



南開大學
Nankai University

深度学习实验报告

实验名称：多层感知机

姓名：张恒硕

学号：2212266

专业：智能科学与技术

目录

一、 实验目的	3
二、 实验原理	3
1. 神经元与激活函数	3
2. 神经网络层级结构	4
3. 损失函数	5
三、 实验步骤	5
1. 数据集导入与划分	5
2. 神经网络搭建	6
3. 模型	6
4. 对于代码的说明	6
四、 基础代码	6
五、 调试训练与分析	8
1. fashion_mnist 数据集结果展示 (multilayer_perceptrons.py)	8
2. mnist 数据集结果展示 (mnist.py)	9
3. 隐藏层数 (3hidden.py)	11
4. 隐藏层神经元数量 (multilayer_perceptrons.py)	12
5. 激活函数 (multilayer_perceptrons.py)	13
6. 损失函数 (loss.py)	14
7. dropout (dropout.py)	22
8. batch normalization (batch_normalization.py)	24
六、 附加题	26
1. 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	26
2. 过拟合现象	26
3. 在损失函数中增加 l_2 正则化	27
4. dropout	27
5. batch normalization	27
6. 亦或问题 (XOR.py)	27

一、实验目的

实现一个多层感知机，至少两个隐藏层，配置为全连接层+ ReLU + softmax。利用模型对 mnist(或进阶为 fashion_mnist)数据集进行图像分类，并对训练情况进行分析，尝试各种改进手段并分析。

二、实验原理

多层感知机（Multilayer Perceptron, MLP）是一种前馈人工神经网络，其模仿生物学中神经元的工作状态，将数学过程转换为一层一层的计算，由多层神经元组成，包括至少一个隐藏层。作为监督学习的基础模型，常用于分类和回归问题。

1. 神经元与激活函数

作为神经网络最基本的单元，执行最基础的计算：

$$a = h(wx + b)$$

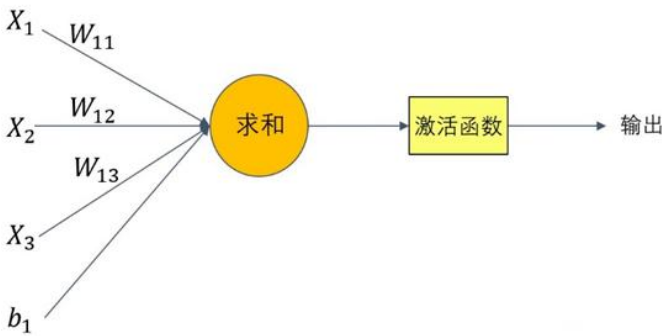
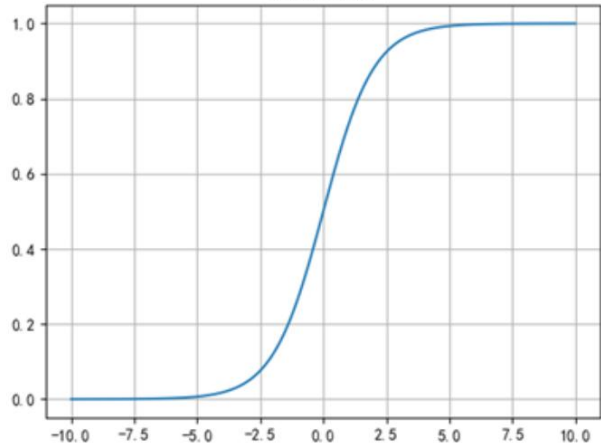
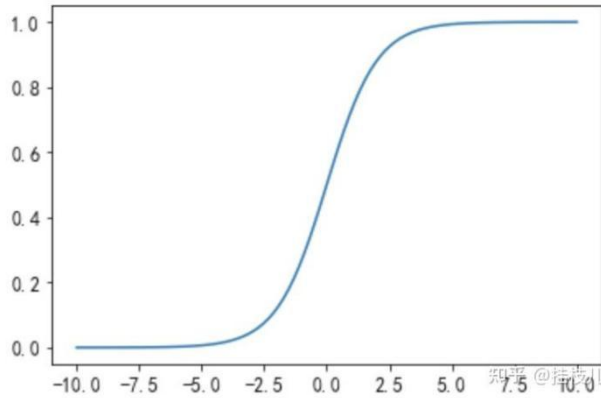
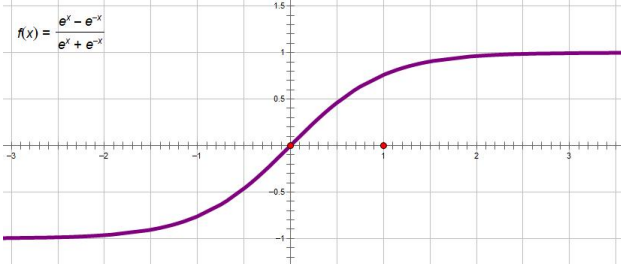


图 2.1 神经元示例

其中，b 为偏置，用于控制神经元被激活的容易程度，而 w 表示控制各信号重要性的权重。h() 为激活函数，是非线性函数，以下给出了几种常用的激活函数的公式和图像。

表 1 常用激活函数

激活函数	函数	导函数	函数图像
ReLU	$y = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases}$	$y' = \begin{cases} 1, & z > 0 \\ 0, & z \leq 0 \end{cases}$	<div><p>ReLU函数</p></div>

sigmoid	$y = \frac{1}{1 + e^{-z}}$	$y' = y(1 - y)'$	
softmax	$y(x_i) = \frac{e^{x_i}}{\sum_{i=1}^n e^{x_i}}$		
tanh	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$		

2. 神经网络层级结构

- (1) 输入层 (Input Layer)：接收输入数据特征，节点数等于特征向量维度。
 - (2) 隐藏层 (Hidden Layer)：包含一定层数的神经元，使网络能够学习复杂的模式。
 - (3) 输出层 (Output Layer)：生成最终预测结果。对于分类问题，二分类常用 sigmoid 激活函数，多分类常用 softmax 激活函数。
- 一个简单的多层感知机如下图所示。

