
 hidden1 = relu(x@w1 + b1)  
 hidden2 = relu(hidden1@w2 + b2)  
 hidden3 = relu(hidden2@w3 + b3)  
 out = hidden3@w4 + b4  
 return out

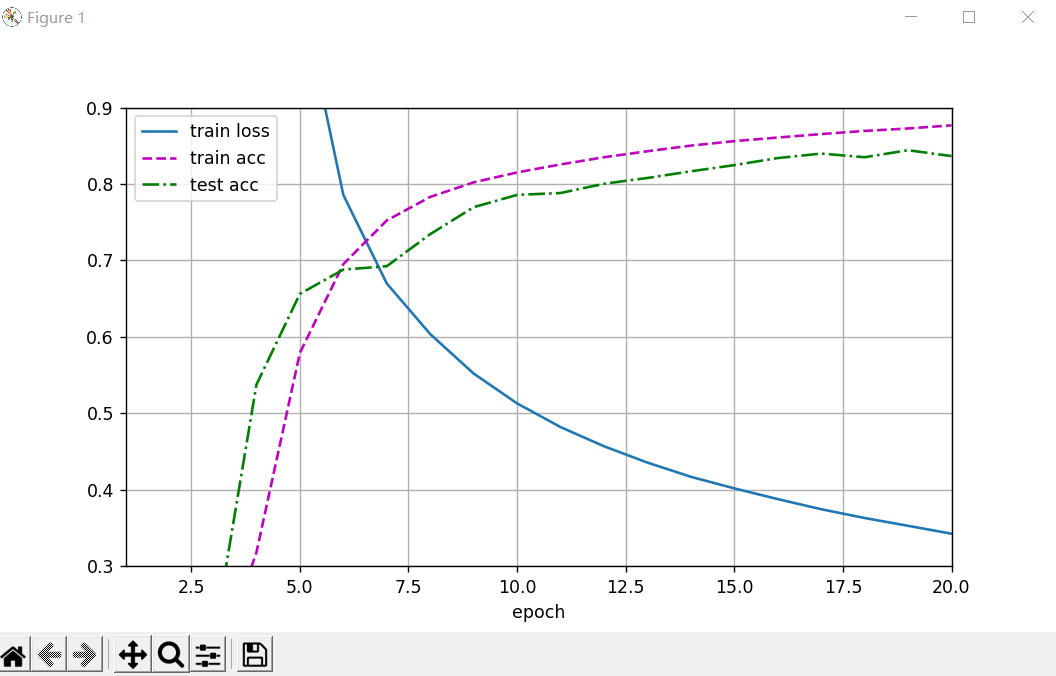
运行得：



可以观察到，当加深了隐藏层之后，在最初的几次迭代中，训练损失较大，而训练准确率和测试准确率较低，在原来的10布之内难以收敛，因此需要加大epoch：

将epoch调整到20：

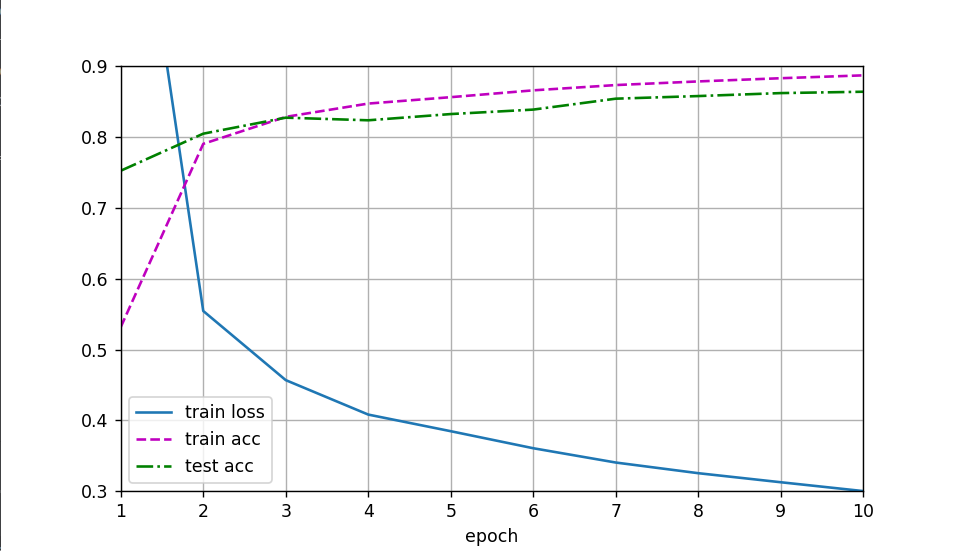
运行结果：



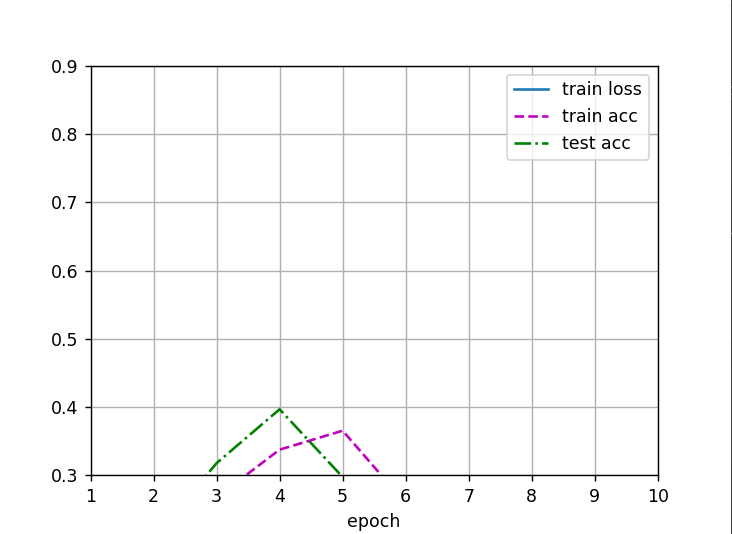
可以观察到，在epoch为12.5左右时，逐渐稳定。所以大致可以认为，更深的网络也需要更多的epoch次数。

