南開大學
Nankai University

深 度 学 习 实 验
实 验 报 告

实验名称：优化算法

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

一、问题简述

使用多层感知机来对 MNIST 数据集进行图像分类并且着重探究优化算法对于训练和预测的影响。

二、实验目的

1. 尝试不同的优化算法对于训练和预测的影响
2. 通过改变参数初始化方式，观察其对训练和预测效果的影响
3. 通过应用学习率调度器，来观察对网络效果的影响

三、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

四、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：

```
import torch
from torch import nn
from d2l import torch as d2l
import dataset as ppd
from torch.optim import lr_scheduler
d2l.use_svg_display()
device = d2l.try_gpu()
train_dataset = ppd.mnist_train
test_dataset = ppd.mnist_test
train_dataloader = ppd.train_dataloader
test_dataloader = ppd.test_dataloader
```

2. 定义高斯初始化和 Xavier 初始化函数：

```
def init_normal(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, mean=0, std=0.01)
        nn.init.zeros_(m.bias)
def init_xavier(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
```

3. 定义训练函数：

```
def train(net, train_iter, test_iter, num_epochs, loss, trainer,
device,
```

```
        scheduler=None):
    net.to(device)
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'test
acc'])

    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)  #
train_loss,train_acc,num_examples
        for i, (X, y) in enumerate(train_iter):
            net.train()
            trainer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l = torch.sum(l)
            l.backward()
            trainer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y),
X.shape[0])
            train_loss = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % 50 == 0:
                animator.add(epoch + i / len(train_iter),
                             (train_loss, train_acc, None))

        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch+1, (None, None, test_acc))

        if scheduler:
            if scheduler.__module__ == lr_scheduler.__name__:
                # UsingPyTorchIn-Builtscheduler
                scheduler.step()
            else:
                # Usingcustomdefinedscheduler
                for param_group in trainer.param_groups:
                    param_group['lr'] = scheduler(epoch)
```

4. 构建网络：

```
num_inputs = 784
num_outputs = 10
num_hiddens1 = 128
num_hiddens2 = 56
```

```
net =
nn.Sequential(nn.Flatten(),nn.Linear(num_inputs,num_hiddens1),nn.ReLU
(),nn.Dropout(0.0),

nn.Linear(num_hiddens1,num_hiddens2),nn.ReLU(),nn.Dropout(0.0),
                      nn.Linear(num_hiddens2,num_outputs))
net.apply(init_normal)
#net.apply(init_xavier)
```

5. 设置 epoch 和学习率，进行训练：

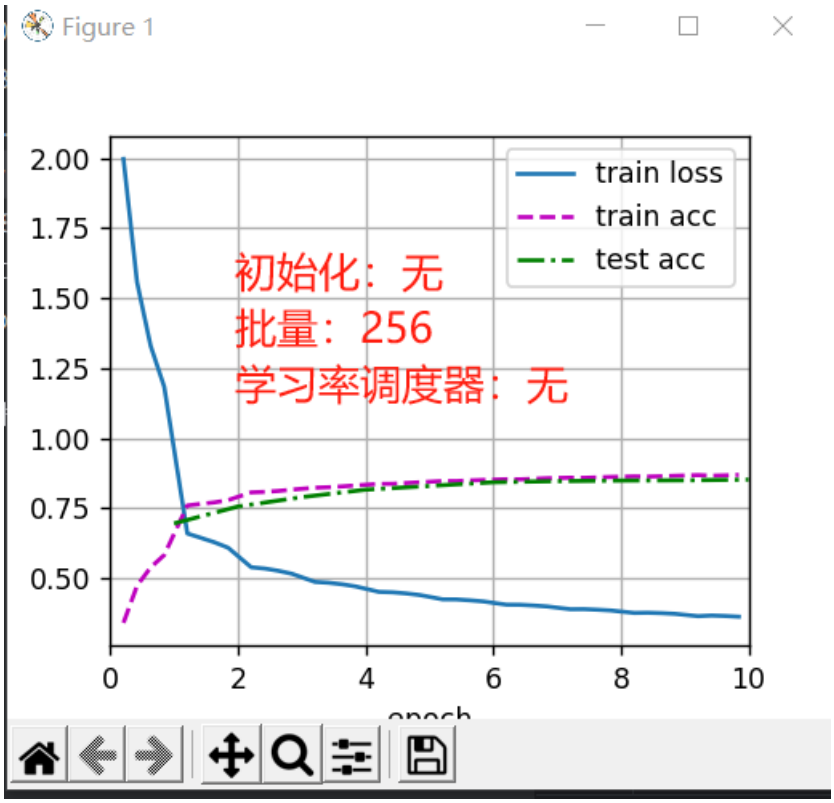
```
loss = nn.CrossEntropyLoss()
num_epochs = 10
lr = 0.1
trainer = torch.optim.SGD(net.parameters(), lr=lr)
train(net, train_dataloader, test_dataloader, num_epochs, loss,
trainer, device)
d2l.plt.show()
```

五、 实验的 3 个要求和 6 个附加题

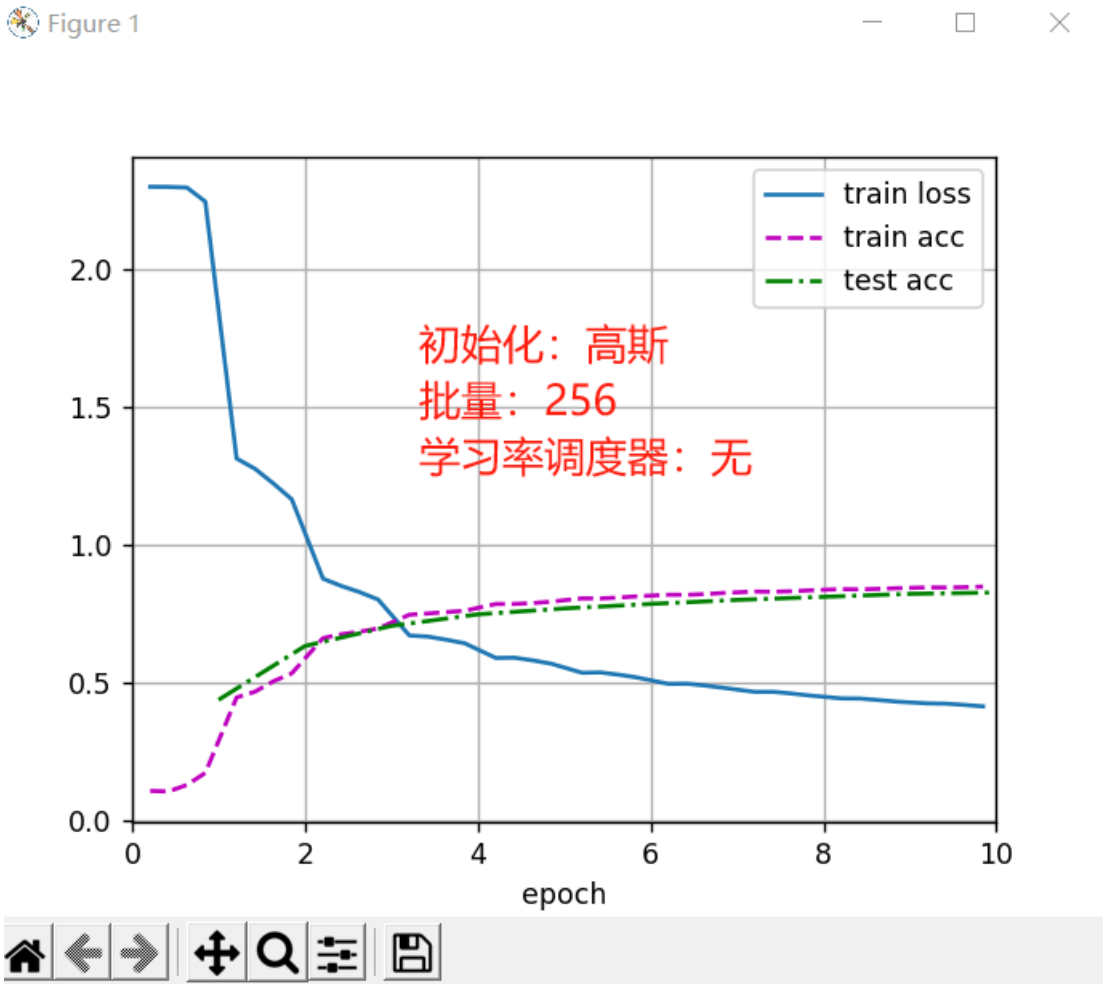
要求 1：使用 SGD（11.5 节） 训练模型，调试下面参数，分别画出损失函数、训练误差、测试误差随迭代 epoch 的变化曲线。

- 初始值：高斯分布随机初始化和 Xavier 初始化（4.8.2 节和 5.2.2 节）
- 批量大小：尝试不同批量大小 s，包括 s=样本数，s=1，及其他大小，尝试画出 11.5.4 节最后一个图，找出最合适的批量大小
- 学习率：尝试不同的学习率策略（11.11.3 节），包括固定学习率、指数衰减、分段学习率、多项式衰减、线性衰减、余弦衰减、预热

当不采用特别的初始化方式时：

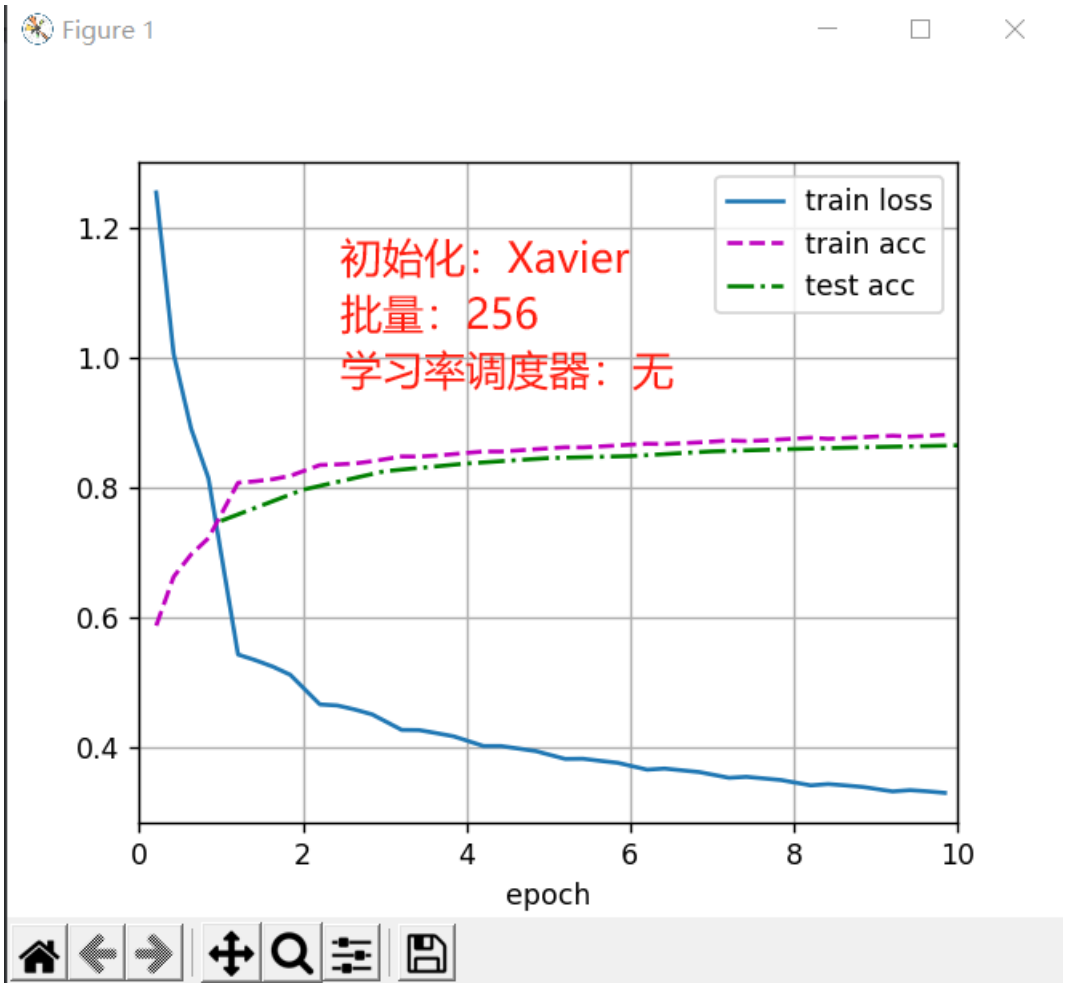


先采用高斯初始化，批量为 256：



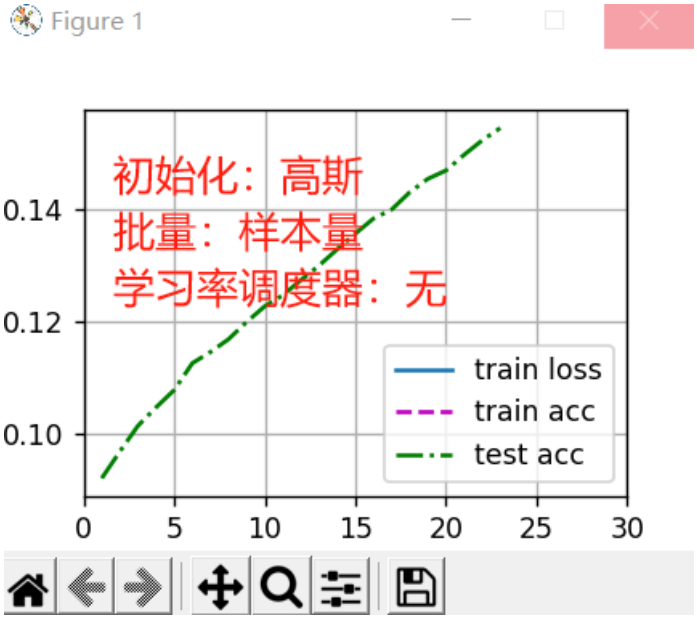
可以看到采用了高斯初始化之后，初始阶段的损失函数反而下降的更慢。推测这是因为告诉初始化对于本次的训练任务并不很好适配。

采用 Xavier 初始化：

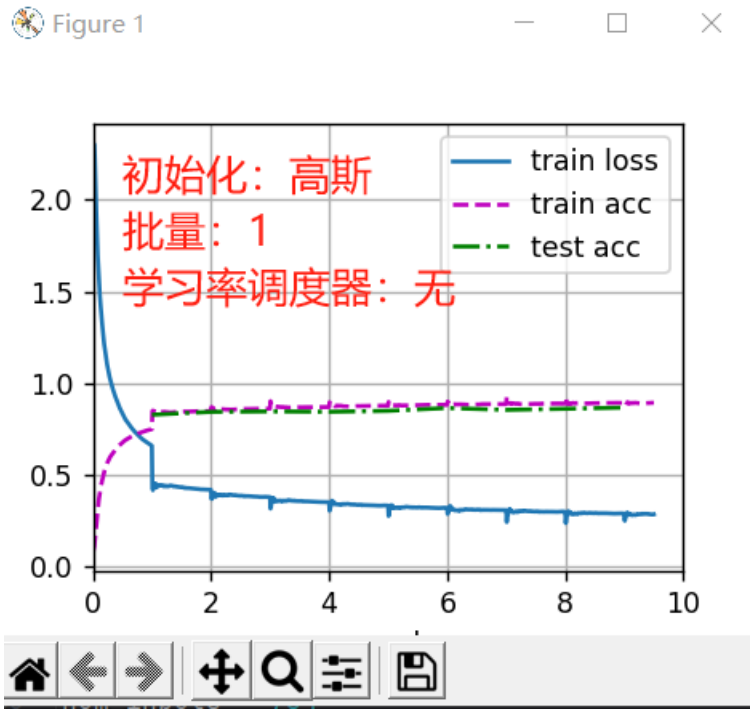


可以看到此时的收敛速度比高速初始化快了一些，但是和不采用初始化策略相差不大。推测是因为本任务中，多层感知机的单层神经元数量本身就很大（784），所以前期收敛效果较好。接下来调整批量大小，为了观察到更明显的现象，之后采用高斯初始化：