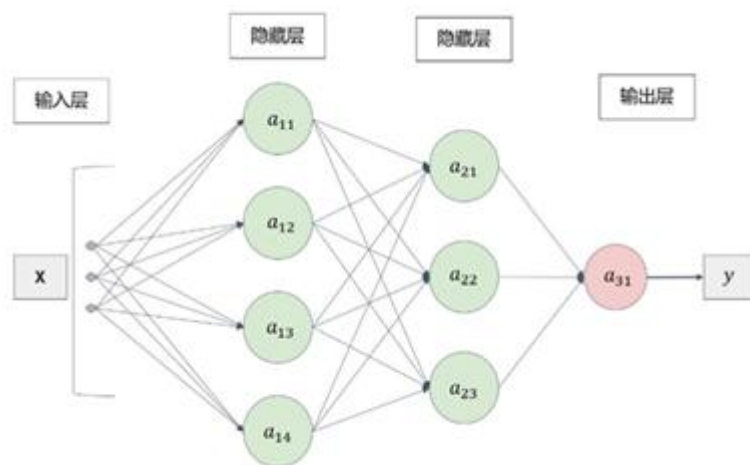


图 2.2 多层感知机示例

3. 损失函数

损失函数用于衡量模型预测结果与实际结果之间的差异，在训练过程中追求最小化，其设计直接影响模型的性能。以下对几种常用损失函数进行介绍，并就交叉熵损失函数的 l_2 正则化进行说明。

表 2 常用损失函数

损失函数	计算公式	特点	适用问题
均方误差 (MSE)	$\frac{1}{2}\Delta y^2$	导数简单，易于优化，但对异常值敏感。	回归
平均绝对误差 (MAE)	$ \Delta y $	对异常值不敏感，但导数不处处连续。	回归
huber	$\begin{cases} \frac{1}{2}\Delta y^2, & \text{for } \Delta y \leq \delta \\ \delta(\Delta y - \frac{1}{2}\delta), & \text{otherwise} \end{cases}$	对小误差使用 MSE，对大误差使用 MAE，具有较好的鲁棒性。	回归
交叉熵	$-\sum_i y_i \log(\hat{y}_i)$	能很好衡量概率分布之间的差异。	多分类
焦点	$-\alpha_t(1-p)^{\gamma} \log(p)$	降低容易分类样本的权重，关注难以分类的样本。	多分类

$\Delta y = y - \hat{y}$ ，其中， Δy 是残差， y 是实际标签， \hat{y} 是预测标签。
 p 是对真实类别的预测概率。

- l_2 正则化：也称为权重衰减 (Ridge Regression)，是一种常用正则化技术，在损失函数中加入正则化项来限制模型参数的大小，从而简化模型并提高泛化能力，防止过拟合。

正则化项为 $\lambda ||w||_2^2$ ，其中 λ 是正则化系数， w 是参数向量。

三、实验步骤

1. 数据集导入与划分

从包中导入 fashion_mnist 数据集，其是衣服的分类数据集（见六、1.）。

或者下载 mnist 数据集，其是数字的分类数据集（见六、2.）。在导入数据集后，划分训练集、验证集、测试集，并设定每次训练的规模。

2. 神经网络搭建

- (1) 激活函数：relu 激活函数手动实现，其他激活函数调库（见五、5.）。
- (2) 隐藏层和神经元个数：设定隐藏层个数（见五、3.）和神经元个数（见五、4.）。
- (3) 构建网络：手动实现的网络运算。可以引入 dropout（见五、7.）和 batch normalization（见五、8.）。

3. 模型

- (1) 损失函数：调用库中交叉熵损失函数，其自带 softmax 激活函数归一。手动增加 l_2 正则化（见五、6.）。其它损失函数手动实现（见五、6.）。
- (2) 训练：设定训练轮次，展开训练，并给出每轮的训练集正确率、验证集正确率和损失。
- (3) 预测：取测试集预测，检验模型。

4. 对于代码的说明

多数代码调用了书本提供的代码库，但在改进过程中，为了适配，也手动实现了全部代码。训练部分见 loss.py，测试部分见 mnist.py。

四、基础代码

由于代码原理比较简单，这里不专门解析代码，请见代码注释。

全连接层+两层 relu 隐藏层+输出层+交叉熵函数

```
import torch

import torch.nn as nn

from d2l import torch as d2l

import time


# 划分数据集

batch_size = 256

train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)


# relu 激活函数

def relu(X):

    a = torch.zeros_like(X)
```

```

    return torch.max(X, a)

# 神经网络

num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256

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, requires_grad=True))

params = [W1, b1, W2, b2, W3, b3]

def net(X):
    X = X.reshape((-1, num_inputs))

    H1 = relu(X @ W1 + b1)

    H2 = relu(H1 @ W2 + b2)

    out = H2 @ W3 + b3

    return out

# 交叉熵损失函数