要求4：在其他参数保持不变的情况下，更改学习率，观察要求1中3个变化曲线

将学习率更改为0.5：



观察到，和lr = 0.1 相比，稳定到最终值所需的epoch更少了，也就是能更快地学习。那么是否lr 越大越好？ 将lr 设置为 5运行结果为：

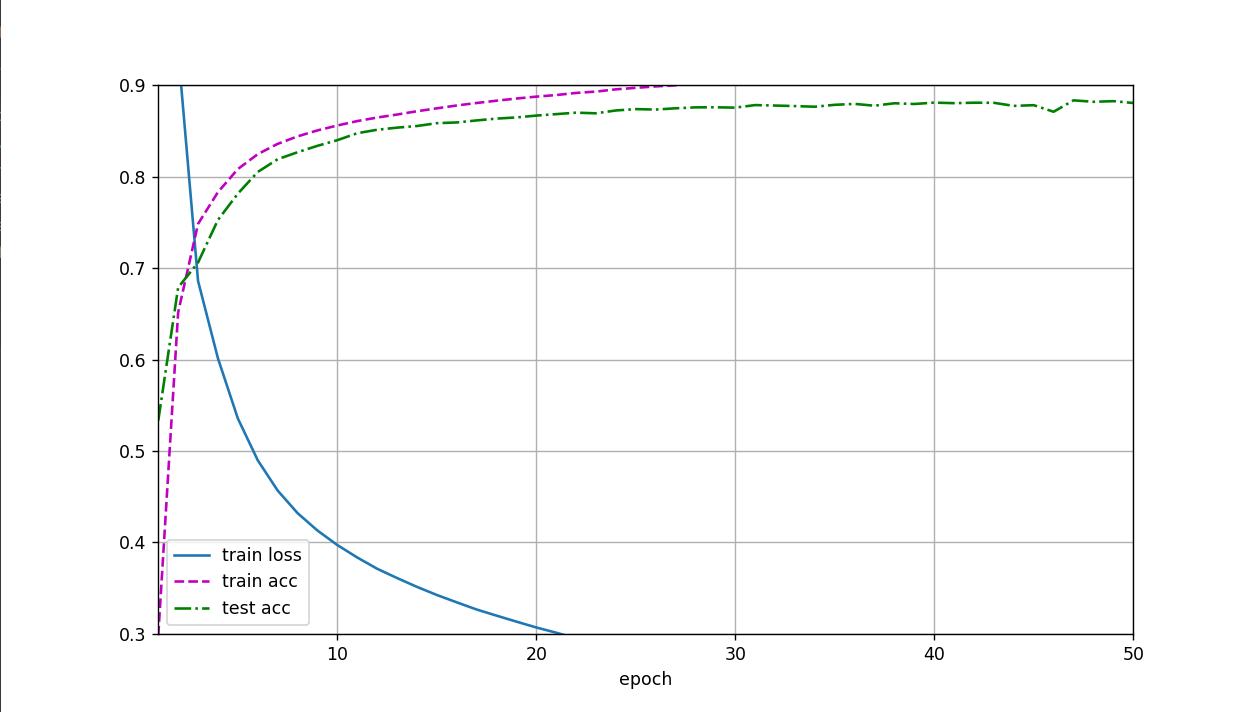


不仅无法完成学习，而且每次运行的结果都不太一样。推测这是因为lr设置过大而“跨过”了参数应该学习的最大步幅，导致不仅无法完成收敛，反而越来越不稳定。

因此，lr过大可能会无法完成学习，过小则学习过慢。

要求5：在其他参数保持不变的情况下，更改训练轮数，观察要求1中3个变化曲线

因为epoch为10时，已经涵盖了epoch为小于10的情况了。同理，epoch较大时，也已经涵盖了小于所取值的情况，所以我们只需要设置一个较大的epoch，便可以观察到曲线随epoch的变化。这里将epoch设置为50



如图，发现epoch从20之后，再增加，已经没有什么实质性变化。同时在47左右时，甚至出现了一次test\_acc的下降，这可能是出现过拟合的风险提示，或者是已经出现了过拟合。