尝试把批量大小设置为样本量大小：



再将批量设置为 1：



虽然批量为 1 看起来能够达到较好的学习效果，但是在实际运行中，速度非常慢，因此并不适合在实践中使用。经过调试，仍选择 256 为合适的批量大小。

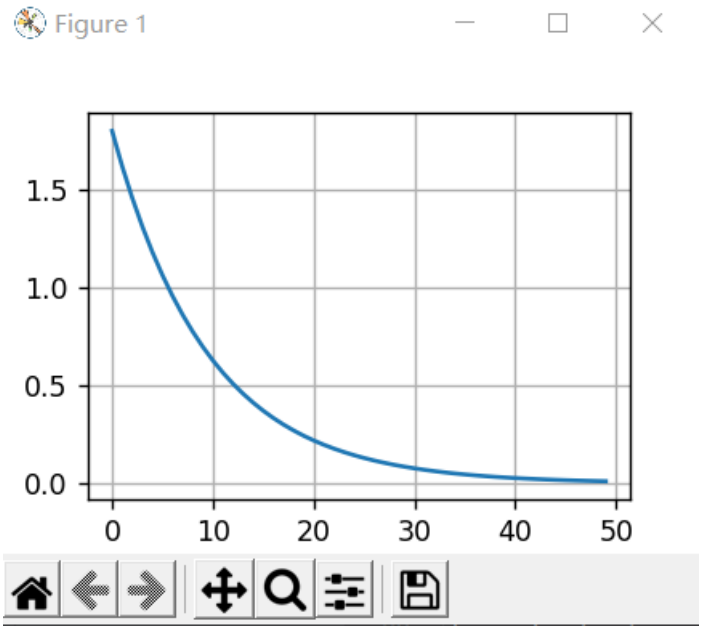
接下来加入学习率调度器：

加入指数衰减调度器：

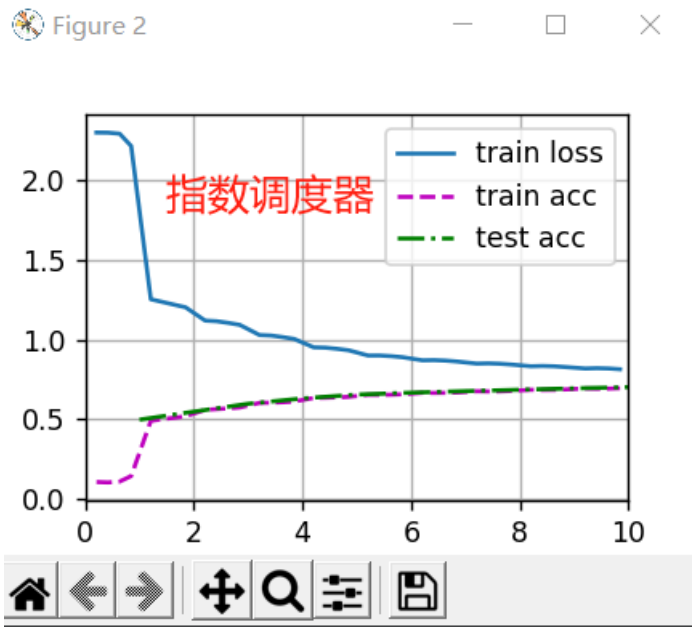
代码:

```
class FactorScheduler:
    def __init__(self, factor=1, stop_factor_lr=1e-7, base_lr=0.1):
        self.factor = factor
        self.stop_factor_lr = stop_factor_lr
        self.base_lr = base_lr

    def __call__(self, num_update):
        self.base_lr = max(self.stop_factor_lr, self.base_lr *
self.factor)
        return self.base_lr

scheduler = FactorScheduler(factor=0.9, stop_factor_lr=1e-2,
base_lr=2.0)
```

学习率的变化:



运行结果:

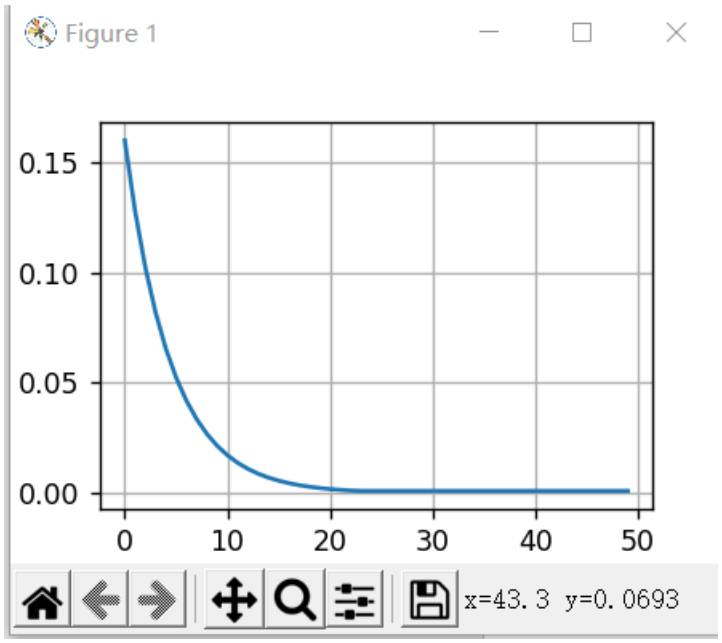


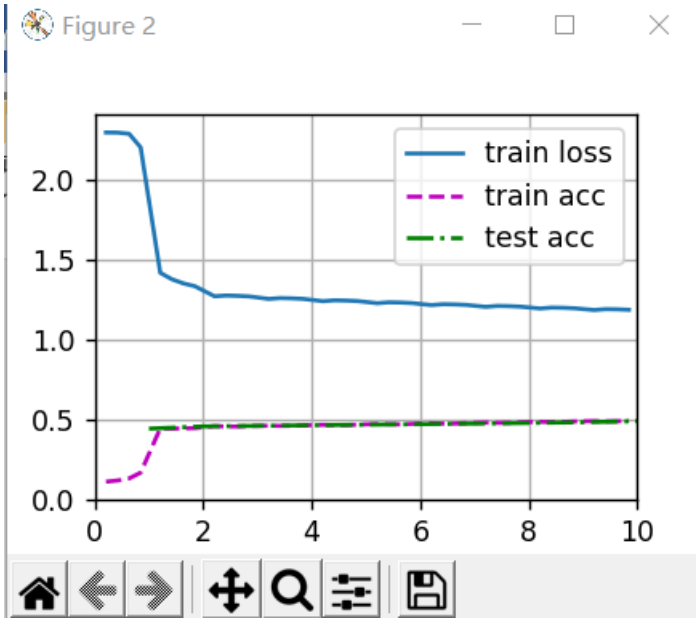
可以看到，损失函数先是很快下降，然后下降的速率变得缓慢。推测这是因为学习率过早得下降，导致训练时的学习率小于理想需求的学习率。

同时，将参数改为：

```
scheduler = FactorScheduler(factor=0.8, stop_factor_lr=1e-3, base_lr
= 0.2)
```

则会出现如下现象：

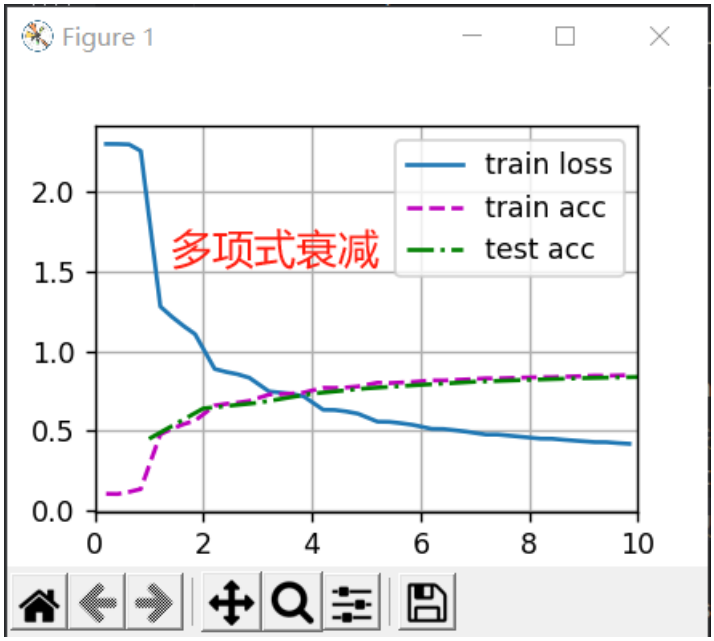




训练损失和准确率在到达一定值之后，便几乎稳定，不再变化。目前还不了解这种现象出现的原因。

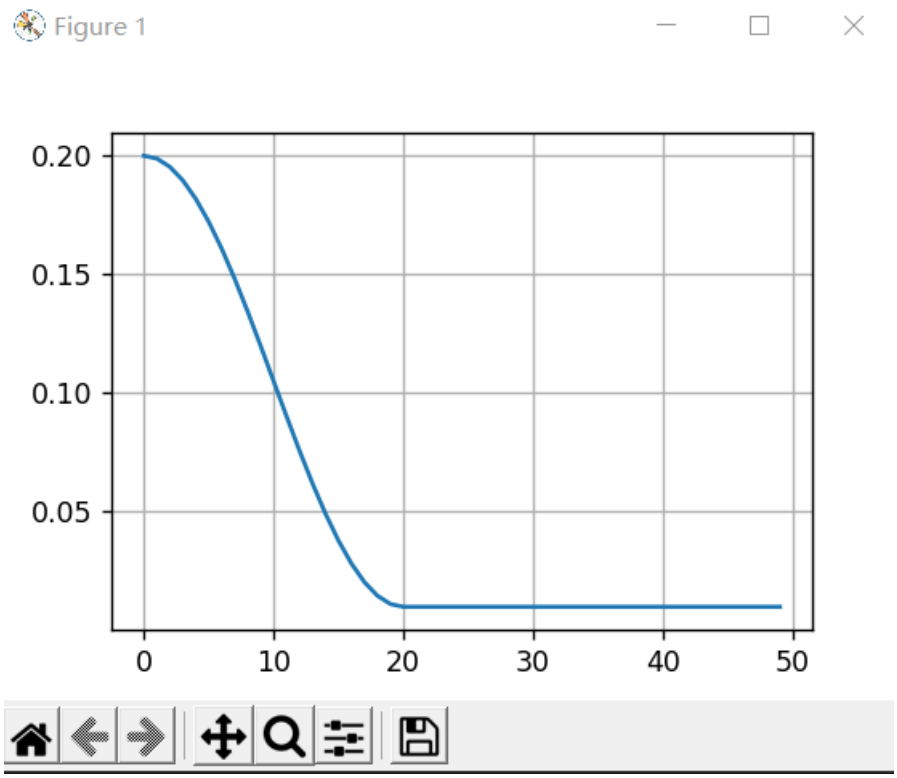
使用多项式衰减调度器：

```
scheduler = lr_scheduler.MultiStepLR(trainer, milestones=[15, 30],
gamma=0.5)
```