```

```

loss_CrossEntropyLoss = nn.CrossEntropyLoss(reduction='none')

# 主函数

start_time = time.time()

updater = torch.optim.SGD(params, lr=0.1)

num_epochs= 10

d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, updater)

d2l.predict_ch3(net, test_iter)

d2l.plt.show()

end_time = time.time()

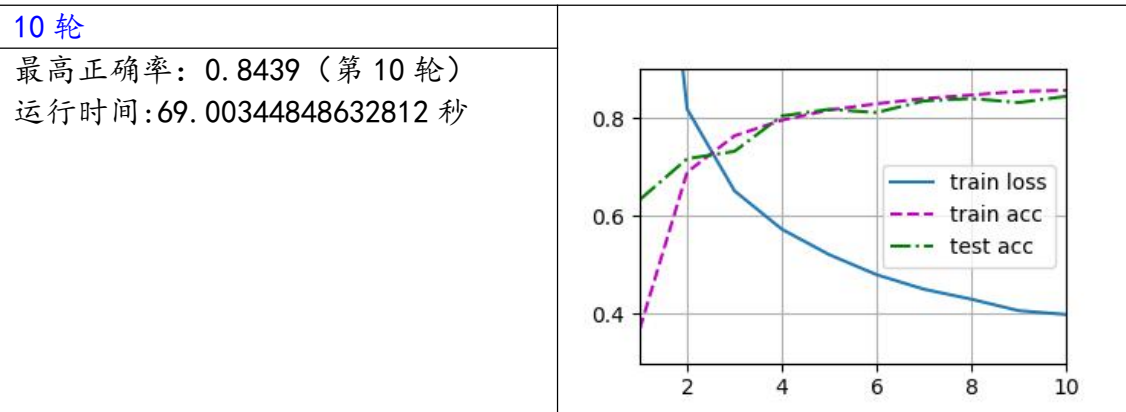
print(f"代码运行时间:{end_time - start_time}秒")

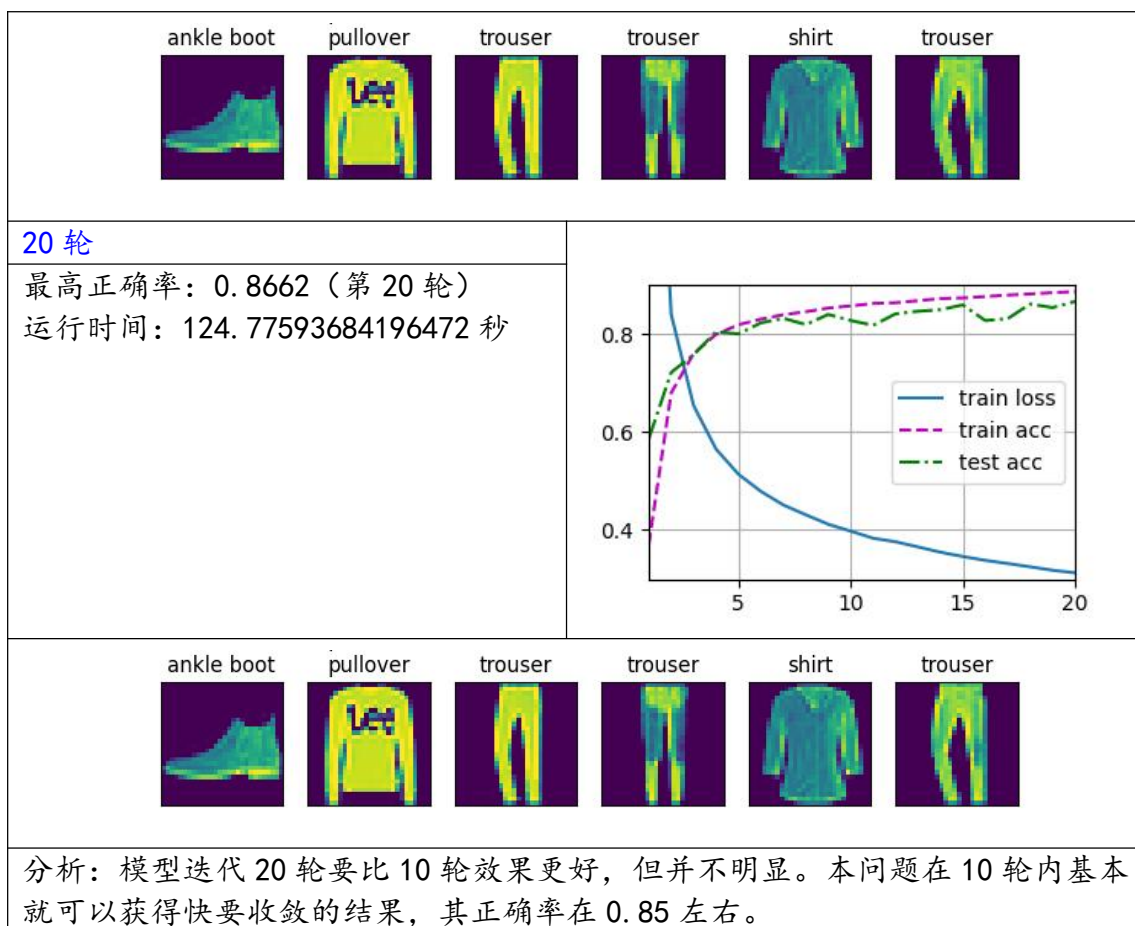
```

五、调试训练与分析

1. fashion_mnist 数据集结果展示 (multilayer_perceptrons.py)

包中能直接调用 fashion_mnist 数据集，其比 mnist 数据集要更复杂一些，更能胜任调试任务，以下以其为数据集对代码展开调试。在针对各部分参数、方法进行改动时，其他参数、方法都会与上述代码中保持相同。本部分关注训练轮次对模型结果的影响。





2. mnist 数据集结果展示 (mnist.py)

由于实验要求使用 mnist 数据集, 这里补充展示, 配置同 1.。mnist 数据集需另外下载, 在本地没有时, 让 `download=True` 即可完成下载, 以下是相关变更的代码。

```
# 加载、划分数据集

train_data = datasets.MNIST(root="../data", train=True,
transform=transforms.ToTensor(), download=False)

test_data = datasets.MNIST(root="../data", train=False,
transform=transforms.ToTensor(), download=False)

batch_size = 256

train_iter = DataLoader(train_data, batch_size=batch_size, shuffle=True)

test_iter = DataLoader(test_data, batch_size=batch_size, shuffle=False)
```

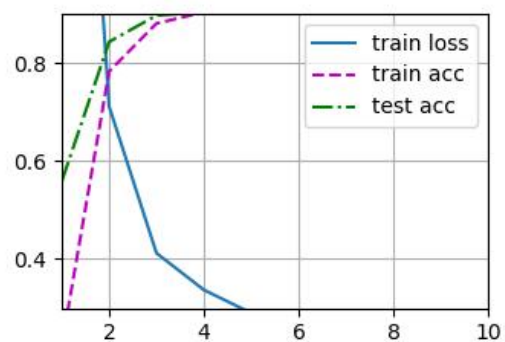
由于数据集发生变化, 相应的预测函数代码也发生了变化, 具体如下:

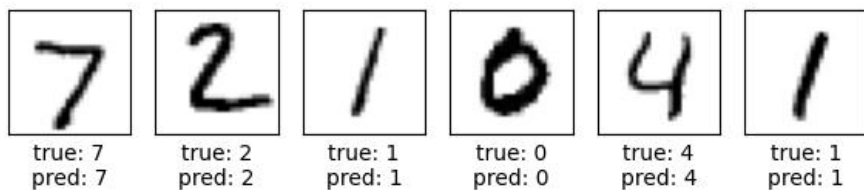
预测函数

```
def predict(net, test_iter, n=6):  
  
    for X, y in test_iter:  
  
        break  
  
    preds = net(X).argmax(dim=1)  
  
    trues = [str(y_i.item()) for y_i in y[:n]]  
  
    preds = [str(pred_i.item()) for pred_i in preds[:n]]  
  
    titles = ['true: ' + true + '\npred: ' + pred for true, pred in zip(trues, preds)]  
  
    plt.figure(figsize=(n * 1.2, 2.4))  
  
    for i in range(n):  
  
        plt.subplot(1, n, i + 1)  
  
        plt.xticks([])  
  
        plt.yticks([])  
  
        plt.grid(False)  
  
        plt.imshow(X[i].reshape(28, 28), cmap=plt.cm.binary)  
  
        plt.xlabel(titles[i])  
  
    plt.show()
```

mnist

最高正确率: 0.9562 (第 10 轮)
运行时间: 50.64063763618469 秒





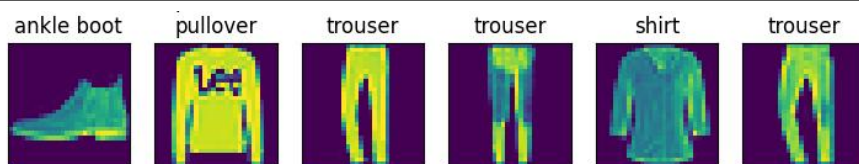
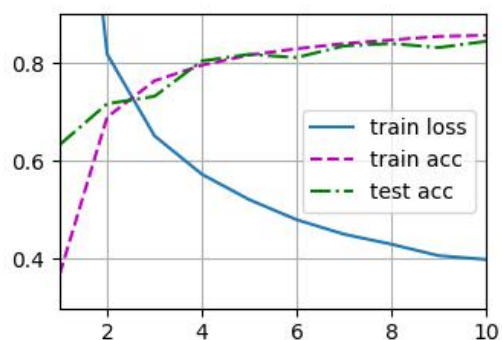
分析：与 1. 对比，可以发现正确率明显更高，运行时间也更短，这验证了 fashion_mnist 数据集比 mnist 更复杂的结论。