因此，epoch既要保证能够使训练达到稳定，也不能冒着过拟合的风险而一味地往大了取。

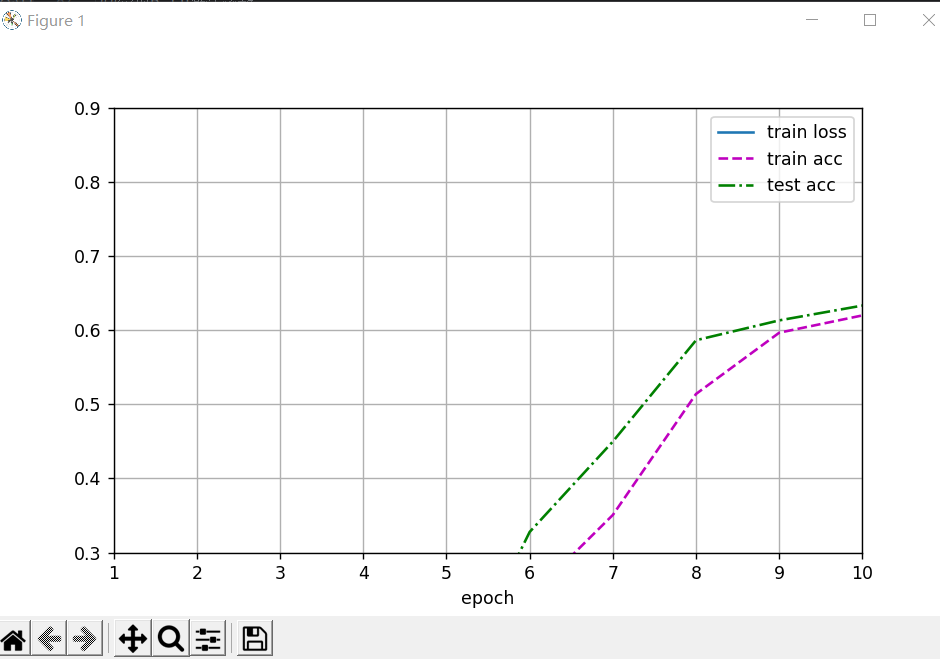
要求6：尝试不同的激活函数，观察要求1中3个变化曲线

考虑将ReLU函数变成sigmond函数，进行测试：

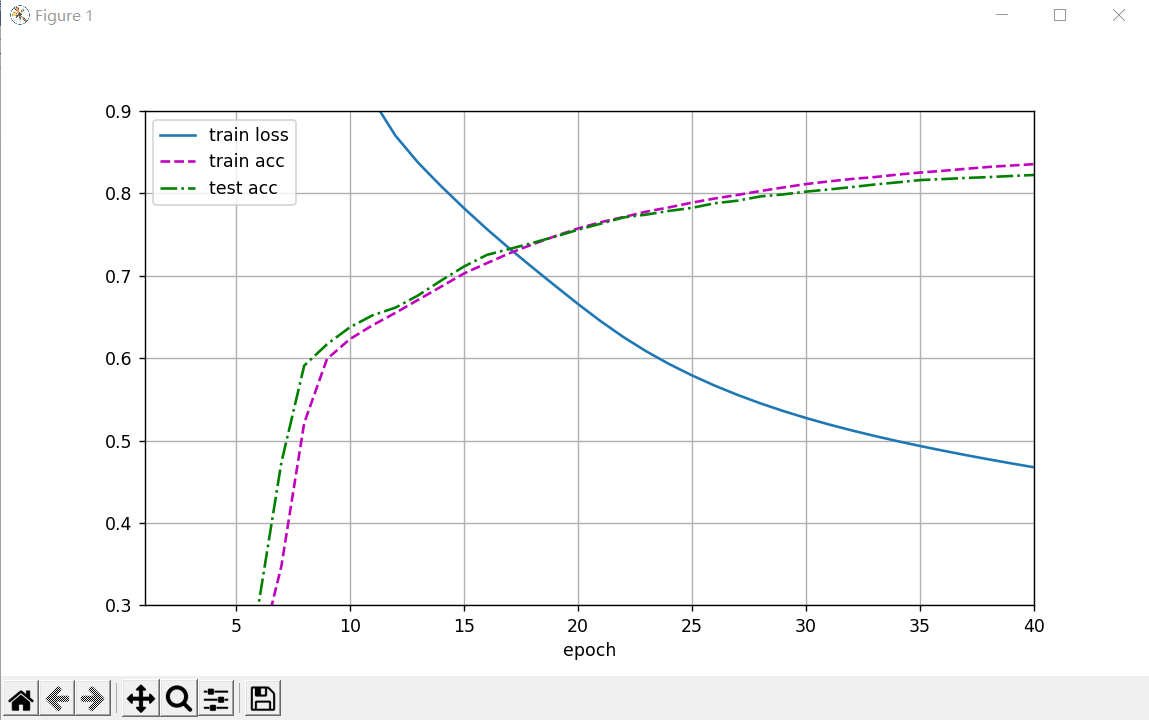
把网络修改为：

def net(x):  
 x= x.reshape(-1,num\_inputs)  
 hidden1 = torch.sigmoid\_(x@w1 + b1)  
 hidden2 = torch.sigmoid\_(hidden1@w2 + b2)  
 out = hidden2@w3 + b3  
 return out

运行得:



发现其学习变慢了，推测可能需要更多的epoch学习次数，因此，把epoch改为40再运行：

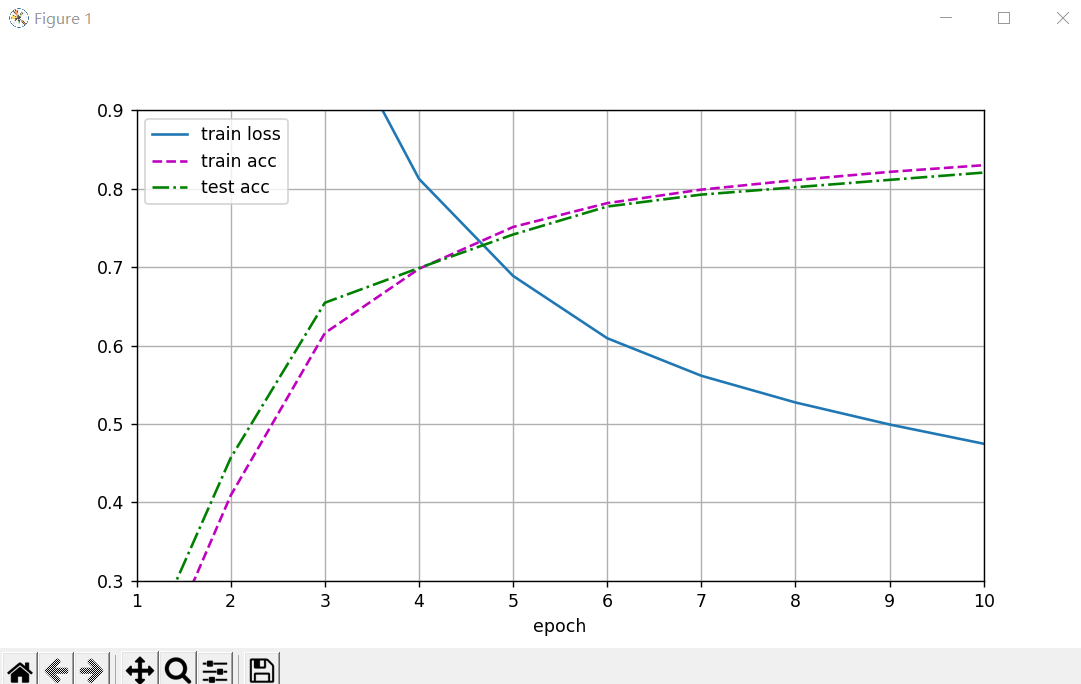


发现更换了激活函数后，最终效果虽然没有明显变化，但是训练所需的epoch变多了。

要求7：尝试不同的初始化权重，观察要求1中3个变化曲线

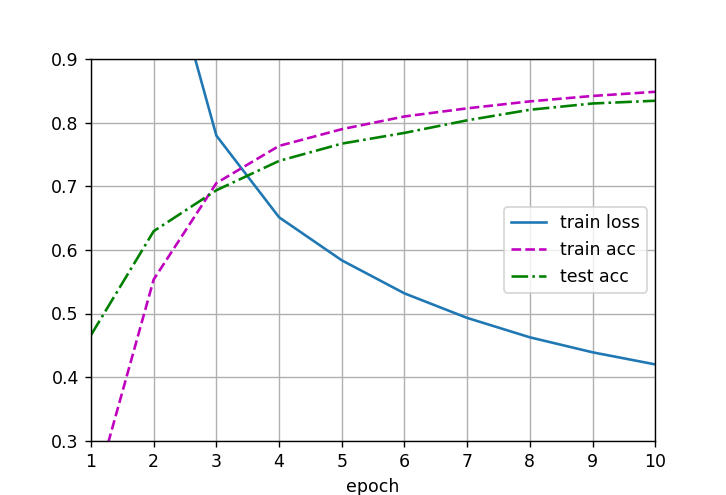
把原来的randn函数改成rand函数进行初始化：

运行结果为：



可以观察到，在epoch较低时，与原版randn初始化有一定的区别（训练和测试的准确率较小），但是之后则与原版的初始化的结果相似。

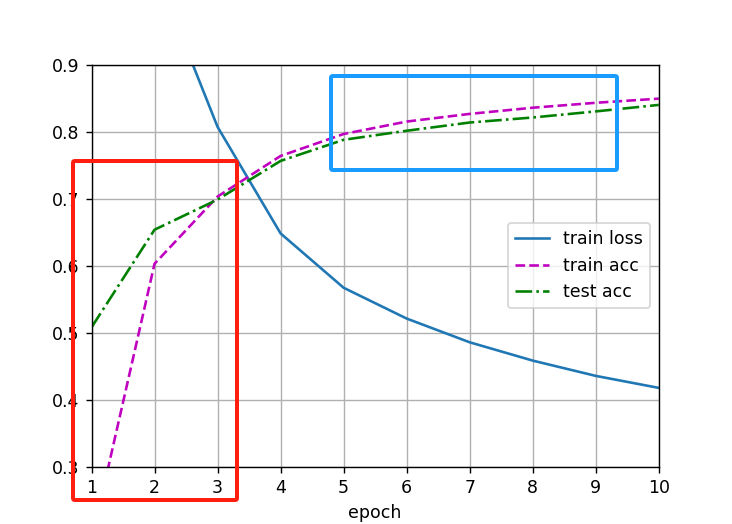
要求8：通过对所有参数进行联合优化，观察要求1中3个变化曲线

通过上述要求1~要求7的分析，做以下的参数优化：将第一个隐藏层的神经元设置为128，将第二个隐藏层的神经元设置为56，学习率为0.1，epoch为10

此为结果较好较稳定的参数设置。

附加题1：从欠拟合的角度解释不同训练轮数对结果的影响

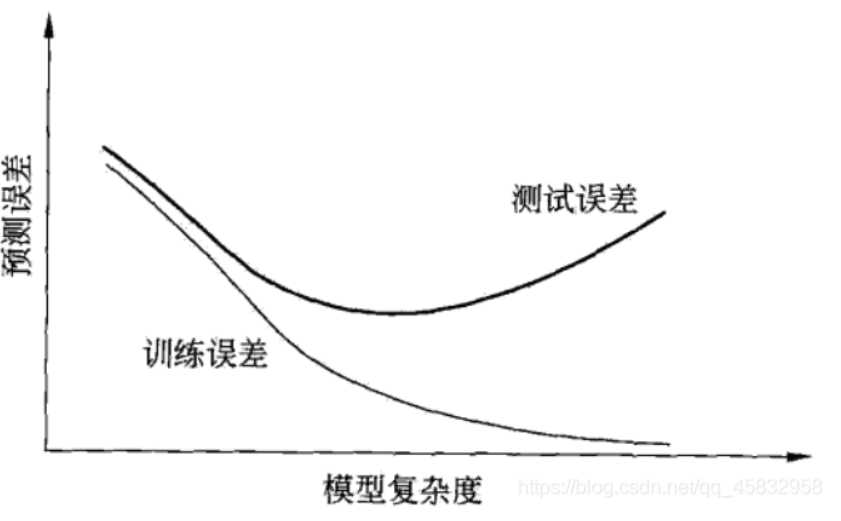
在原本的运行结果图（见下图）中，可以看到，当epoch较小时，出现了测试集准确率比训练集高的现象（红色框框出来的），这时即发生了欠拟合现象。当训练轮数较小时，各个参数没有达到最优解，损失函数也没有降到最低，因此此时可以看成是还没完成学习，也就出现了欠拟合现象。然而随着epoch增加，可以看到欠拟合现象逐渐消失。而再增加，则出现了过拟合（蓝色框框出来的）现象。