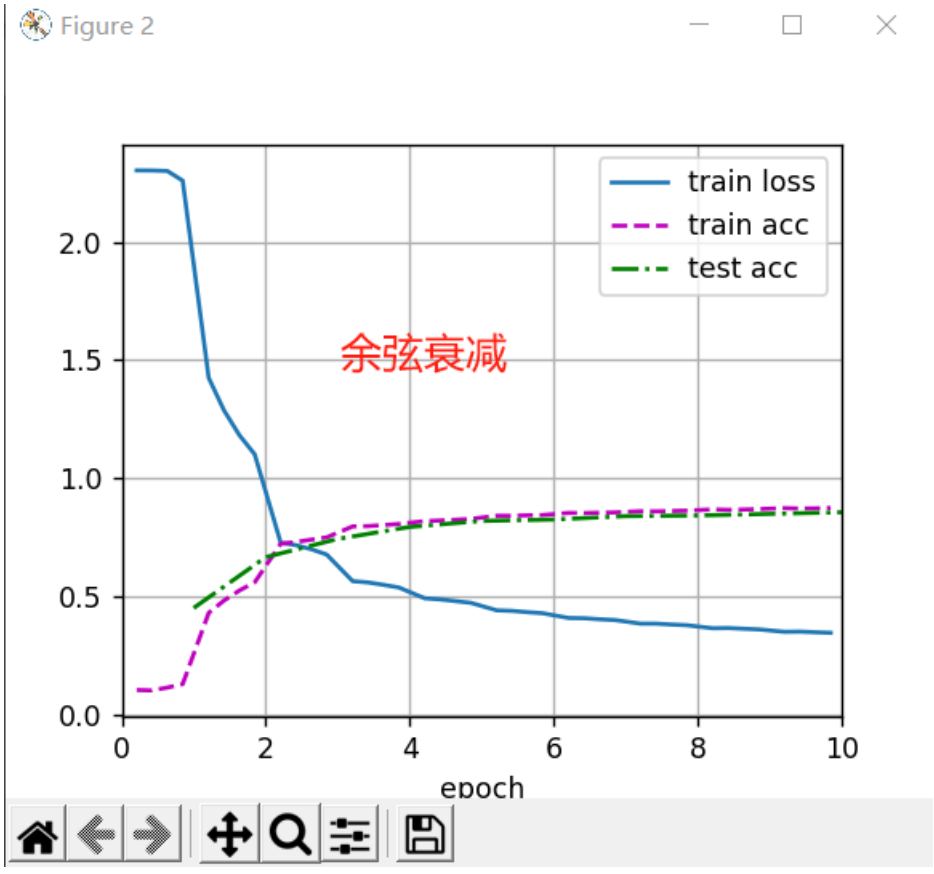


可以看到比单因子的效果更加稳定。

然后使用余弦衰减：

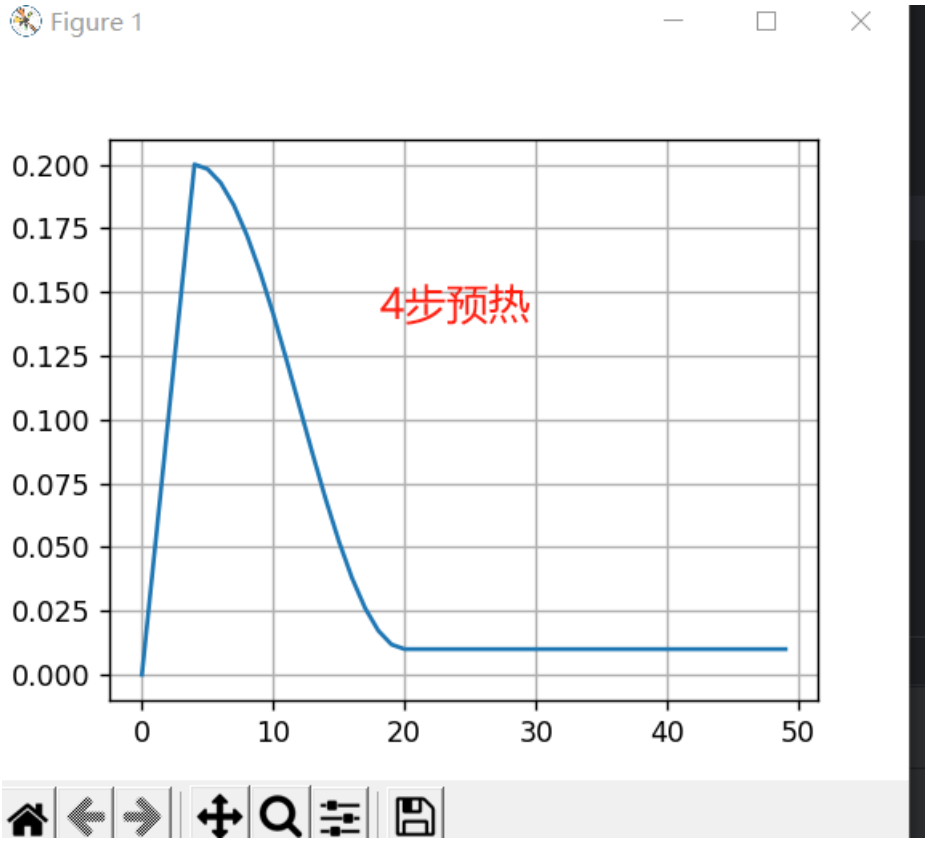


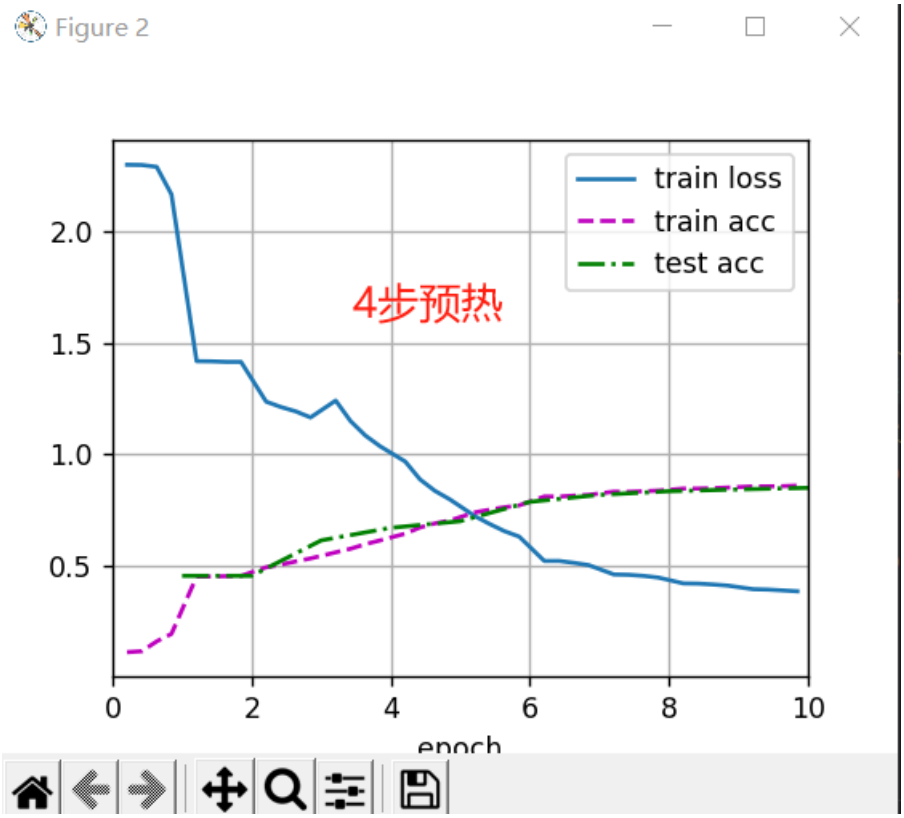
运行结果：



相较于前两个学习率调度器，余弦衰减更加稳定，最终效果也更好。

接下来在余弦衰减的基础上加入预热。因为本实验中，epoch 在很小就可以实现稳定，所以先采用 4 步预热：





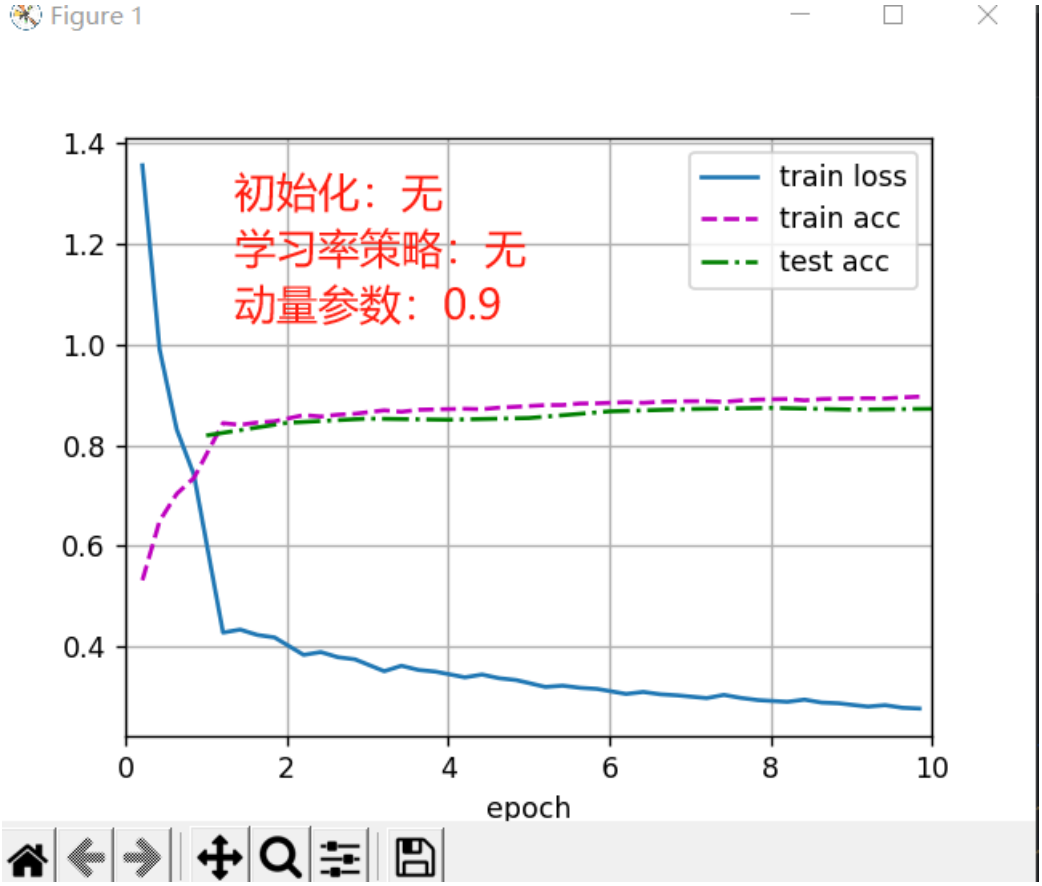
要求 1 的总结：通过上述的操作发现，一些学习率调度器、预热技术的加入，反而不如没有使用时的分类效果。推测是因为本次实验是把图片展平成了一个一维张量，然后采用线性网络进行分类；而不是想常规的那样使用卷积网络对图像进行处理。

要求 2：使用 SGD+动量法（11.6 节） 训练模型，调试要求 1 中初始值和学习率，画出相应曲线

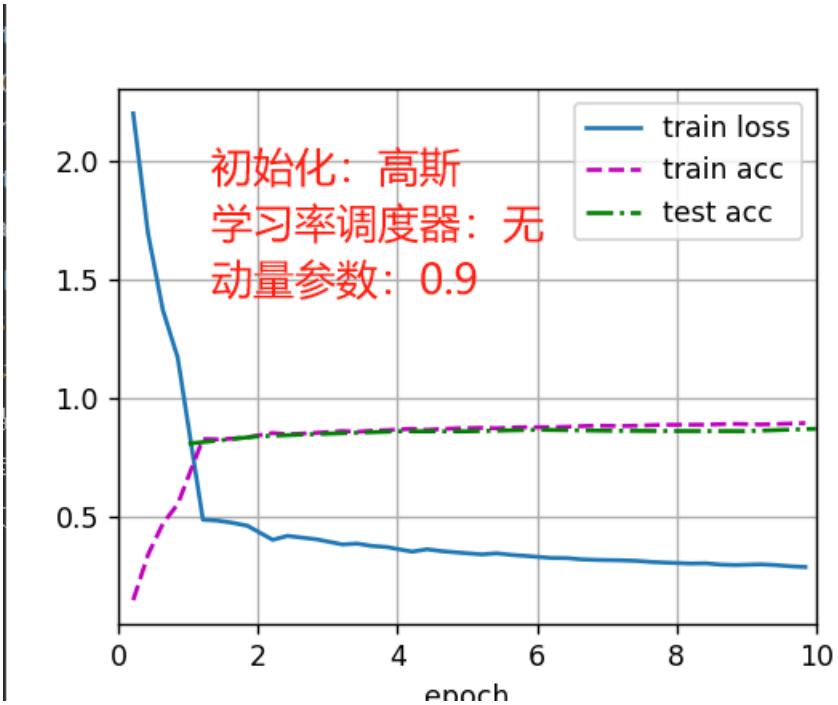
添加动量参数，并且设置动量参数为 0.9， 代码为：

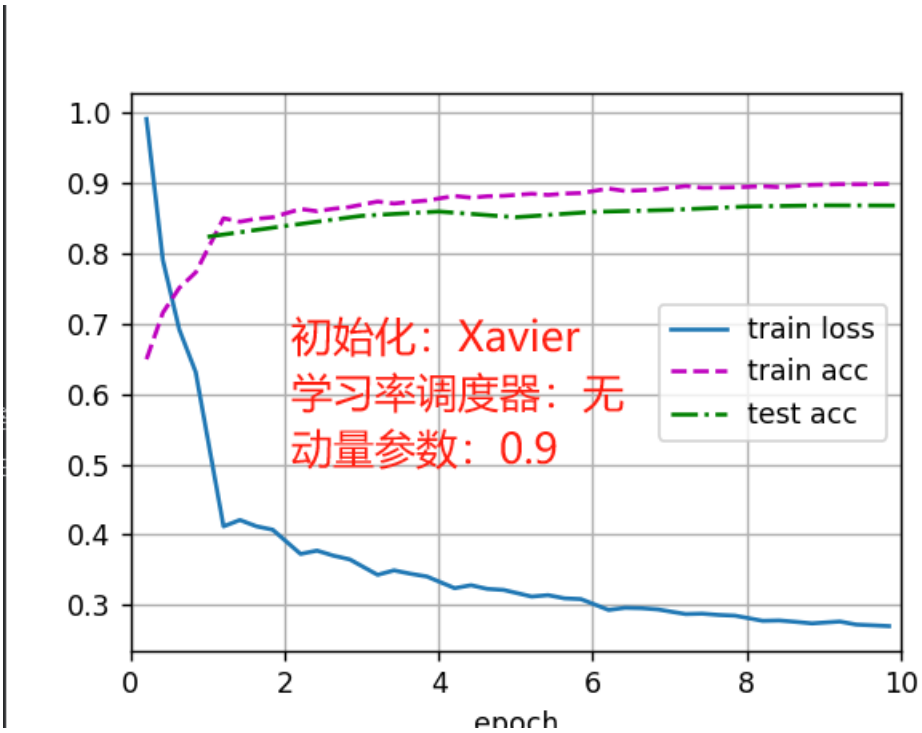
```
trainer = torch.optim.SGD(net.parameters(), lr=lr,momentum=0.9)
```