3. 隐藏层数 (3hidden.py)

增加一个隐藏层会使参数增加，改变的代码较为简单，这里不再展示。

2 层

最高正确率：0.8439 (第 10 轮)
运行时间:69.00344848632812 秒



参数量：

W1: 784×256 b1: 256

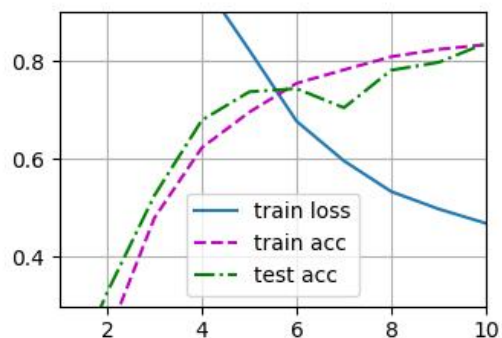
W2: 256×256 b2: 256

W3: 256×10 b3: 10

总: 268234

3 层

最高正确率：0.8362 (第 10 轮)
运行时间: 67.1825487613678 秒

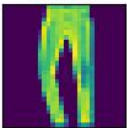
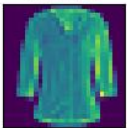
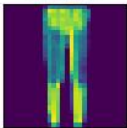
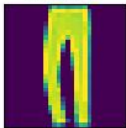
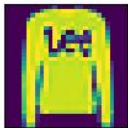
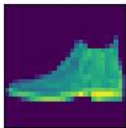
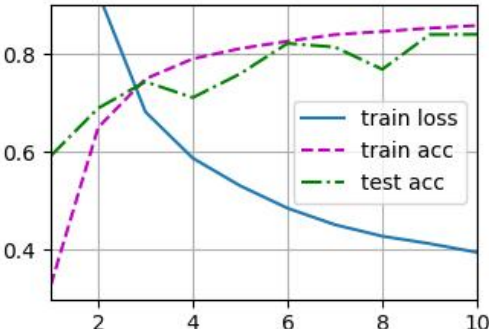
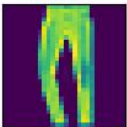
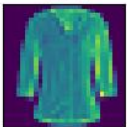
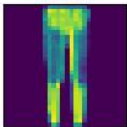
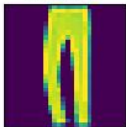
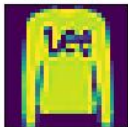
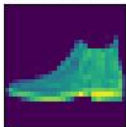


<div> <div>ankle boot</div> <div>pullover</div> <div>trouser</div> <div>trouser</div> <div>shirt</div> <div>trouser</div> </div>					
参数量： $W1: 784 \times 256$ $b1: 256$ $W2: 256 \times 256$ $b2: 256$ $W3: 256 \times 256$ $b3: 256$ $W4: 256 \times 10$ $b4: 10$ 总：334026					
分析：对比两个折线图，可以发现，增加一个隐藏层不但没有增加正确率，还导致迭代过程中每个轮次都比未增加时差。参数量增加了约 1/4，却没有使结果变好，但是代码运行时间没有增加，说明在该数据集的规模下，多进行一次矩阵计算并没有多大时间成本损失。					

4. 隐藏层神经元数量 (multilayer_perceptrons.py)

改变隐藏层的神经元数也可能带来结果上的变化，改变的代码较为简单，这里不再展示。

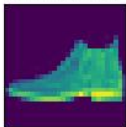

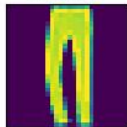
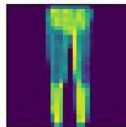
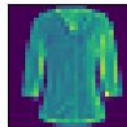
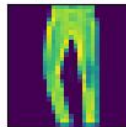
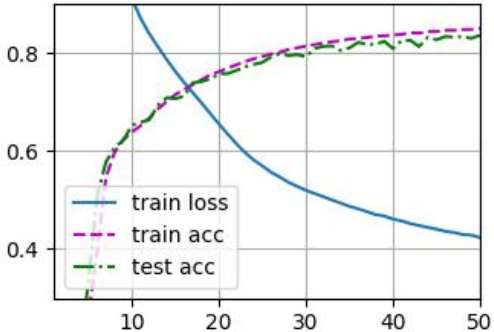
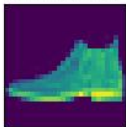

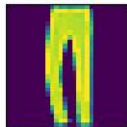
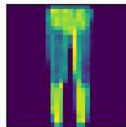
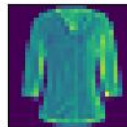
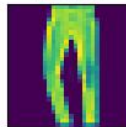
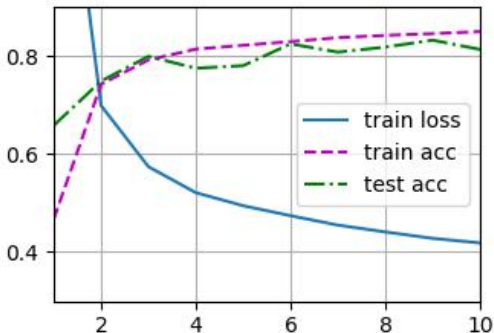
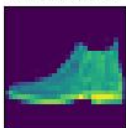

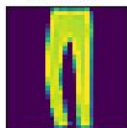
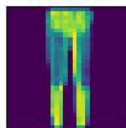
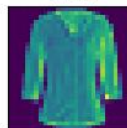
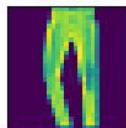
<div>256*256</div> <div>最高正确率：0.8439（第 10 轮） 运行时间：69.00344848632812 秒</div>			
<div>ankle boot</div> <div>pullover</div> <div>trouser</div> <div>trouser</div> <div>shirt</div> <div>trouser</div>			
参数量： $W1: 784 \times 256$ $b1: 256$ $W2: 256 \times 256$ $b2: 256$ $W3: 256 \times 10$ $b3: 10$ 总：268234			
<div>512*256</div> <div>最高正确率：0.8459（第 9 轮） 运行时间：60.63050699234009 秒</div>			