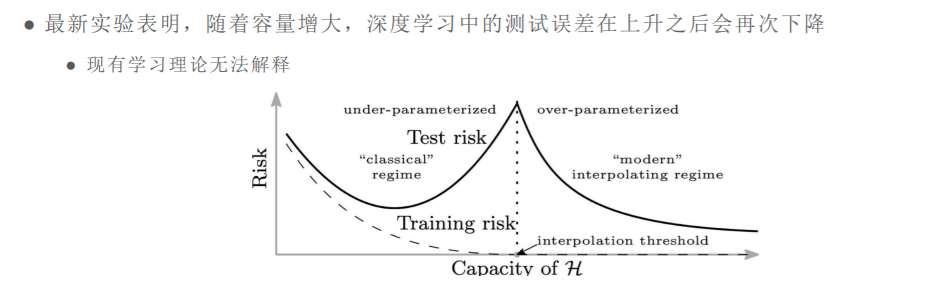
要求2：解释模型复杂度（隐藏层层数、隐藏层神经元数）与训练误差和测试误差的关系

答：随着模型复杂度增加，训练误差不断减小，而测试误差先减小，到达一个较低点后，又逐渐增加（过拟合）。

然而最新的研究表明，当模型复杂度足够大时，先前增加了的测试误差（过拟合）又会逐渐下降。

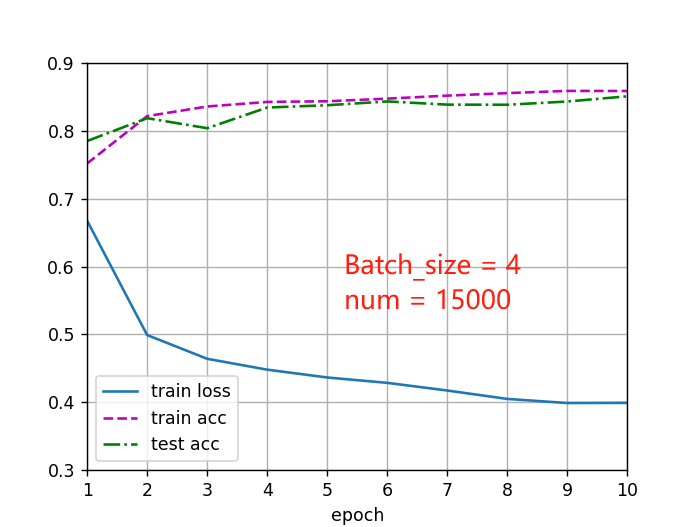


最新的研究发现：



要求3：观察不同样本数下的训练和测试误差，并解释样本数与训练误差和测试误差的关系

答：通过对dataset.py中，最后两行的batch\_size参数的改变，可以完成每次的训练样本数不同。

把batch\_size设置为4：（即每次取15000个样本进行训练）：结果为：

可以看到，当减小了Batch\_size,也就是增大了训练样本数时，测试集和训练集的准确率在第一次训练轮数就可以达到很高。但是相应的，过拟合现象也更加严重。

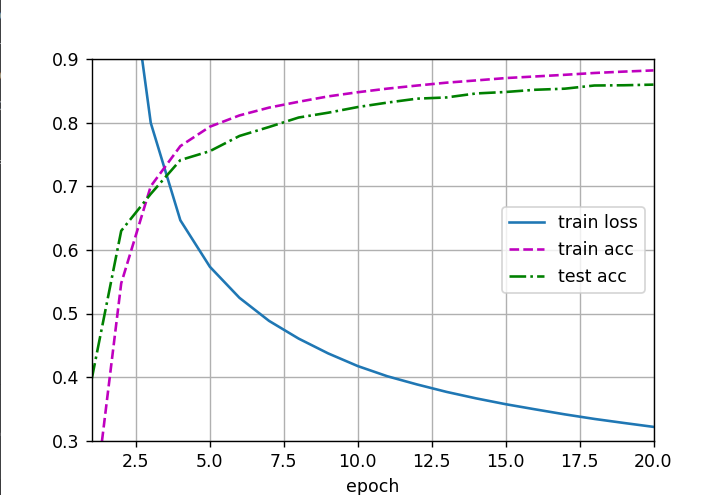
要求4：观察训练误差和测试误差的走势，解释模型复杂度和样本数与过拟合的关系

答： 由要求3和要求2综合分析，模型复杂度越高、样本数越多，模型过拟合的风险就越大。

要求5：在损失函数中增加L2正则化，观察训练误差和测试误差

答：考虑到L2正则化主要解决了过拟合问题，而为了观察到更加明显的过拟合现象，将epoch增大到20。

不加正则化时：

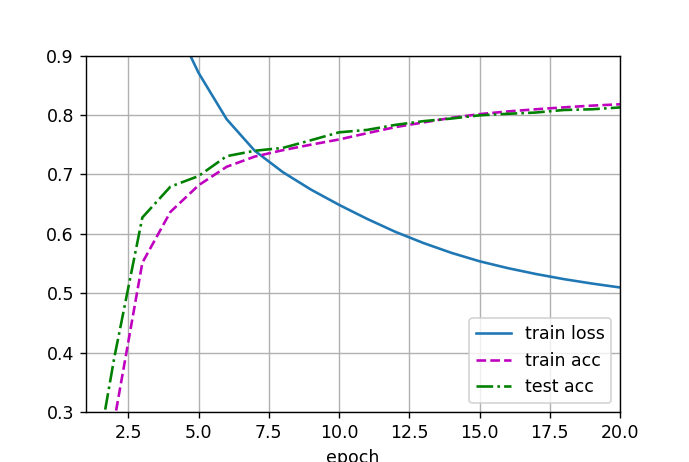


可以观察到，当epoch大于2.6时，已经出现了较为明显的过拟合现象。

现在加入L2正则化：代码为：、

updater = torch.optim.SGD([{"params":w1,'weight\_decay': 0},  
 {"params":b1},{"params":w2,'weight\_decay': 0.2},  
 {"params":b2},{"params":w3,'weight\_decay': 0},  
 {"params":b3}],lr=lr)