运行结果：

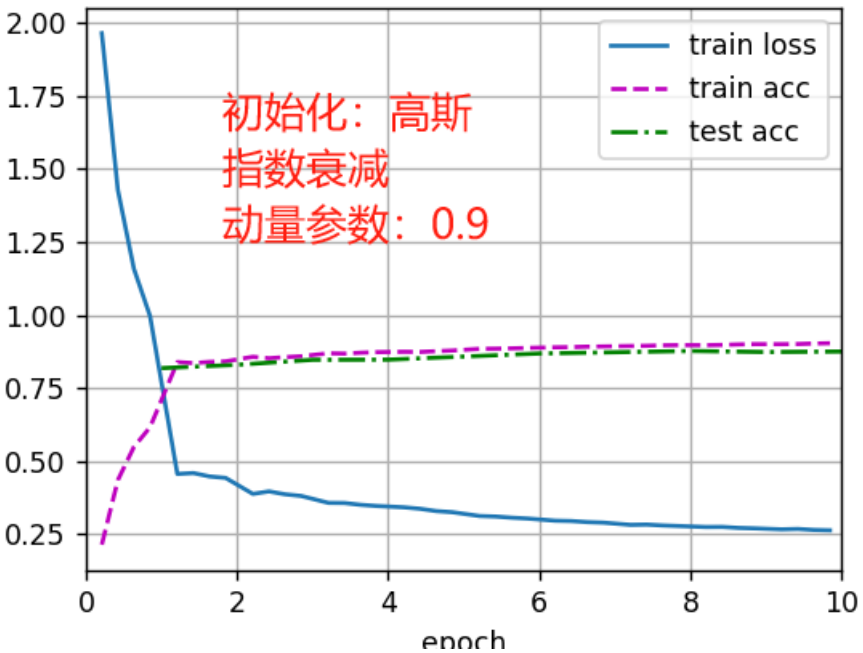


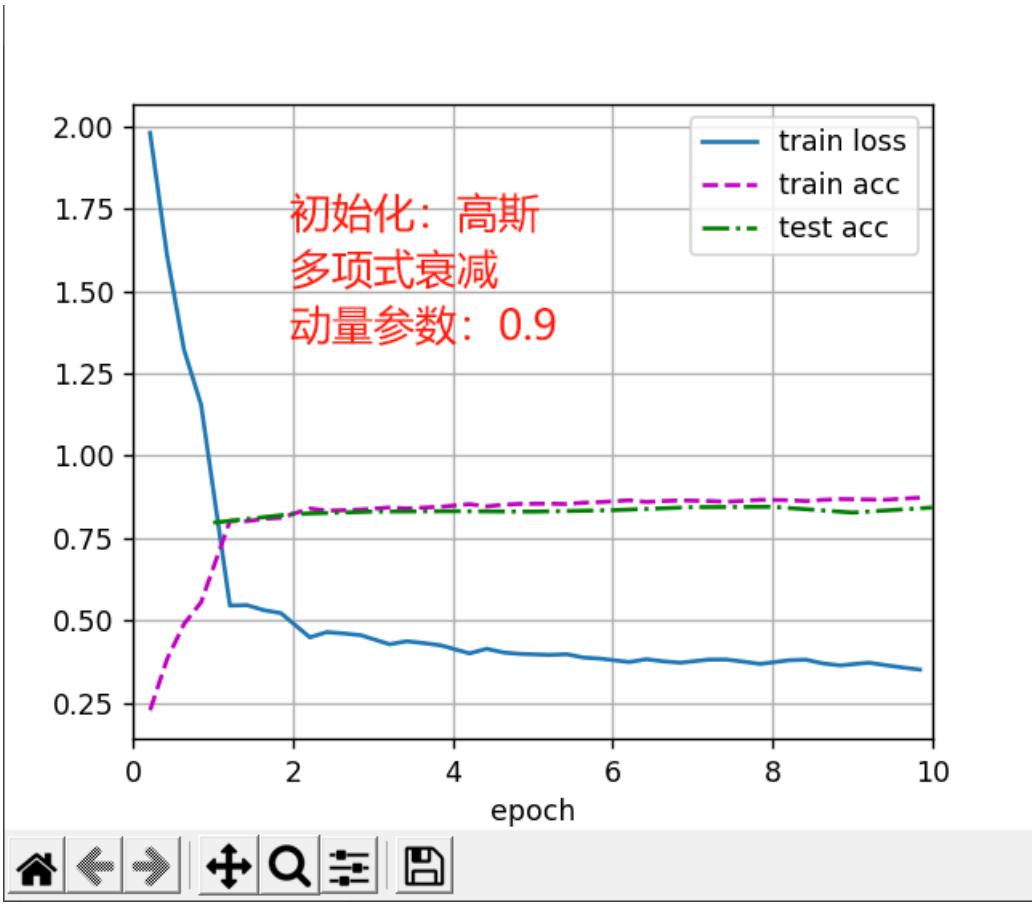
然后进行高斯初始化和 Xavier 初始化的对比：

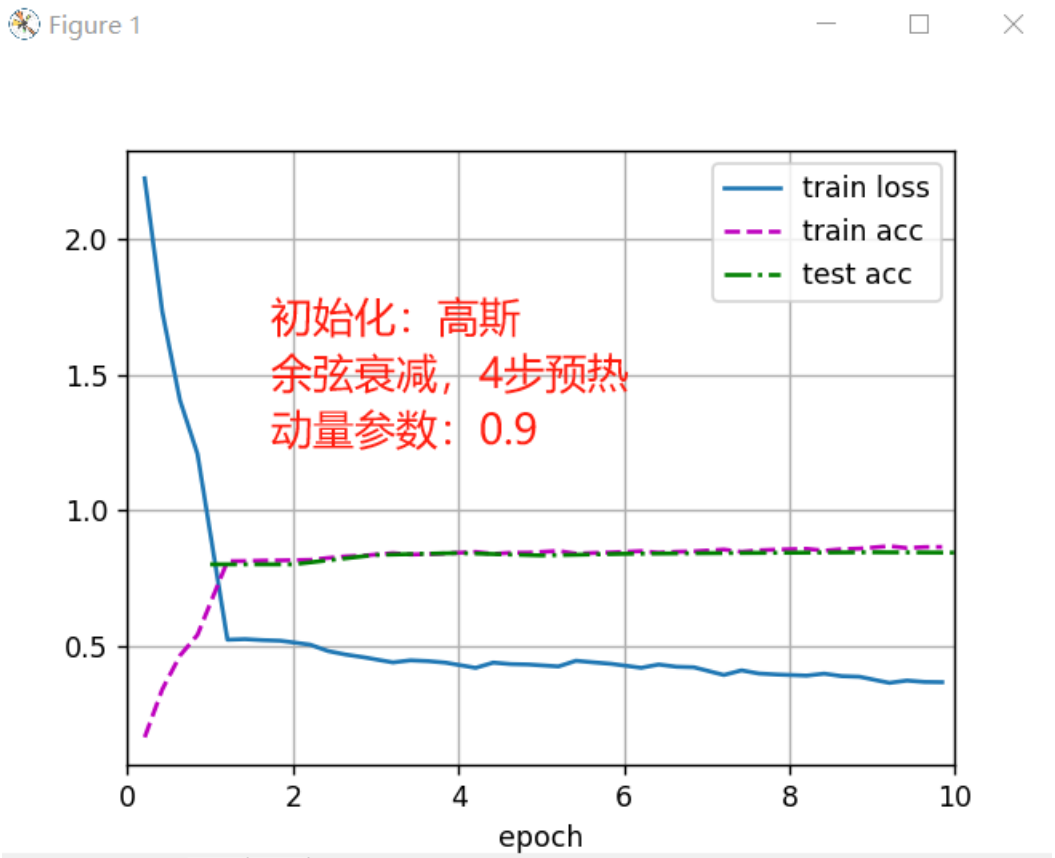


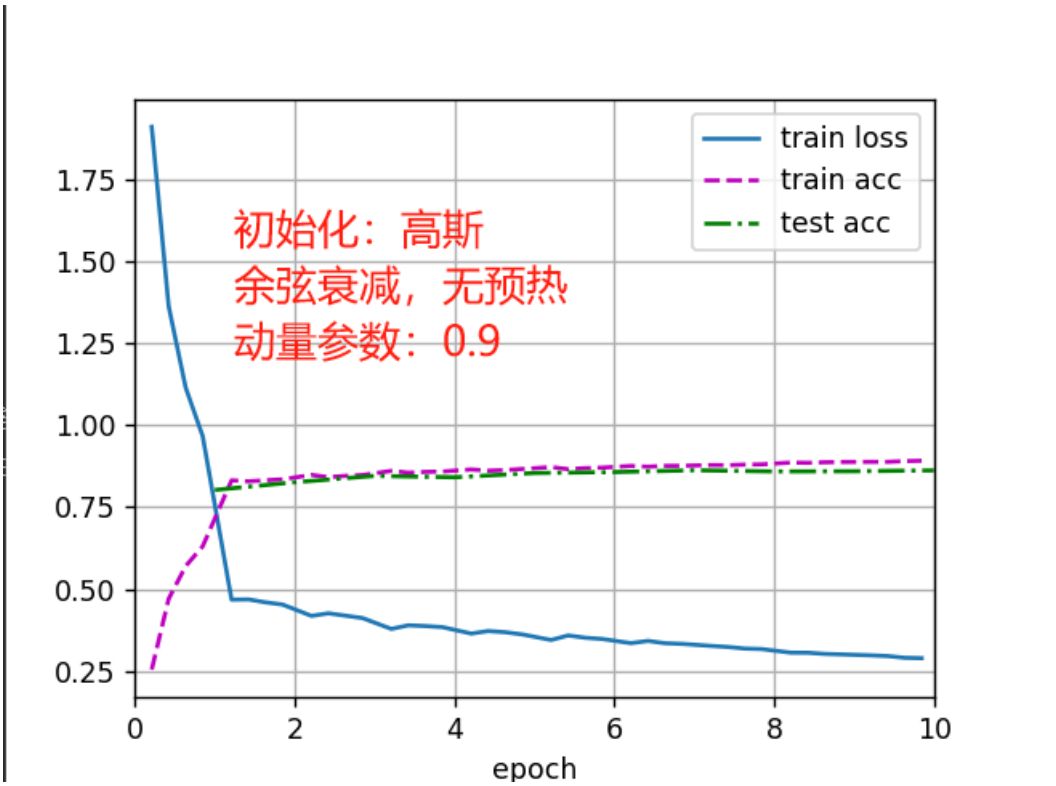


可以看到，相较于随机初始化和 Xavier 初始化，高斯初始化在保证了准确率的情况下，很好地缓解了过拟合现象的发生，因此采用高斯初始化。
然后进行学习率调度器的选用：


train loss
train acc
test acc
初始化：高斯
指数衰减
动量参数：0.9
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0
2
4
6
8
10
epoch


train loss
train acc
test acc
初始化：高斯
多项式衰减
动量参数：0.9
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0
2
4
6
8
10
epoch





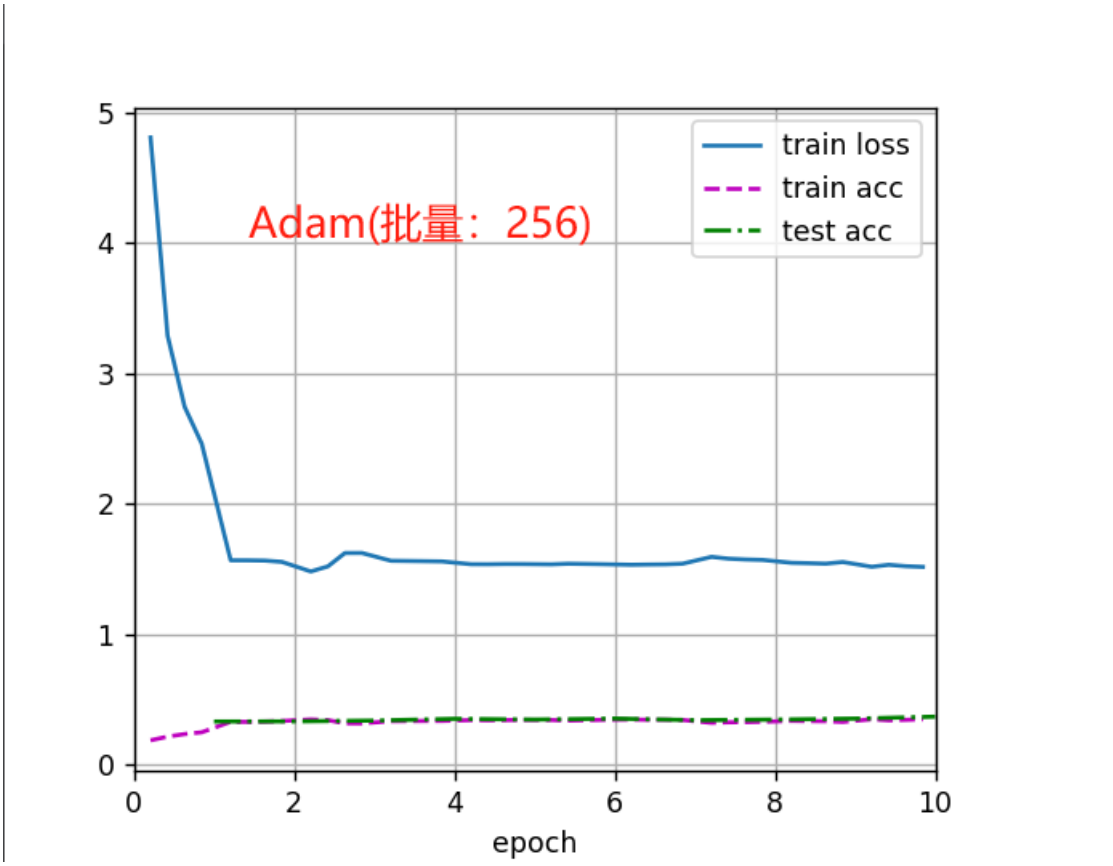
可以看到，各个学习率调度器的准确率其实相差不大，但是相较于指数衰减、多项式衰减、无预热余弦衰减，只有 4 步预热余弦衰减可以有效地抑制过拟合现象。因此，我们选择 4 步预热余弦衰减。

综上所述，我们最终找到的最佳参数配置为：高斯初始化，4 步预热余弦衰减，动量参数为 0.9。

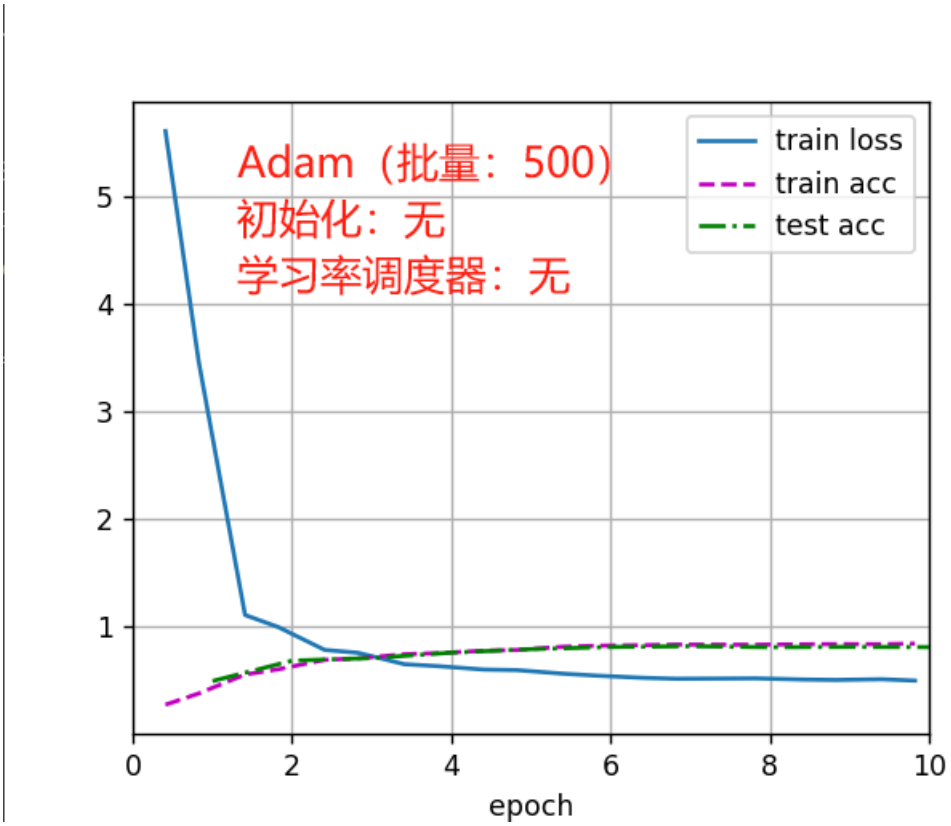
要求 3：使用 Adam（11.10 节） 训练模型，调试要求 1 中初始值和学习率，画出相应曲线

更换为 Adam 迭代器：

```
trainer = torch.optim.Adam(params=net.parameters(),lr=lr)
```

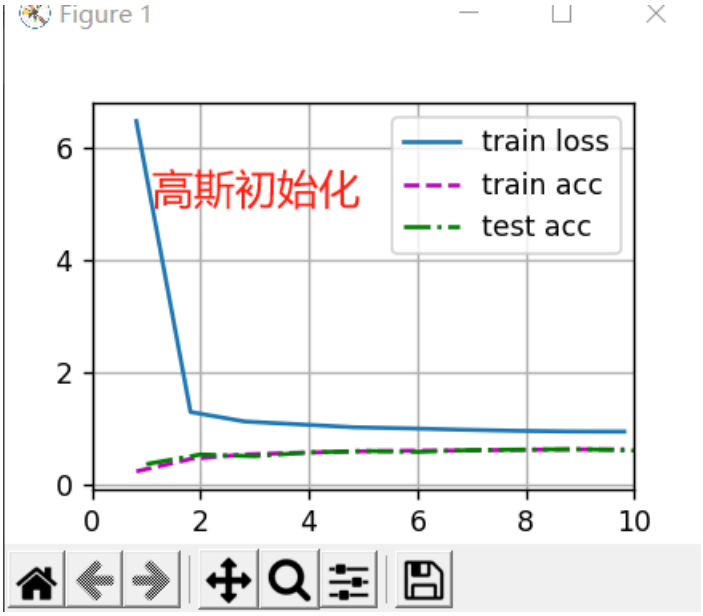


此时发现，在将 SGD 更换为 Adam 后，训练和预测的效果都大打折扣。查询了相关资料得知，Adam 算法在进行大规模并行计算时，效果不好。因此为了较少并行计算数量，我将原有的 bacth size 从 256 调整为 500，运行结果为：