<div>ankle boot pullover trouser trouser pullover trouser</div> <div></div>					
<div>参数量:</div> <div>W1: 784×512 b1: 512</div> <div>W2: 512×256 b2: 256</div> <div>W3: 256×10 b3: 10</div> <div>总: 535,738</div>					
<div>256*128</div> <div>最高正确率: 0.8398 (第 10 轮)</div> <div>运行时间: 64.39833474159241 秒</div>			<div></div>		
<div>ankle boot pullover trouser trouser shirt trouser</div> <div></div>					
<div>参数量:</div> <div>W1: 784×256 b1: 256</div> <div>W2: 256×128 b2: 128</div> <div>W3: 128×10 b3: 10</div> <div>总: 235,146</div>					
<div>分析: 对比三个折线图, 可以发现最终正确率上没有明显的差别, 然而, 参数量越小, 损失函数收敛越慢, 测试误差在收敛过程中的波动越大。这一差别主要体现在第 1 组和第 3 组上, 第 1 组和第 2 组并不明显, 这说明参数量增加对模型效果的改善并不是无限的, 在参数量增加到一定程度后, 再增加并不会改善模型。</div>					

5. 激活函数 (multilayer_perceptrons.py)

激活函数的变化也会影响模型的表达能力, 选取合适的激活函数十分重要。以下对几个常见的激活函数展开对比, 由于都是调用库中的函数, 对代码的修改不做展示。由于训练结果对损失值有要求, 对于表现较差的 Sigmoid 激活函数进行了更多轮次的迭代, 这样也可以更加凸出地进行对比。

relu	
最高正确率: 0.8439 (第 10 轮) 运行时间: 69.00344848632812 秒	

<div> <div>ankle boot</div>  </div> <div> <div>pullover</div>  </div> <div> <div>trouser</div>  </div> <div> <div>trouser</div>  </div> <div> <div>shirt</div>  </div> <div> <div>trouser</div>  </div>	
<div>Sigmoid (50 轮)</div> <div> <div>最高正确率: 0.836 (第 50 轮)</div> <div>运行时间: 183.69251465797424 秒</div> </div>	
<div> <div>ankle boot</div>  </div> <div> <div>pullover</div>  </div> <div> <div>trouser</div>  </div> <div> <div>trouser</div>  </div> <div> <div>shirt</div>  </div> <div> <div>trouser</div>  </div>	
<div>tanh</div> <div> <div>最高正确率: 0.832 (第 9 轮)</div> <div>运行时间: 57.47852039337158 秒</div> </div>	
<div> <div>ankle boot</div>  </div> <div> <div>pullover</div>  </div> <div> <div>trouser</div>  </div> <div> <div>trouser</div>  </div> <div> <div>shirt</div>  </div> <div> <div>trouser</div>  </div>	
<div>分析: 对比以上结果可知, relu 激活函数是最适合本问题的激活函数。Sigmoid 激活函数的表现十分差, 这是因为其更适用于二分类而非本问题中的多分类。tanh 激活函数的结果相对较好, 但也不如 relu 激活函数, 收敛速度比较慢。</div>	

6. 损失函数 (loss.py)

损失函数有很多种, 本部分对适用于多分类任务或回归任务的损失函数进行对比, 并补充 l_2 正则化给交叉熵损失函数带来的影响。以下先给出相应的代码, 包含损失函数、训练函数、绘图的手动实现。

● 损失函数:


```
# 均方误差损失函数
```

```
class MSELoss(nn.Module):
```

```
    def __init__(self):
```

```
        super(MSELoss, self).__init__()
```

```
    def forward(self, output, target, params=None):
```

```
        target_one_hot = torch.nn.functional.one_hot(target,
```

```
num_classes=num_outputs).float()
```

```
        mse = torch.square(output - target_one_hot)
```

```
        return mse.mean()
```

```
# 平均绝对误差
```

```
class MAELoss(nn.Module):
```

```
    def __init__(self):
```

```
        super(MAELoss, self).__init__()
```

```
    def forward(self, output, target, params=None):
```

```
        target_one_hot = torch.nn.functional.one_hot(target,
```

```
num_classes=num_outputs).float()
```

```
        mae = torch.abs(output - target_one_hot)
```

```
        return mae.mean()
```

```
# Huber 损失函数
```

```
class HuberLoss(nn.Module):
```

```

def __init__(self, delta=1.0):

    super(HuberLoss, self).__init__()

    self.delta = delta

def forward(self, output, target, params=None):

    target_one_hot = torch.nn.functional.one_hot(target,
num_classes=num_outputs).float()

    mask = (output - target_one_hot).abs() < self.delta

    loss = torch.zeros_like(output)

    loss[mask] = 0.5 * (output[mask] - target_one_hot[mask]) ** 2

    loss[~mask] = self.delta * (output[~mask] - target_one_hot[~mask]).abs() - 0.5
* self.delta ** 2

    return loss.mean()

# 焦点损失函数

class FocalLoss(nn.Module):

    def __init__(self, alpha=0.25, gamma=2.0):

        super(FocalLoss, self).__init__()

        self.alpha = alpha

        self.gamma = gamma

    def forward(self, output, target, params=None):

        target_one_hot = torch.nn.functional.one_hot(target,
num_classes=num_outputs).float()

```



```
softmax_output = torch.softmax(output, dim=1)

probs = (softmax_output * target_one_hot).sum(dim=1)

loss = -self.alpha * torch.pow(1.0 - probs, self.gamma) * torch.log(probs)

return loss.mean()
```

带 l2 正则化的交叉熵损失函数

```
class CrossEntropy_L2(nn.Module):

    def __init__(self, weight_decay=0.01):

        super(CrossEntropy_L2, self).__init__()

        self.weight_decay = weight_decay

        self.cross_entropy_loss = nn.CrossEntropyLoss()

    def forward(self, output, target, params):

        ce_loss = self.cross_entropy_loss(output, target)

        # L2 正则化项

        l2_reg = torch.tensor(0., device=params[0].device)

        for param in params:

            l2_reg += torch.norm(param, p=2) ** 2

        total_loss = ce_loss + self.weight_decay * l2_reg

        return total_loss
```

loss = MSELoss()

loss = MAELoss()

```
# loss = HuberLoss()

# loss = FocalLoss()

loss = CrossEntropy_L2()
```