其中对w2进行参数为0.2的正则化，运行结果为：



可以观察到，在添加了L2正则化之后，尽管epoch较大时，train acc和test acc也仍然较为接近，有效地抑制了过拟合的现象。

要求6：使用dropout，观察其对泛化的影响

为了更方便地使用dropout，首先将原有对网络的详细定义改成简洁定义：

simple\_net = nn.Sequential(nn.Flatten(),nn.Linear(num\_inputs,num\_hiddens1),nn.ReLU(),  
 nn.Linear(num\_hiddens1,num\_hiddens2),nn.ReLU(),  
 nn.Linear(num\_hiddens2,num\_outputs))

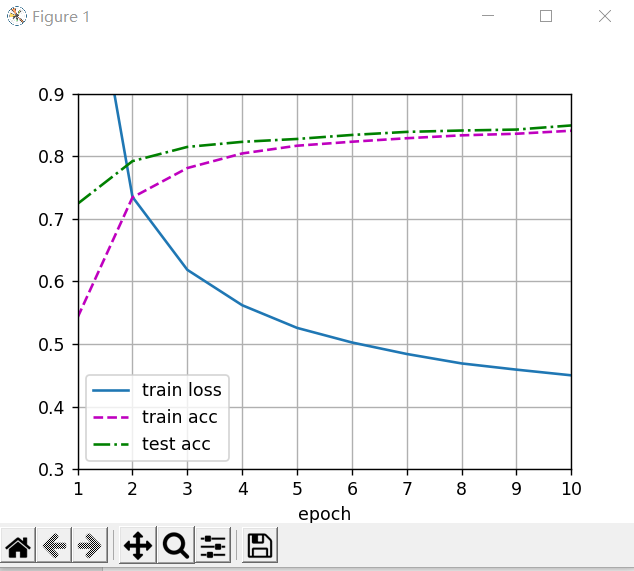
然后在第一个和第二个隐藏层之后使用dropout，其中抛弃率分别为0.5和0.2：

simple\_net = nn.Sequential(nn.Flatten(),nn.Linear(num\_inputs,num\_hiddens1),nn.ReLU(),nn.Dropout(0.5),  
 nn.Linear(num\_hiddens1,num\_hiddens2),nn.ReLU(),nn.Dropout(0.2),  
 nn.Linear(num\_hiddens2,num\_outputs))

因为修改了网络定义（由手动定义变为简洁定义），训练时的代码也进行更改：

updater = torch.optim.SGD(simple\_net.parameters(),lr=lr)  
simple\_net.train()  
d2l.train\_ch3(simple\_net,train\_dataloader,test\_dataloader,loss,num\_epochs,updater)  
simple\_net.eval()  
d2l.predict\_ch3(simple\_net,test\_dataloader)  
d2l.plt.show()

运行结果为：



可以观察到，在epoch大于5后，test acc 和 train acc便十分接近，并且test acc 甚至要始终高于train acc一些，因此可以认为，dropout的加入使得模型的泛化能力有了明显提高。

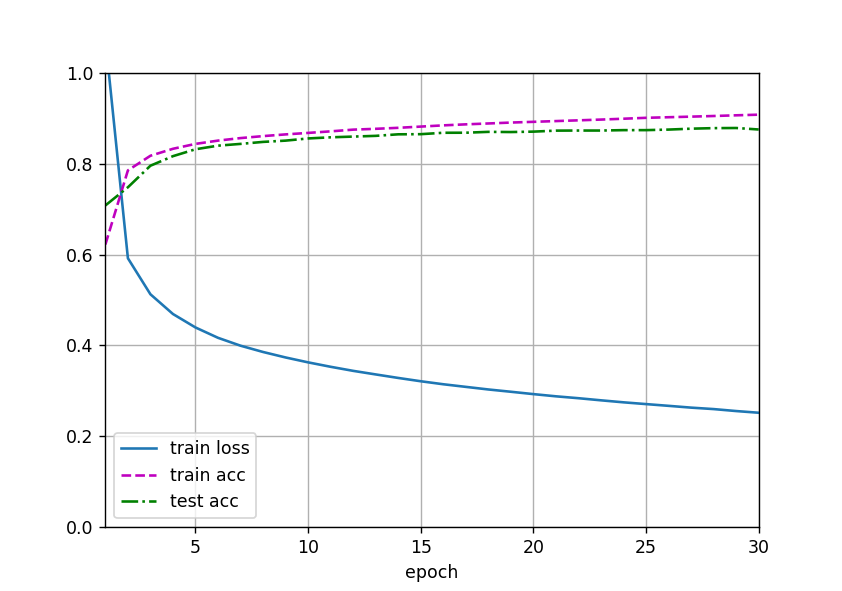
要求7：使用batch normalization，观察其对训练误差和测试误差的影响