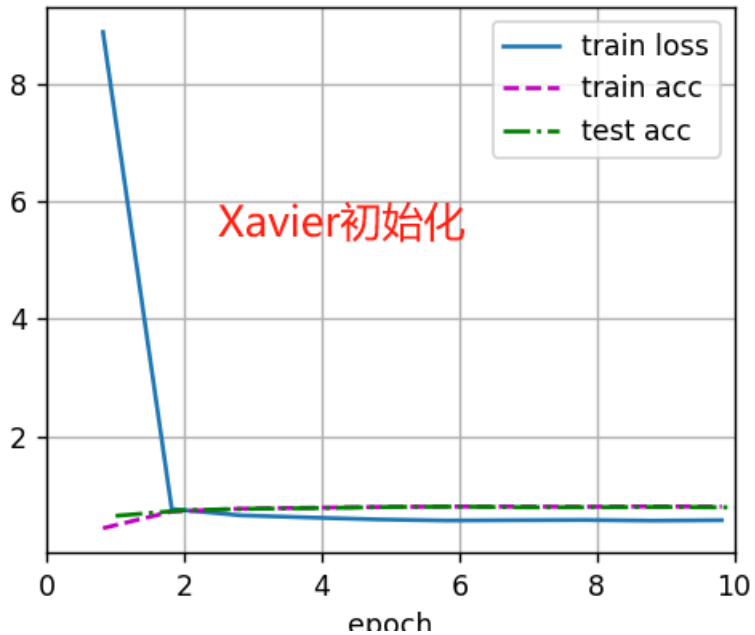


可以看到，在将批量增大之后，Adam 算法才不至于效果那么差劲。接下来正式寻找合适的初始化方式和学习率调度器。

首先考察初始化方式：

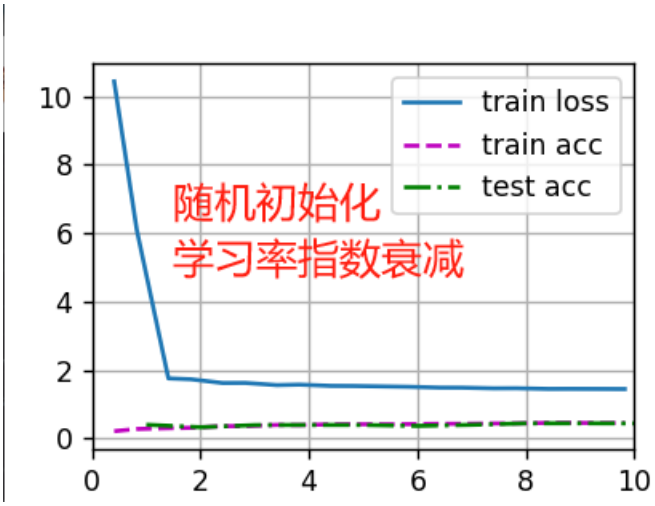


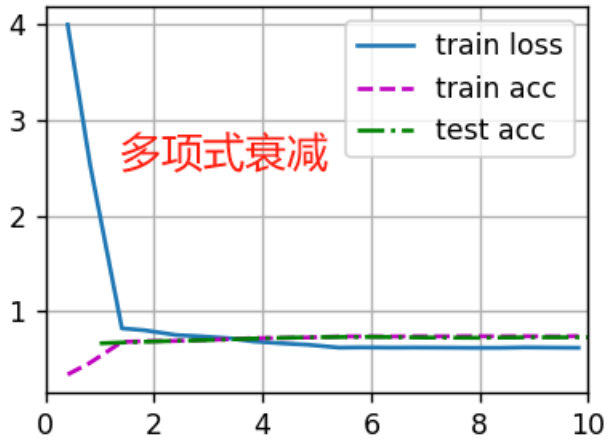
采用高斯初始化后，最终训练和测试准确率在 0.6 左右



采用 Xavier 初始化后，其最终训练和预测准确率为 0.799 和 0.784，虽然比高斯初始化高一些，但是依然不如最初的随机初始化。因此最终采用随机初始化。

接下来选择学习率调度器：




train loss
train acc
test acc
多项式衰减
0
1
2
3
4
0
2
4
6
8
10

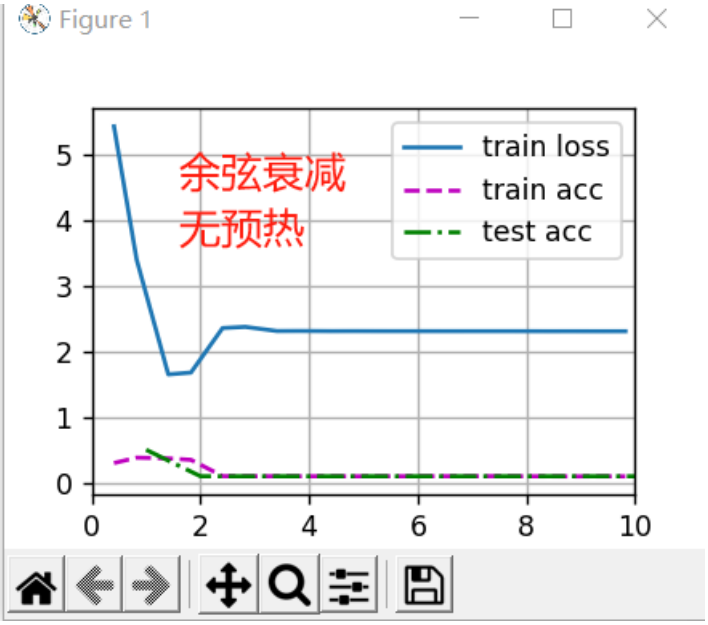


Figure 1
train loss
train acc
test acc
余弦衰减
无预热
0
1
2
3
4
5
0
2
4
6
8
10

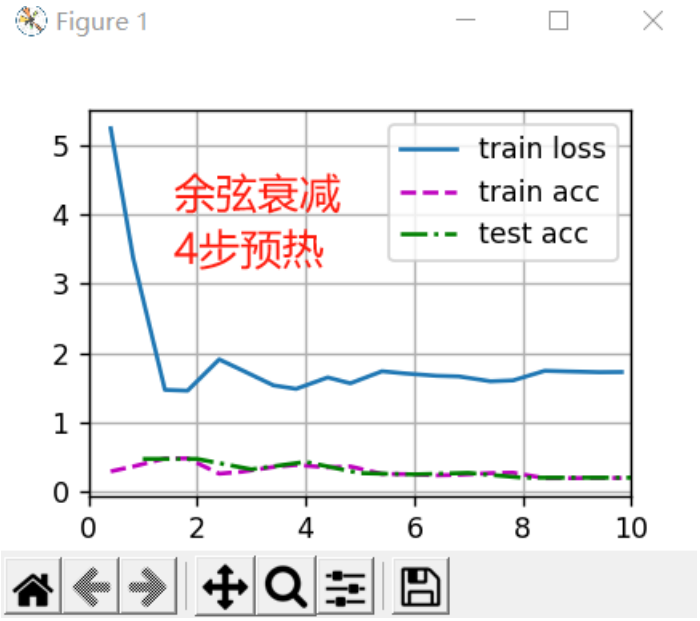


Figure 1
train loss
train acc
test acc
余弦衰减
4步预热
0
1
2
3
4
5
0
2
4
6
8
10

由上述各个结果图分析，只有多项式衰减能保持相对较好的训练预测结果，而其他几个学习率调度器则出现了较差的学习效果。其中无预热余弦衰减更是损失函数不降反增，准确率不增反降，推测是因为学习率没能及时减小，而产生了一次震荡。

六道附加题：

要求 1：从并行计算的角度解释不同批量大小对算法训练速度的影响
首先对程序加上计时器：

```
start = time.time()
end = time.time()
print('用时：',end - start)
```

同时，将模型设置为随机初始化，SGD 优化器，学习率无衰减。
首先将批量设置为 256，运行：

```
<Figure size 437x312 with 1 Axes>
<Figure size 437x312 with 1 Axes>
<Figure size 437x312 with 1 Axes>
train loss 0.359, train acc 0.870, test acc 0.854
用时： 240.21639895439148
```

运行时长为 240.2 秒

然后将批量设置为 500，运行：

```
<Figure size 437x312 with 1 Axes>
<Figure size 437x312 with 1 Axes>
<Figure size 437x312 with 1 Axes>
<Figure size 437x312 with 1 Axes>
<Figure size 437x312 with 1 Axes>
train loss 0.421, train acc 0.851, test acc 0.833
用时： 61.950169801712036
```

运行时长 61.95 秒

然后将批量设置为 1000，运行：

```
<Figure size 437x312 with 1 Axes>
<Figure size 437x312 with 1 Axes>
<Figure size 437x312 with 1 Axes>
<Figure size 437x312 with 1 Axes>
<Figure size 437x312 with 1 Axes>
train loss 0.518, train acc 0.816, test acc 0.806
用时： 54.431206941604614
```

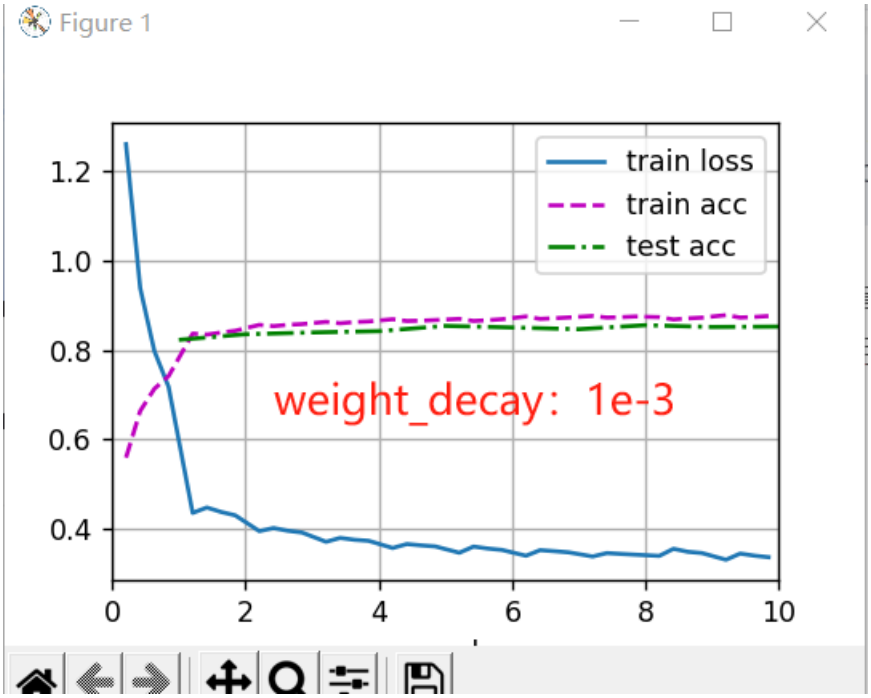
运行时长为 54.43 秒。
由此可以看出，随着批量的增大，算法的运行速度也增大，但是存在边界递减效应，也就是算法运行的加速度在减小。
因为批量越大，并行运算数增多，而串行运算数减小；也就是说，批量越大，gpu 的工作越多，cpu 的工作越少。而深度学习中 gpu 的速度要远远高于 cpu，也就是说运算瓶颈是由 cpu 决定的，因此才会出现我们观察到的现象。

- 要求 2：在 SGD+动量法 中，进一步尝试如下情况
- 权重衰减：尝试不同的权重衰减率
- 动量参数：尝试不同动量参数，观察是否比 API 默认的值更好
- Nesterov 加速法：进一步尝试该策略，观察是否有效

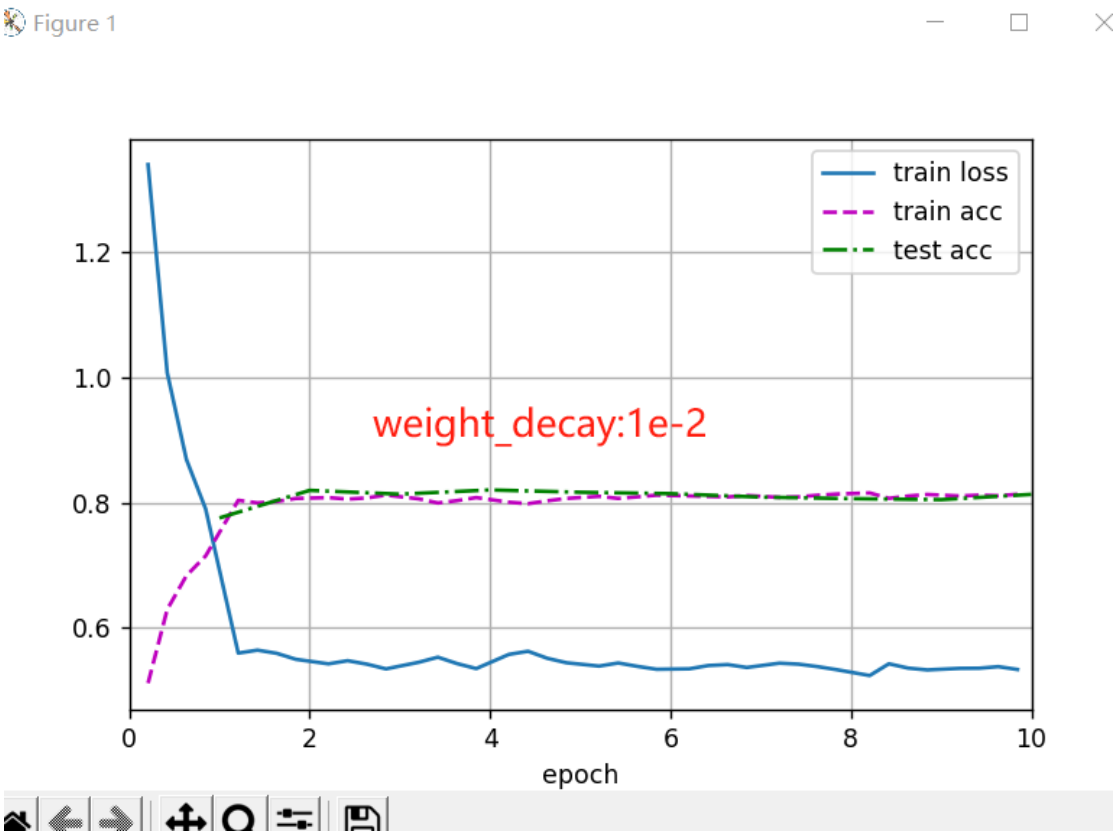
首先尝试权重衰减率，将其设置为 1e-3：

```
trainer = torch.optim.SGD(net.parameters(),
lr=lr,momentum=0.9,weight_decay=1e-3)
```