● 训练函数:

```
# 正确率评估函数

def accuracy(data_iter, net):

    metric = Accumulator(2)

    with torch.no_grad():

        for X, y in data_iter:

            y_hat = net(X)

            cmp = (y_hat.argmax(axis=1) == y).type(y.dtype)

            metric.add(float(cmp.sum()), cmp.numel())

    return metric[0] / metric[1]

class Accumulator:

    def __init__(self, n):

        self.data = [0.0] * n

    def add(self, *args):

        self.data = [a + float(b) for a, b in zip(self.data, args)]

    def reset(self):

        self.data = [0.0] * len(self.data)

    def __getitem__(self, idx):

        return self.data[idx]
```

训练函数

```
def train_and_record(train_iter, test_iter, net, loss, trainer, num_epochs, params):
```

```
    train_l_history = []
```

```
    train_acc_history = []
```

```
    test_acc_history = []
```

```
    for epoch in range(num_epochs):
```

```
        train_l_sum, train_acc_sum, n = 0.0, 0.0, 0
```

```
        for X, y in train_iter:
```

```
            y_hat = net(X)
```

```
            l = loss(y_hat, y, params)
```

```
            trainer.zero_grad()
```

```
            l.backward()
```

```
            trainer.step()
```

```
            train_l_sum += l.cpu().item()
```

```
            n += y.shape[0]
```

```
        train_l_history.append(train_l_sum / n)
```

```
        train_acc_sum = accuracy(train_iter, net)
```

```
        train_acc_history.append(train_acc_sum)
```

```
        test_acc = accuracy(test_iter, net)
```

```
        test_acc_history.append(test_acc)
```

```
        print(f'epoch {epoch + 1}, test acc {test_acc:.3f}')
```

```
    return train_l_history, train_acc_history, test_acc_history
```

● 绘图：

```
# 绘制曲线

plt.figure(figsize=(10, 5))

epochs = list(range(1, num_epochs + 1))

plt.subplot(1, 2, 1)

plt.plot(epochs, train_l_history, label='Training Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend()

plt.subplot(1, 2, 2)

plt.plot(epochs, train_acc_history, label='Training Accuracy')

plt.plot(epochs, test_acc_history, label='Test Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

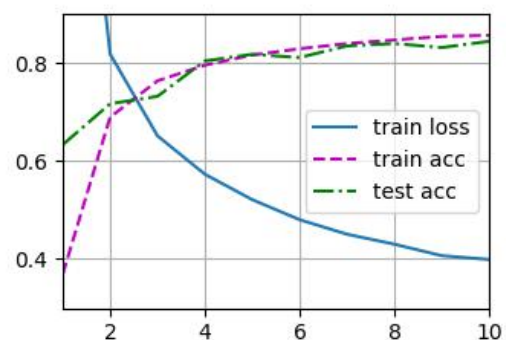
plt.tight_layout()

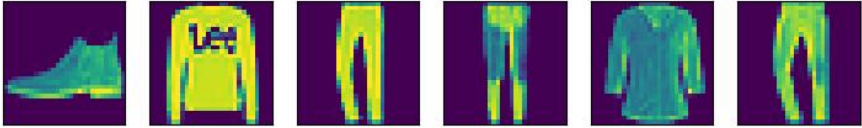
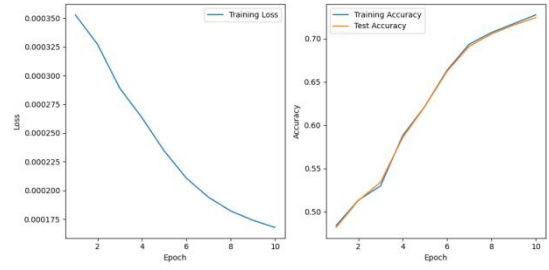
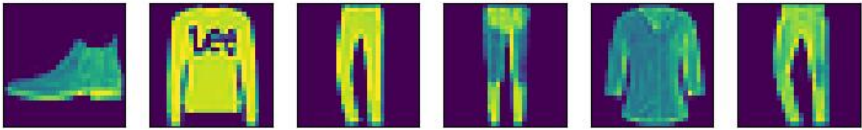
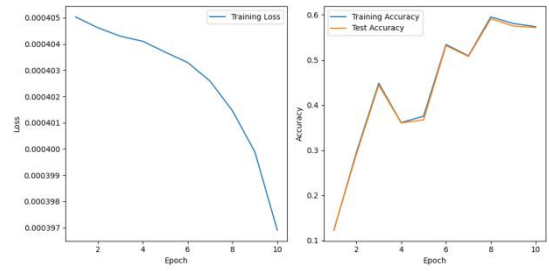
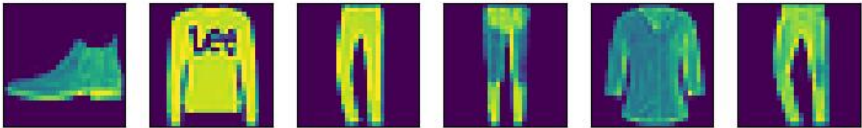
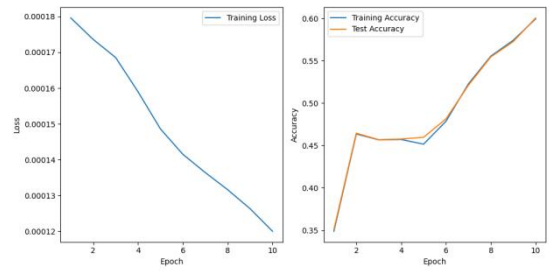
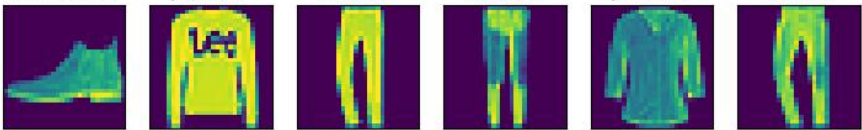
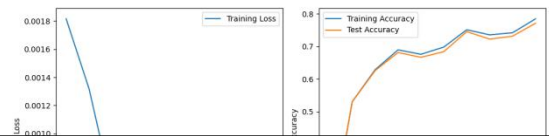
plt.show(block=False)

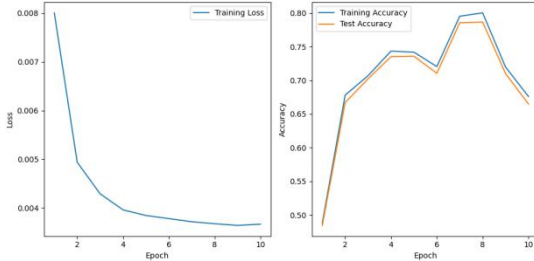
plt.pause(0.001)
```