在隐藏层一和二之后，都加入批量标准化层：

simple\_net = nn.Sequential(nn.Flatten(),nn.Linear(num\_inputs,num\_hiddens1),nn.BatchNorm1d(num\_hiddens1),nn.ReLU(),nn.Dropout(0.5),  
 nn.Linear(num\_hiddens1,num\_hiddens2),nn.BatchNorm1d(num\_hiddens2),nn.ReLU(),nn.Dropout(0.2),  
 nn.Linear(num\_hiddens2,num\_outputs))

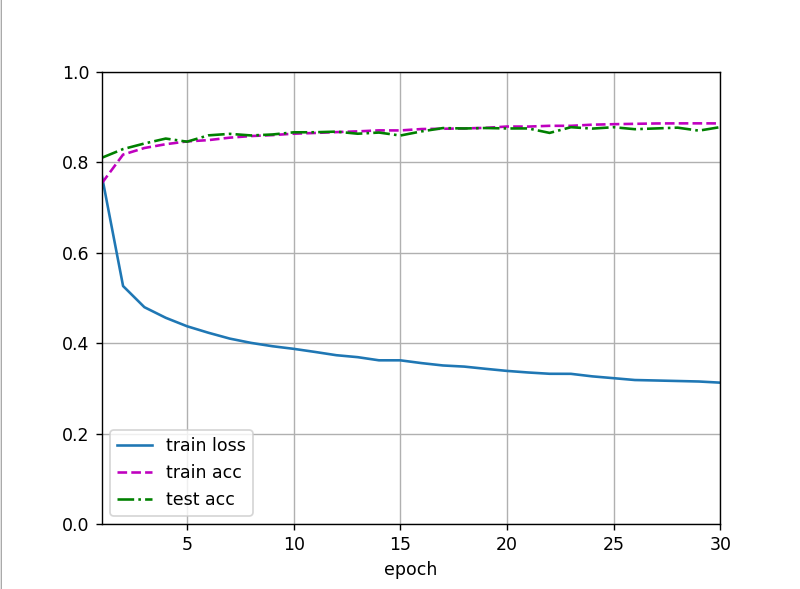
考虑到batch\_normalization主要是为了缓解过拟合，所以我们再次把epoch调大至30，从而更好观测其对过拟合的影响：

不加batch\_normalization：



可以看到，随着epoch增大，test acc始终低于train acc（过拟合），而且二者的差距有着逐渐变大的趋势（过拟合现象逐渐严重）。

加入batch\_normalization之后：



可以观察到，即使epoch较大，即过拟合的风险较大，batch\_normalization技术依然可以很好地抑制过拟合的现象，使得test和train的准确率不至于相差太多。

要求8：画出每个隐藏层的输出，尝试解释深度学习的工作原理

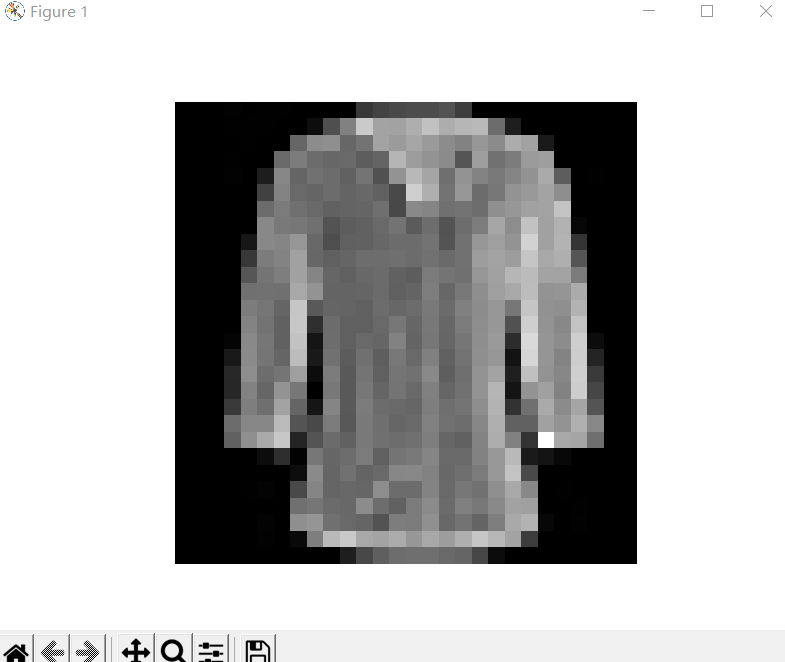
因为要查看隐藏输出，所以要以一张图片为例。同时，因为此时的多层感知机是线性模型，因此每一个隐藏输出实际上是一个列向量，我们需要调用reshape函数将其转化成2维张量，然后才能以图片形式查看。

我们选择测试集的第四张图片，先看看它是什么：

代码：

the\_img\_we\_look = test\_dataset[4][0].squeeze()  
plt.imshow(the\_img\_we\_look, cmap="gray")  
plt.axis("off")  
plt.show()

运行结果：



可以看到是一件衬衫。

然后编写查看中间隐藏层输出的函数：

代码：

定义函数：

def what\_is\_going\_on(x) :  
 b11 = b1.data  
 b22 = b2.data  
 w11 = w1.data  
 w22 = w2.data  
 x = x.squeeze()  
 x = x.reshape(-1,num\_inputs)  
 hidden1 = relu(x@w11+b11)  
 the\_hidden1\_out = hidden1.reshape(16, 8)  
 plt.imshow(the\_hidden1\_out, cmap="gray")  
 plt.axis("off")  
 plt.show()  
 print(hidden1.shape)  
 hidden2 = relu(hidden1@w22 + b22)  
 the\_hidden2\_out = hidden2.reshape(7, 8)  
 plt.imshow(the\_hidden2\_out, cmap="gray")  
 plt.axis("off")  
 plt.show()  
 print(hidden2.shape)