进行训练：



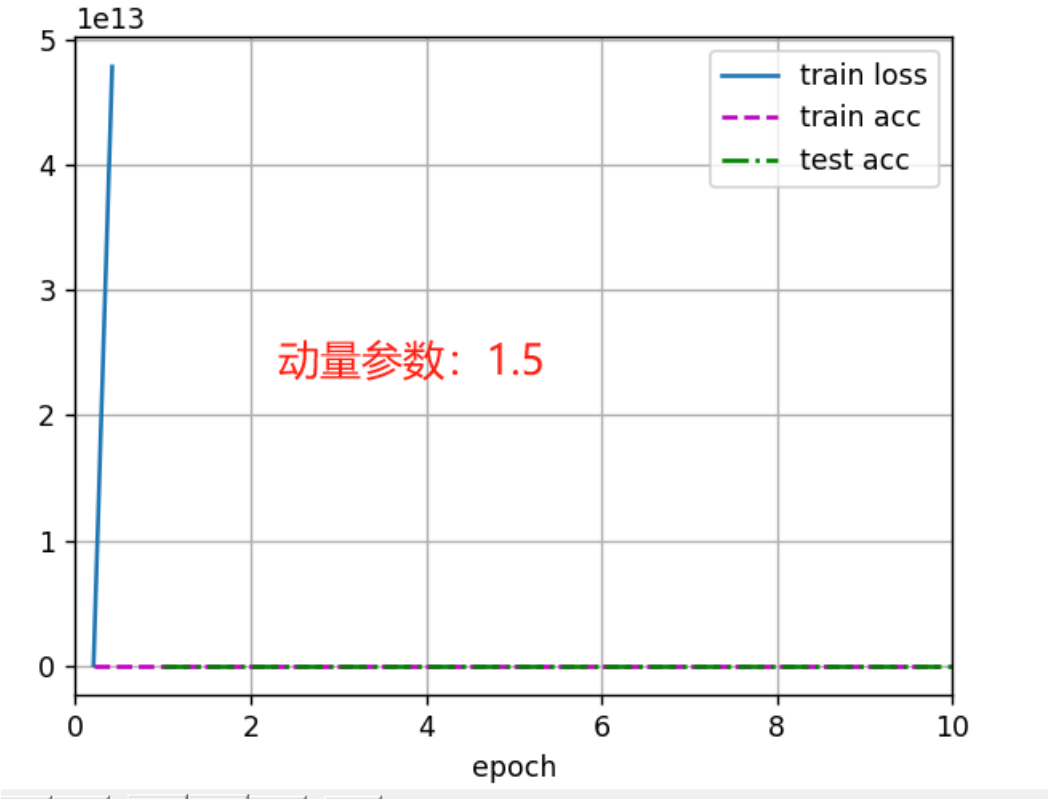
其中 train_acc 为 0.87，test_acc 为 0.85。虽然二者都较高，但是发生了明显的过拟合现象。
接下来把 weight_decay 设置为 1e-2：



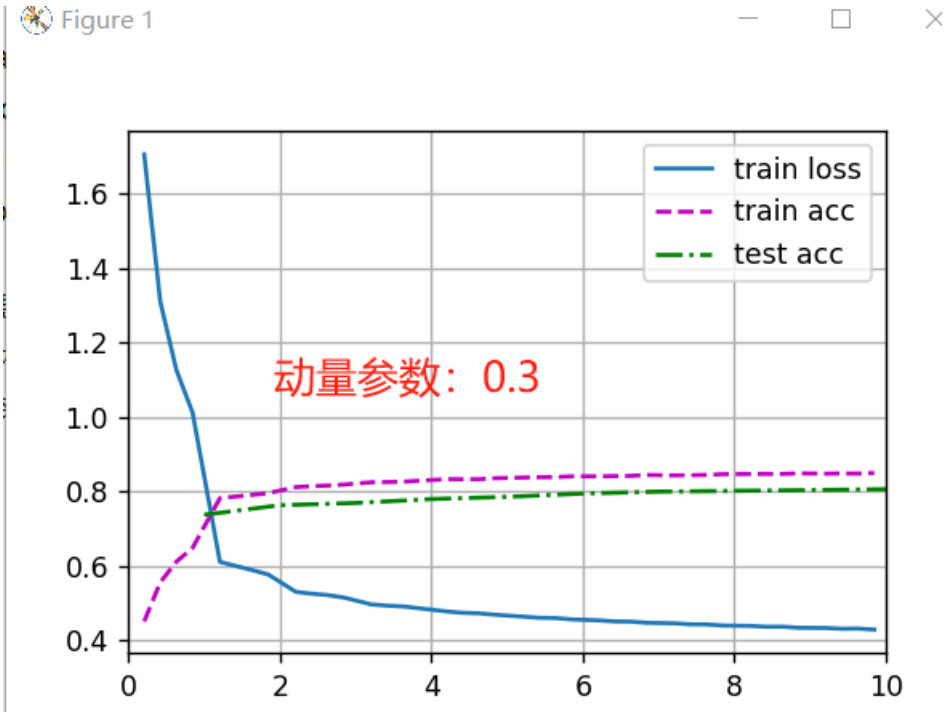
可以观察到，虽然增大 weight_decay 参数确实使得过拟合现象得到明显减弱，但是相应地，发生了模型“光顾着减小复杂度“的现象，因为 train_acc 和 test_acc 都明显降低。
因此再取一个 1e-3 和 1e-2 的中位数：5.5e-3，运行：

果然，设置为 5.5e-3 后，既保留了对过拟合现象的良好抑制，也避免了准确率的明显下滑。
因此我们最终选择 weight_decay 为 5.5e-3.

调整动量参数：
将默认的 0.9 改为 1.5：



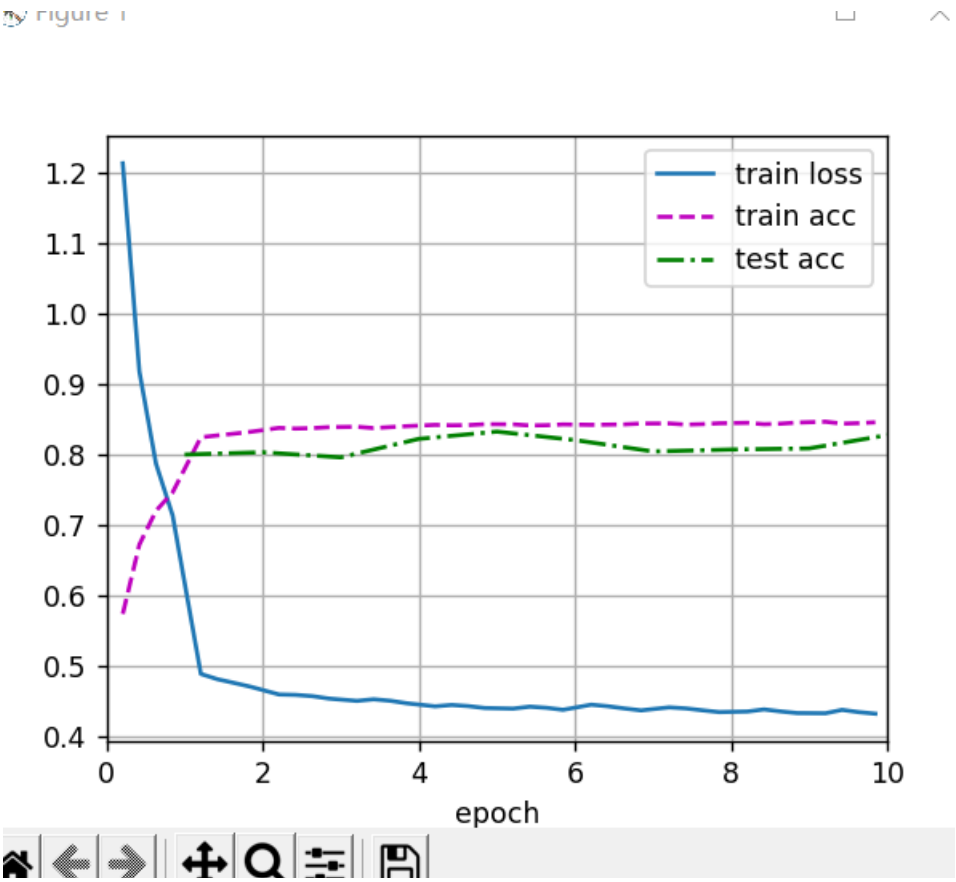
可以看到过高的动量参数使得优化算法更新力度过大，无法完成正常的学习。
再将参数设置为 0.3：



可见，过小的动量参数，使得过拟合现象再次出现，并且最终的准确率也有所下降。因此我们选择默认动量参数：0.9.

我们使用 Nesterov 加速法：

```
trainer = torch.optim.SGD(net.parameters(),
lr=lr,momentum=0.9,weight_decay=5.5e-3,nesterov=True)
```

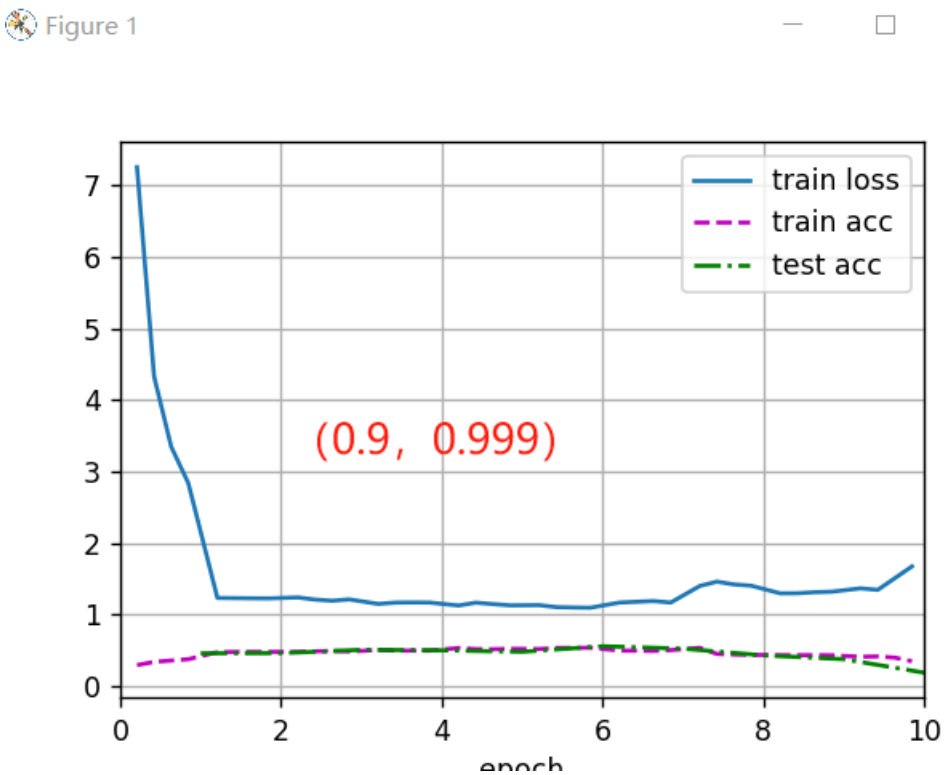


可见，使用 Nesterov 加速后，模型虽然会更快的收敛，但是过拟合现象也更加明显。

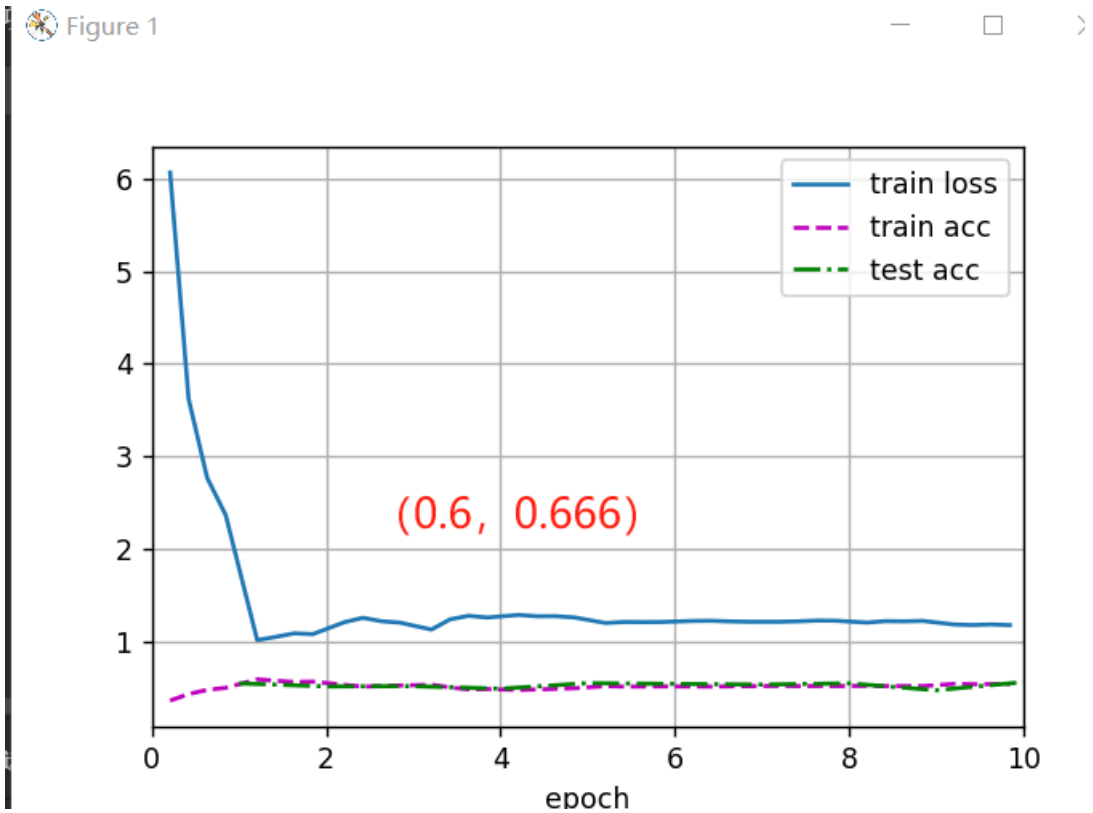
- 要求 3：在 Adam 中，进一步尝试如下情况
- 动量参数：尝试不同动量参数，观察是否比 API 默认的值更好
- AMSGrad：进一步尝试该策略，观察是否有效

首先使用常用的动量参数（0.9，0.999）：

```
trainer =
torch.optim.Adam(params=net.parameters(),lr=lr,betas=(0.9,0.999))
```