交叉熵损失函数

最高正确率：0.8439（第10轮）
运行时间：69.00344848632812 秒



<div>ankle boot pullover trouser trouser shirt trouser</div> 	
<p>均方误差损失函数 (MSE)</p> <p>最高正确率: 0.724 (第 10 轮)</p> <p>运行时间: 130.6605625152588 秒</p>	
<div>ankle boot pullover trouser trouser t-shirt trouser</div> 	
<p>平均绝对误差损失函数 (MAE)</p> <p>最高正确率: 0.572 (第 10 轮)</p> <p>运行时间: 121.93230104446411 秒</p>	
<div>ankle boot pullover trouser trouser t-shirt trouser</div> 	
<p>Huber 损失函数</p> <p>最高正确率: 0.601 (第 10 轮)</p> <p>运行时间: 131.12822937965393 秒</p>	
<div>ankle boot pullover trouser trouser pullover trouser</div> 	
<p>焦点损失函数</p> <p>最高正确率: 0.771 (第 10 轮)</p> <p>运行时间: 130.4335880279541 秒</p>	

<div><div>ankle boot</div><div>pullover</div><div>trouser</div><div>trouser</div><div>shirt</div><div>trouser</div></div>					
<p>分析：对比以上各损失函数的结果可知，针对目前多分类任务，交叉熵损失函数的结果最好。作为改进的焦点损失函数也没有交叉熵损失函数好，这可能是因为数据集的分类是均匀的，焦点损失函数的特性并未生效。</p>					
<p>l_2正则化的交叉熵损失函数</p> <p>最高正确率：0.786（第8轮）</p> <p>运行时间：132.86843848228455 秒</p>					
<div><div>ankle boot</div><div>shirt</div><div>trouser</div><div>trouser</div><div>shirt</div><div>trouser</div></div>					
<p>分析：l_2正则化的加入在本实验中并未改进效果，其原因是有多方面的。一是参数λ可能不合适，二是题目数据集比较简单，又足够大，不需要正则化来补充信息。</p>					

7. dropout (dropout.py)

dropout 作为另一种正则化技术，在训练过程中随机失活网络中的一部分神经元（输出 0），来减少模型对训练数据的过度依赖。其可以防止过拟合、提高鲁棒性、间接减小计算量、实现模型平均的效果。以下给出具体实现方法：

```
dropout_rate = 0.5

def net(X, training=True):

    X = X.reshape((-1, num_inputs))

    H1 = relu(X @ W1 + b1)

    if training:

        mask = (torch.rand(H1.shape) > dropout_rate).float()

        H1 = H1 * mask / (1 - dropout_rate)

    H2 = relu(H1 @ W2 + b2)
```

if training:

```
mask = (torch.rand(H2.shape) > dropout_rate).float()
```

```
H2 = H2 * mask / (1 - dropout_rate)
```

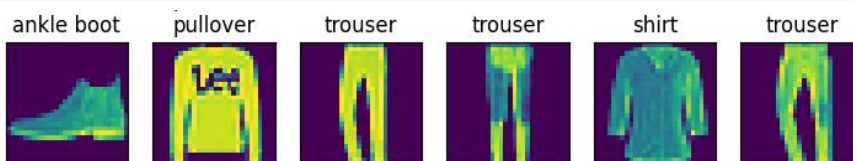
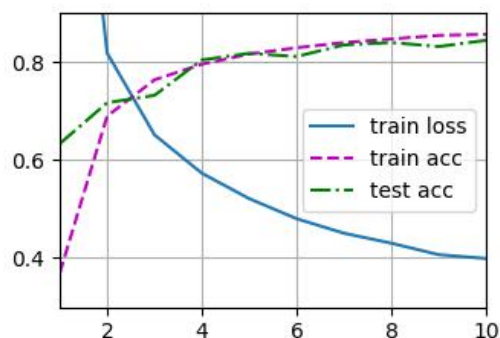
```
out = H2 @ W3 + b3
```

return out

无 dropout

最高正确率: 0.8439 (第 10 轮)

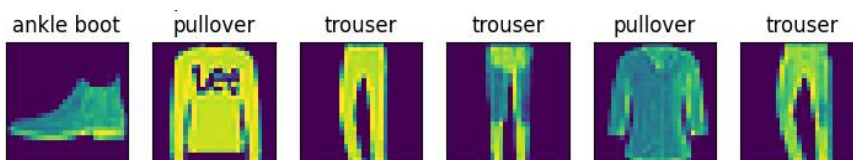
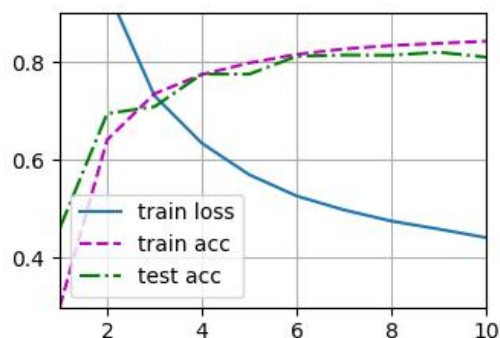
运行时间: 69.00344848632812 秒



有 dropout (10 轮, 失活率 0.5)

最高正确率: 0.8197 (第 9 轮)