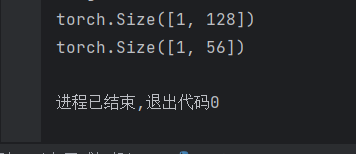
在这里，为了不让我们的查看影响到网络的参数变化，因此使用了参数的.data属性。同时输出了两个隐藏层的原始输出大小，并且把它俩进行reshape后，再作为图片展示。

在主代码的最后调用函数：

what\_is\_going\_on(test\_dataset[4][0])

运行结果：

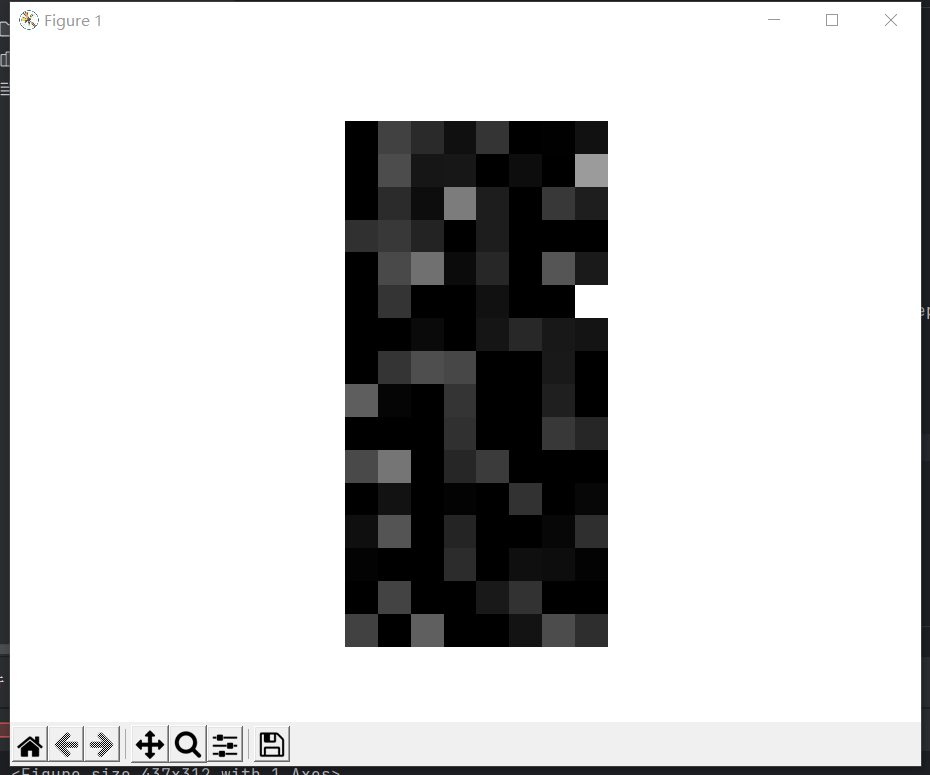
两个隐藏层的原始输出大小：



也就是说，隐藏层1的输出为1 x 128的张量。我将其reshape为16 x 8的张量。

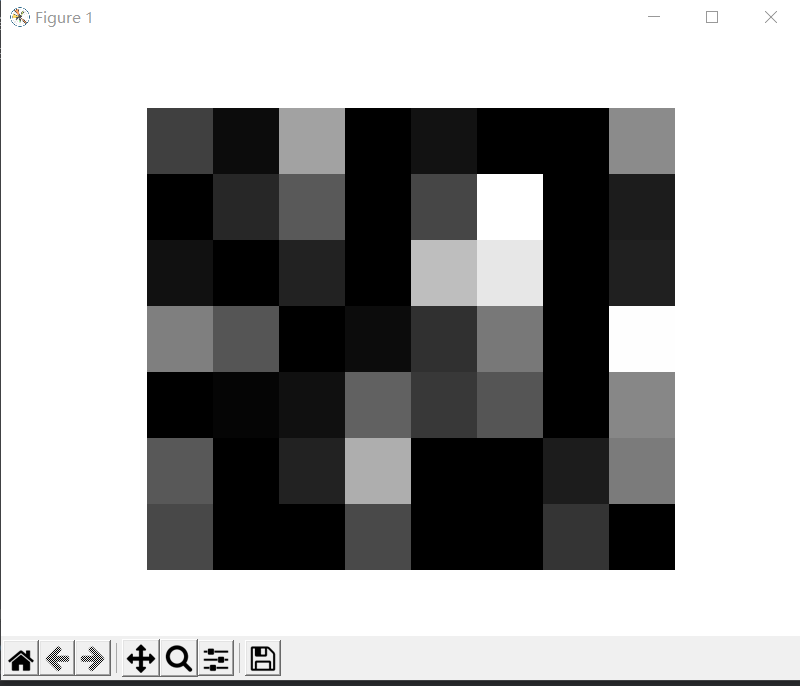
隐藏层2的输出为1 x 56 的张量，我将其reshape为7 x 8的张量。

展示结果：隐藏层1的输出：



如上图，是一个16 x 8的像素图。

隐藏层2的输出展示：



如图，是一个7 x 8的像素图。

总结：因为本次多层感知机使用的是线性模型，在处理图像时，事实上并没有把它当成“图像”看待，而是把它展平为了一个一维张量（具体实现中的

x= x.reshape(-1,num\_inputs)

以及简洁实现中的

nn.Flatten()

都是在做这个事），因此在各个隐藏层的输出，其实也都是一个列向量。因此，我们并不能期望能够像卷积神经网络那样，直观地看出来隐藏层对于图像特征的提取，而只是看到了各种作为“特征”的列向量。我们将其reshape并且作为二维图像进行输出展示，也是为了以后的卷积神经网络中的卷积层进行对比，从而加深对深度学习网络的理解。