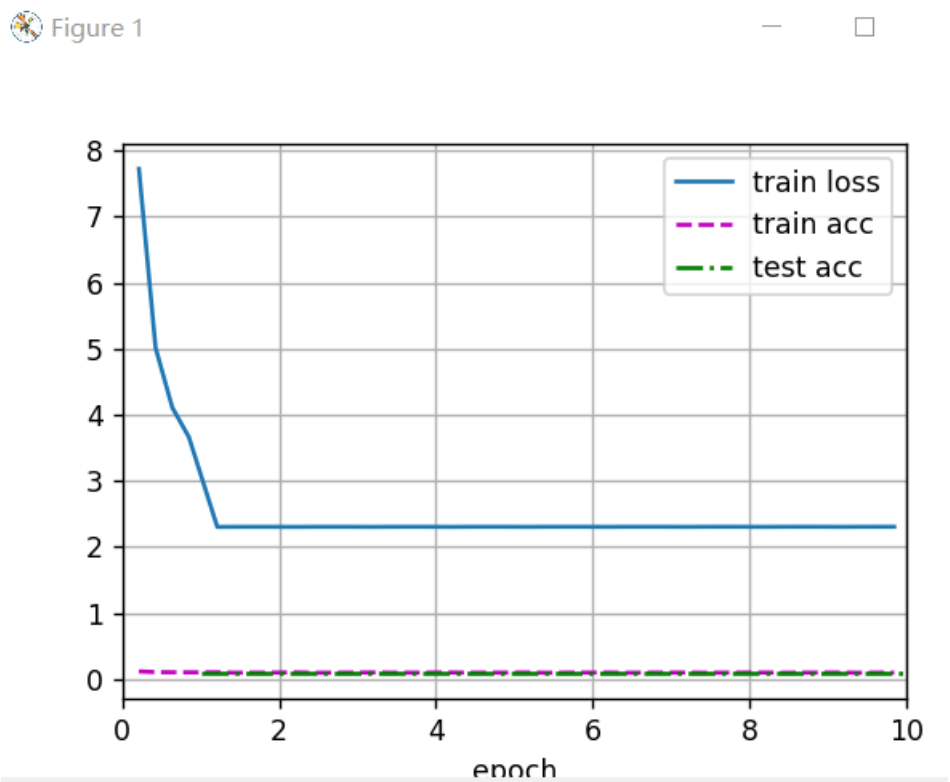


可以看到效果依然有限，虽然过拟合现象得到很好抑制，但是最终准确率和损失函数的表现都不佳。再将其设置为（0.6，0.666）：



观察到训练和测试准确率更加稳定，但是最终上限并不算高。
接下来使用 AMSGrad 方法：

```
trainer =
torch.optim.Adam(params=net.parameters(),lr=lr,amsgrad=True)
```

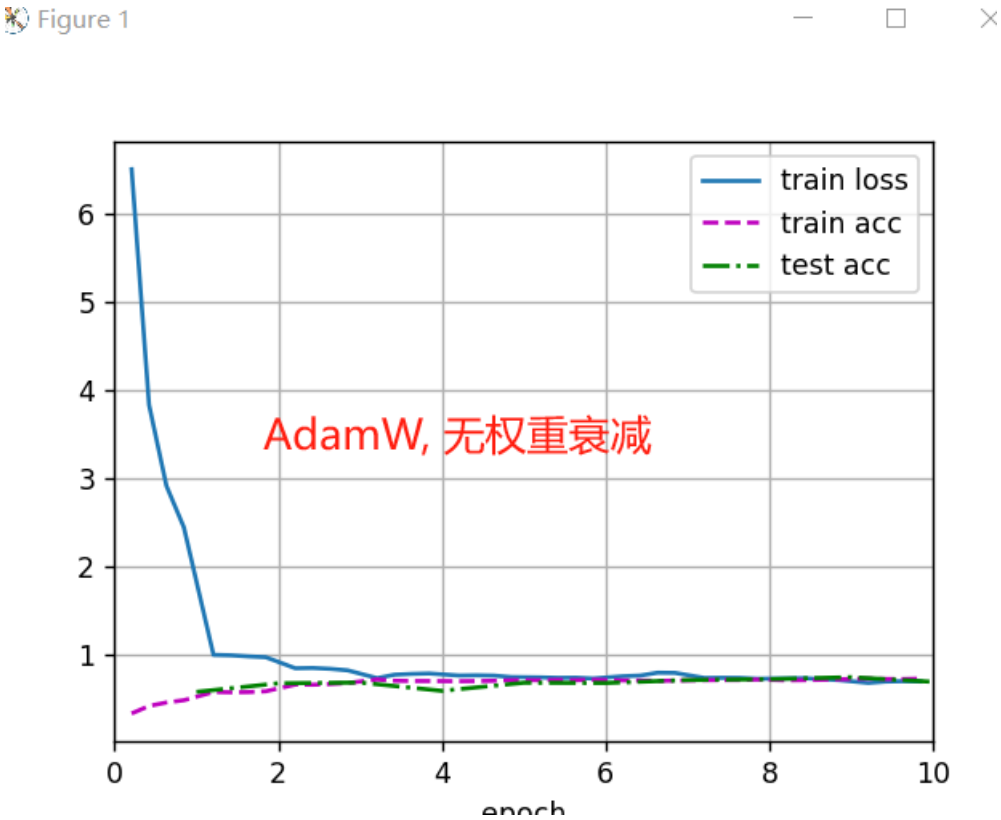


可见，Adam 和 amsgrad 方法对于本次任务和模型并不适配。

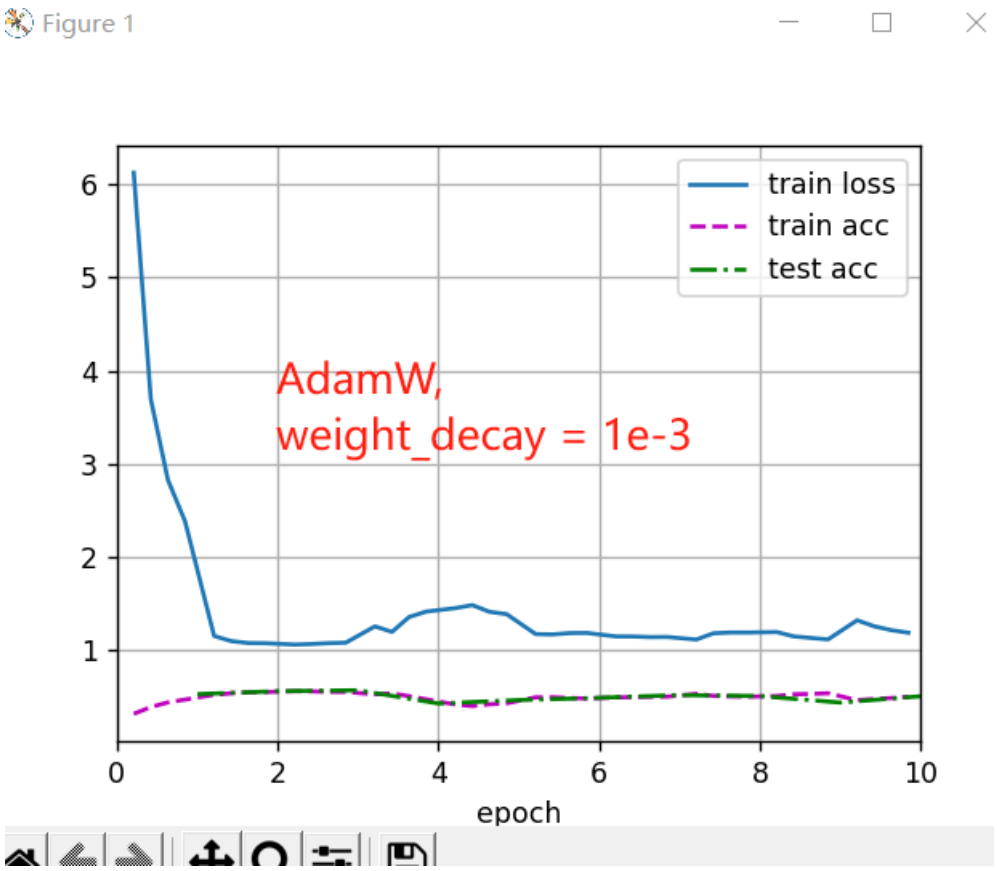
- 要求 4：使用 AdamW 训练模型，从权重衰减的角度比较与 Adam 不同

首先查看无权重衰减时，AdamW 的效果：

```
trainer = torch.optim.AdamW(params=net.parameters(),lr=lr)
```



可见，即使没有加入权重衰减，AdamW 也仍然能较好地抑制过拟合现象。现在将其权重衰减参数设置为 1e-3:



与 Adam 优化器相比，效果会更好一些。

在原始的 Adam 优化器中，权重衰减（weight decay）是在梯度更新中直接应用的，它等价于在损失函数中添加了一个 L2 正则化项，用于减小权重的幅度。这意味着权重衰减对所有权重参数都是一样的，而 AdamW 是对 Adam 的改进，它将权重衰减与梯度更新分开处理。具体来说，权重衰减只应用于权重参数，而不应用于偏置参数。这是通过在损失函数中添加一个额外的 L2 正则化项来实现的，而不是直接应用于梯度更新。

要求 5：画出梯度随迭代 epoch 的变化曲线，解释训练缓慢的原因，是因为鞍点？局部极小点？还是其他？

因为整个网络的参数较多，我选择对第一层中的某些 weight 参数的梯度进行观察。首先在训练函数中添加如下代码：

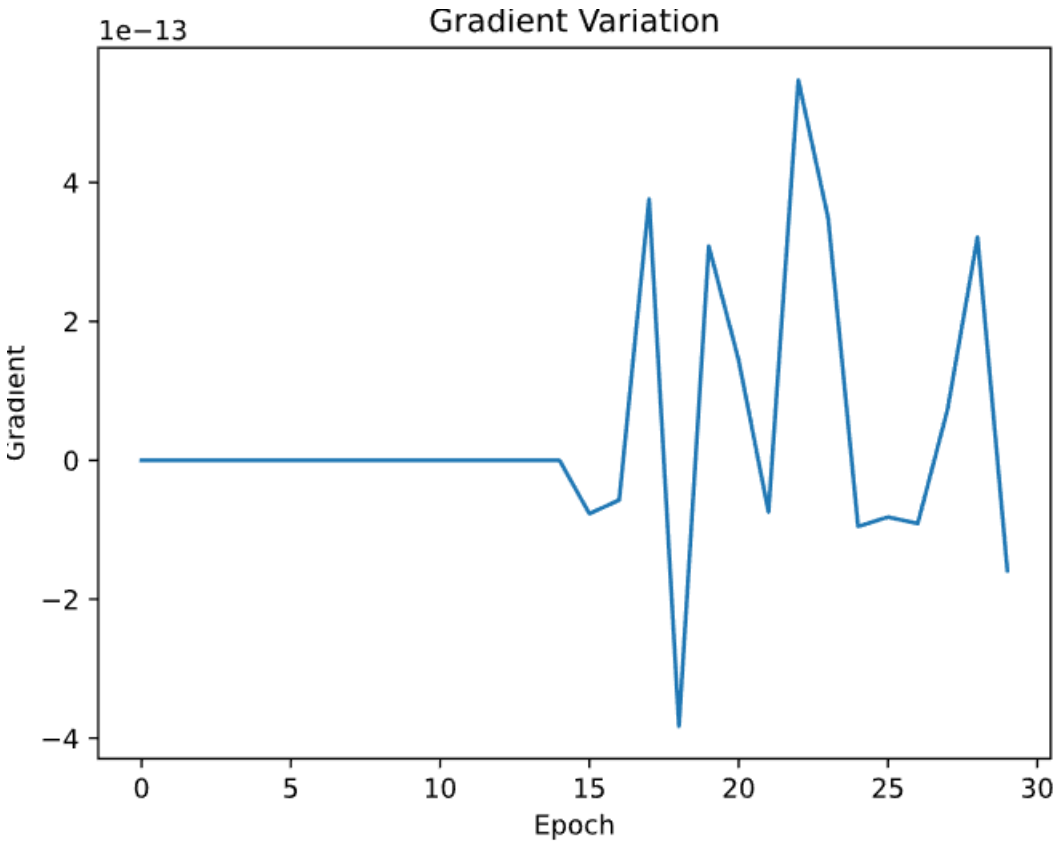
```
gradients = []
        for param in net[1].parameters():
            gradients.append(param.grad.detach().cpu().numpy())
        gradient_values.append(gradients)
```

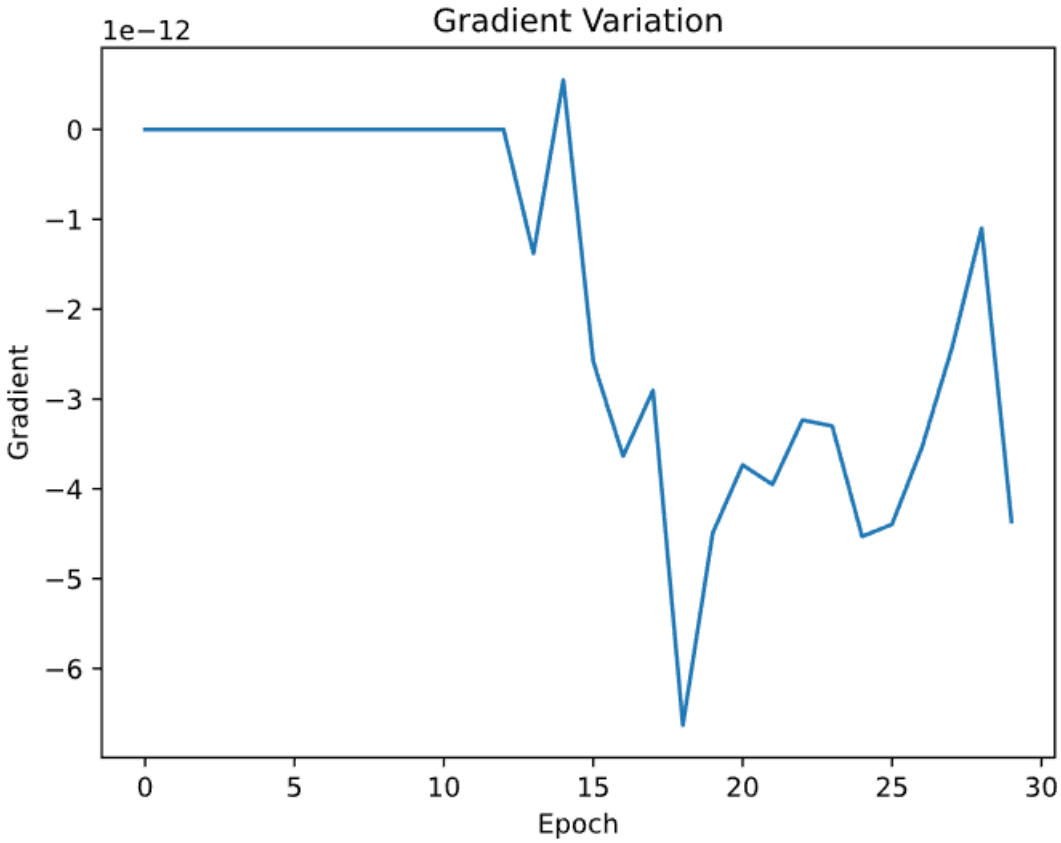
这将把第一个隐藏层的所有参数：（784 x 128）个 w，和 128 个 b，的梯度全部保存到 gradient_values 列表中。然后先进行 w 参数的观察：

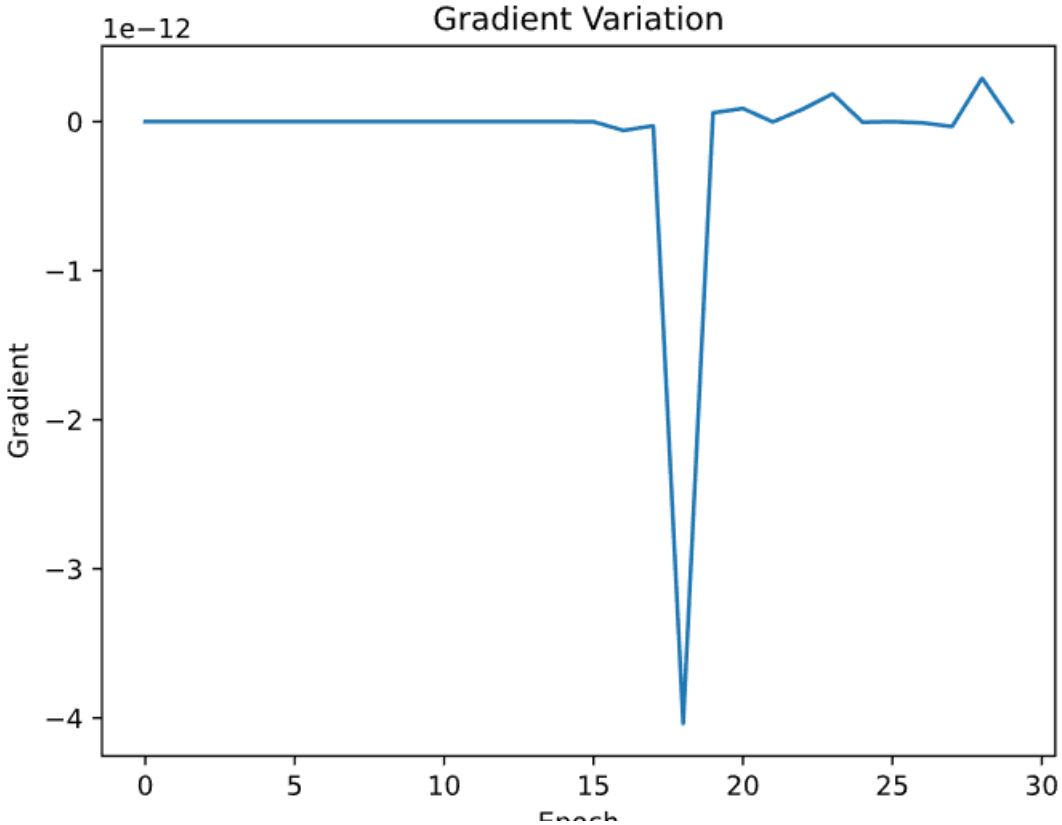
```
gv = []
for i in range(num_epochs):
    a = gradient_values[i][0][0][725]
    gv.append(a)

print(gv)
print(gv)
    # 绘制梯度随迭代 epoch 的变化曲线
plt.plot(range(0,num_epochs),gv)
plt.xlabel('Epoch')
plt.ylabel('Gradient')
plt.title('Gradient Variation')
plt.show()
```

在上述代码中，gradient_values 的最后一位索引 j 代表第 j 个参数，倒数第二个索引 p 取 0 或者 1，代表考察的是 weight(0)，还是 bias(1)。

选取 3 个 weight 参数的梯度进行查看：


Gradient Variation
1e−13
Gradient
4
2
0
−2
−4
0
5
10
15
20
25
30
Epoch


Gradient Variation
1e−12
Gradient
0
−1
−2
−3
−4
−5
−6
0
5
10
15
20
25
30
Epoch



可见，三者都在 epoch 等于 15 左右时，开始进行更新。同时第一个梯度在 0 附近做近似相等幅度的正负震荡，猜测是此时学习率恰好不能使得其收敛到最佳位置。
第二个梯度在开始更新后，一直为负，推测是此时的学习率对于它来说太小，因而 30epoch 之后也依然在朝着最优方向行进，未能到达最优点。
第三个梯度在 epoch 等于 18 时，进行了一次更新，之后梯度便一直在 0 附近小幅震荡，推测是因为此时的学习率能够使得它只进行一次更新便找到最优位置。

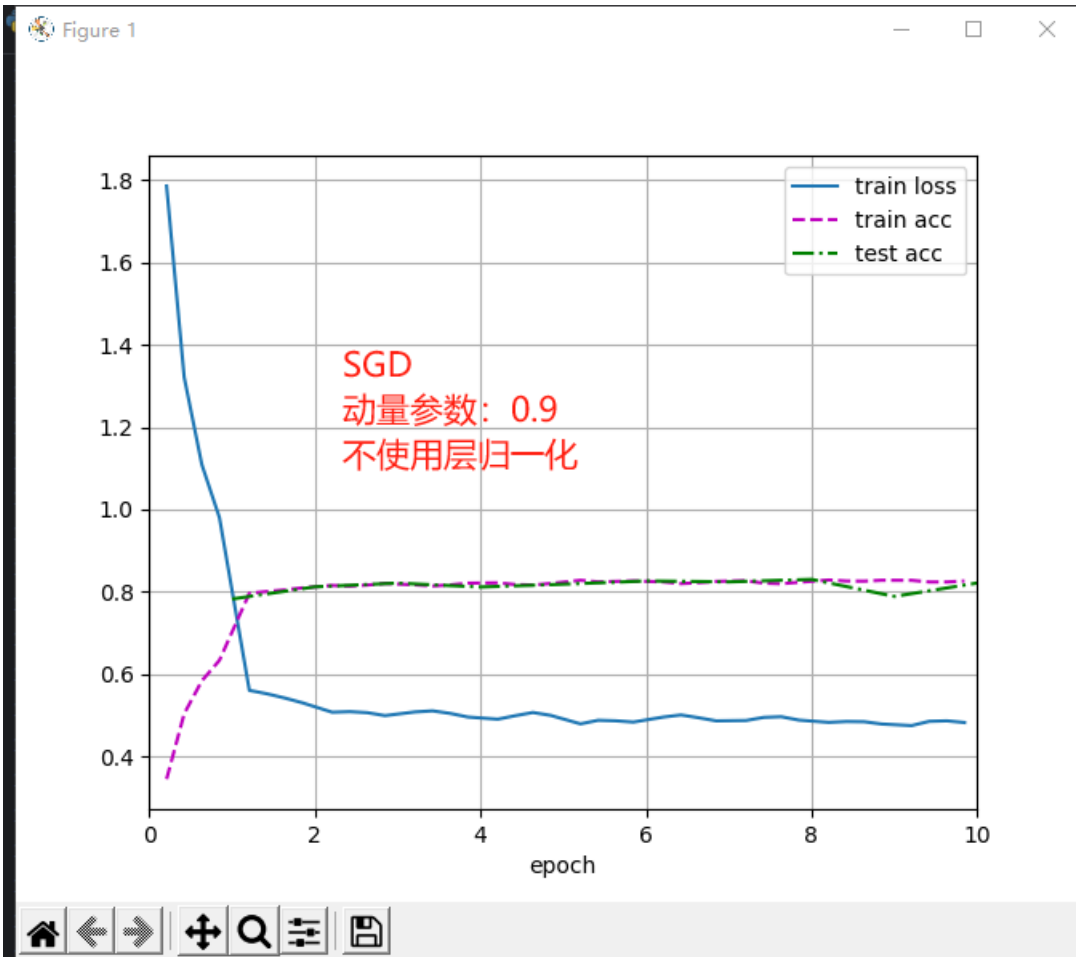
- 要求 6：在计算能力支持的情况下，加深网络，在 SGD+动量法 和 Adam 中使用层归一化（arxiv1904.00962），观察是否有效

增加一层隐藏层，使用 SGD 并且动量参数为 0.9：

```
num_inputs = 784
num_outputs = 10
num_hiddens1 = 128
num_hiddens2 = 56
num_hiddens3 = 32

net =
nn.Sequential(nn.Flatten(),nn.Linear(num_inputs,num_hiddens1),nn.ReLU
(),nn.Dropout(0.0),

nn.Linear(num_hiddens1,num_hiddens2),nn.ReLU(),nn.Dropout(0.0),
```

```
nn.Linear(num_hiddens2,num_hiddens3),nn.ReLU(),
                    nn.Linear(num_hiddens3,num_outputs))
```

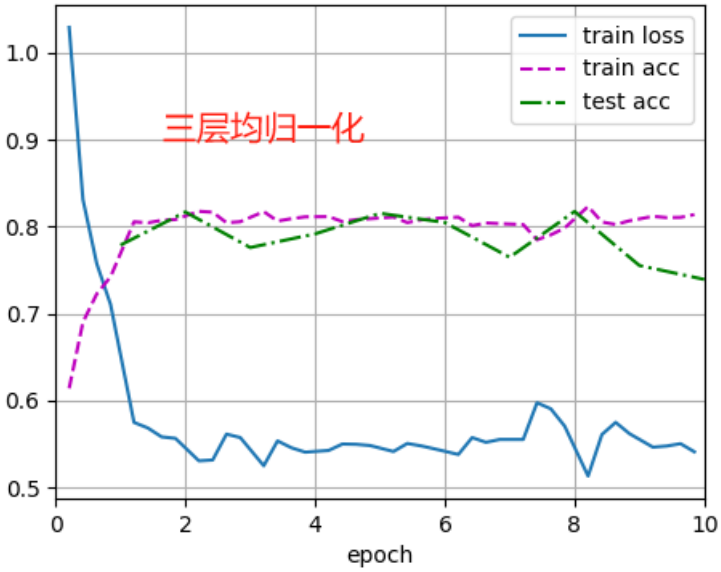
运行结果为：



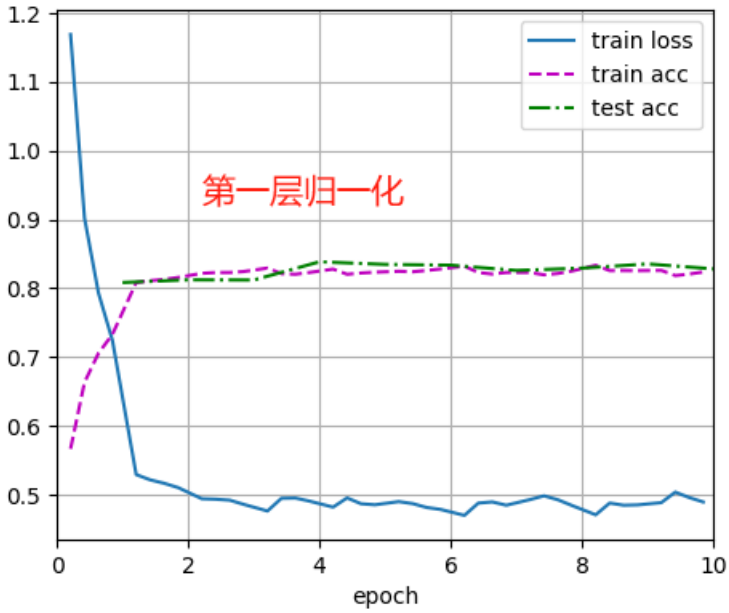
然后在三个隐藏层的每一层之后都加入层归一化：

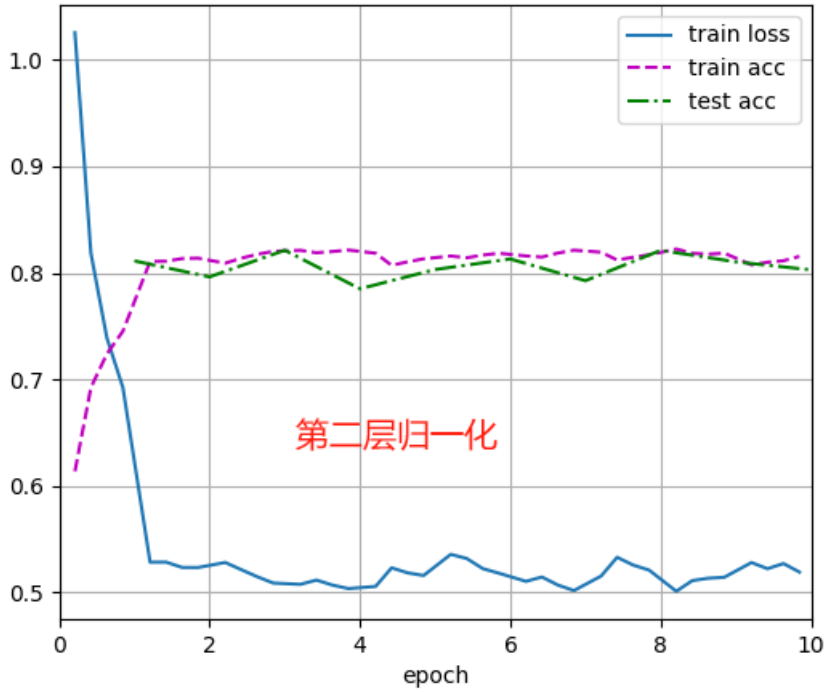
```
net =
nn.Sequential(nn.Flatten(),nn.Linear(num_inputs,num_hiddens1),nn.Laye
rNorm(num_hiddens1),nn.ReLU(),nn.Dropout(0.0),

nn.Linear(num_hiddens1,num_hiddens2),nn.LayerNorm(num_hiddens2),nn.Re
LU(),nn.Dropout(0.0),

nn.Linear(num_hiddens2,num_hiddens3),nn.LayerNorm(num_hiddens3),nn.Re
LU(),
                    nn.Linear(num_hiddens3,num_outputs))
```

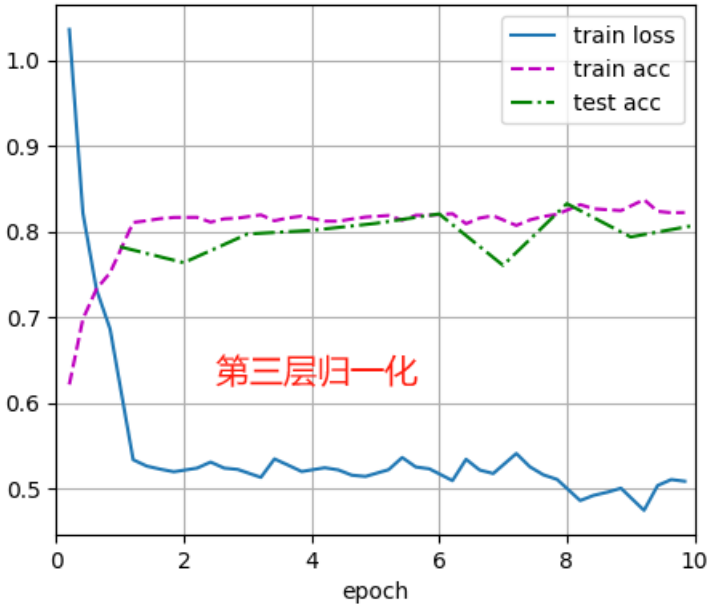
运行结果：



可以看出，发生了较为明显的震荡以及过拟合现象。现在考虑只对某一层进行归一化：







可以看到，只有对第一层进行归一化，能在保证收敛速度和训练准确度的同时，有效地缓解过拟合的发生。并且测试集准确率甚至高于训练集准确率，可见此时模型的泛化性较好。然而对第二、第三层进行层归一化后，训练效果明显下降，并且没能完成抑制过拟合的功能。推测是因为第一层的神经元数目较多，因此归一化的效果更好；而第二第三层的神经元数目少并且参数本身数值小、数值差距不大，因此归一化效果有限。