运行时间: 77.03412342071533 秒

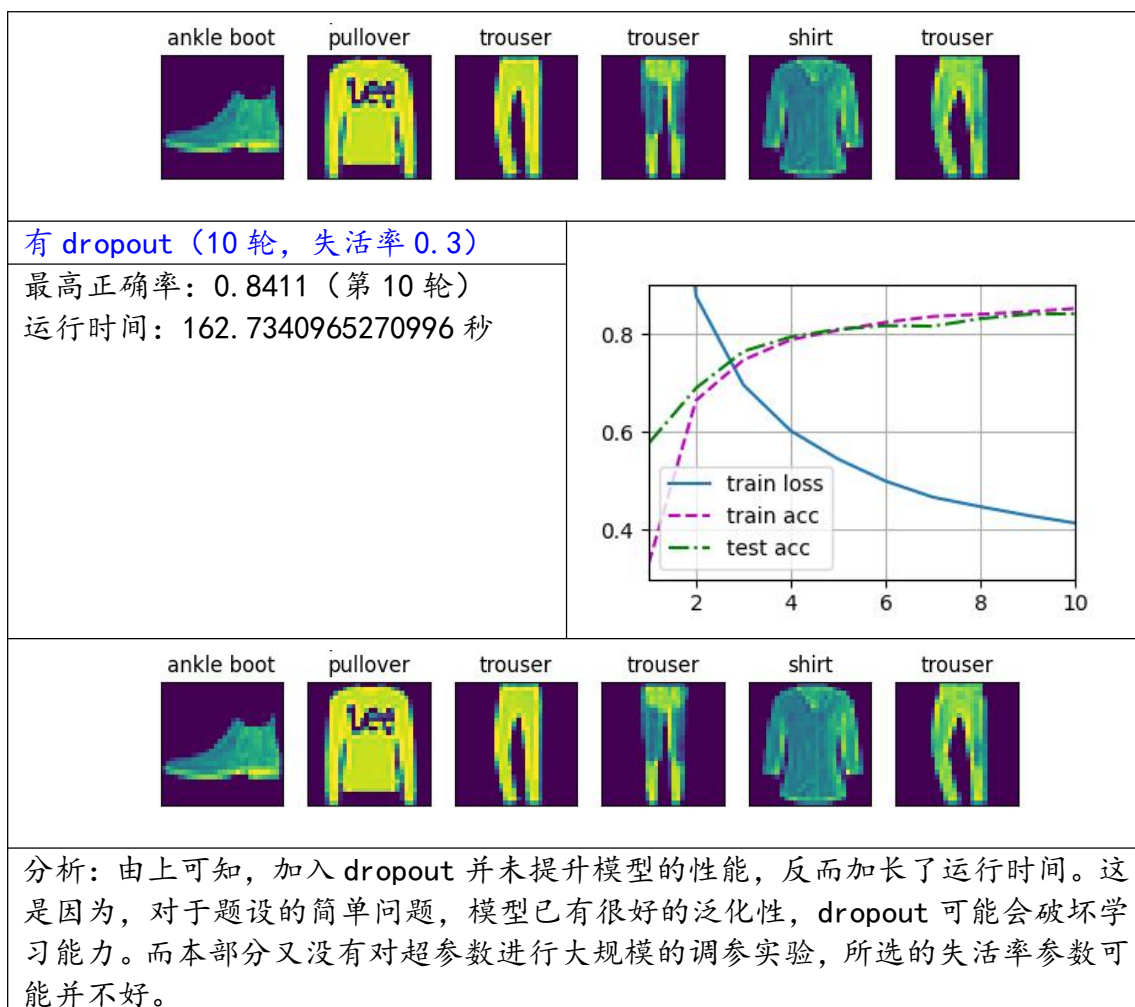


有 dropout (20 轮, 失活率 0.5)

最高正确率: 0.8438 (第 19 轮)

运行时间: 288.8818304538727 秒





8. batch normalization (batch_normalization.py)

Batch Normalization 通过对每个小批量样本进行归一化操作, 使得输入的特征具有零均值和单位方差, 这有助于减少梯度消失和梯度爆炸问题, 从而加速神经网络的收敛。其可以提高模型稳定性、泛化能力, 并减少对其他正则化技术的依赖。以下给出具体实现方法:

```
def batch_norm(X, gamma, beta, running_mean, running_var, eps=1e-5,
momentum=0.9):

    if X.ndim > 2:

        X = X.reshape(X.shape[0], -1)

        mean = X.mean(0)

        var = ((X - mean) ** 2).mean(0)

        running_mean = momentum * running_mean + (1 - momentum) * mean
```



```

    running_var = momentum * running_var + (1 - momentum) * var

    X_hat = (X - mean) / torch.sqrt(var + eps)

    Y = gamma * X_hat + beta

    return Y, running_mean, running_var

gamma1 = nn.Parameter(torch.ones(num_hiddens1))
beta1 = nn.Parameter(torch.zeros(num_hiddens1))
gamma2 = nn.Parameter(torch.ones(num_hiddens2))
beta2 = nn.Parameter(torch.zeros(num_hiddens2))
params = [W1, b1, gamma1, beta1, W2, b2, gamma2, beta2, W3, b3]

def net(X):

    X = X.reshape((-1, num_inputs))

    H1, running_mean1, running_var1 = batch_norm(X @ W1 + b1, gamma1, beta1,
torch.zeros(num_hiddens1), torch.ones(num_hiddens1))

    H1 = relu(H1)

    H2, running_mean2, running_var2 = batch_norm(H1 @ W2 + b2, gamma2, beta2,
torch.zeros(num_hiddens2), torch.ones(num_hiddens2))

    H2 = relu(H2)

    out = H2 @ W3 + b3

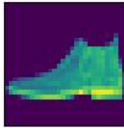
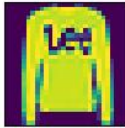
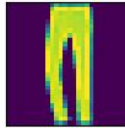
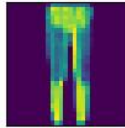
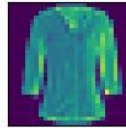
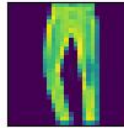
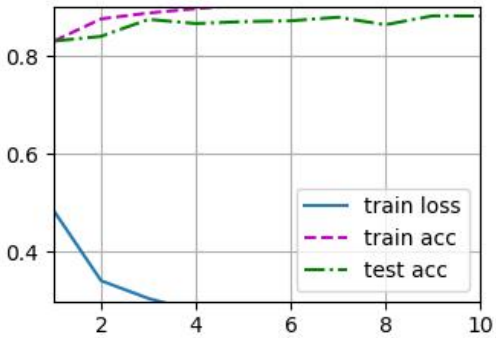
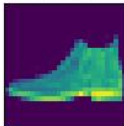
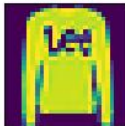
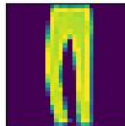
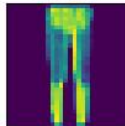
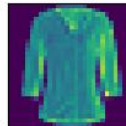
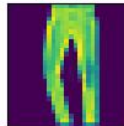
    return out

```

无 batch normalization

最高正确率: 0.8439 (第 10 轮)



运行时间:69.00344848632812 秒		
<div><div>ankle boot</div><div>pullover</div><div>trouser</div><div>trouser</div><div>shirt</div><div>trouser</div></div>		
有 batch normalization		
最高正确率: 0.8817 (第 9 轮) 运行时间: 76.78819370269775 秒		
<div><div>ankle boot</div><div>pullover</div><div>trouser</div><div>trouser</div><div>shirt</div><div>trouser</div></div>		
分析: 对比两个折线图, 可以明显发现, batch normalization 起到了很明显的作用。训练准确率和检验准确率的初值都很高, 快速收敛后最终准确率也高出不少, 和其他改进方法相比有很明显的优势。而且训练误差的初值也较小, 变小得也很快。		

六、附加题

1. 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见五、3. 和见五、4. 。

在一定范围内, 增加复杂度, 模型的训练误差和测试误差都会更快变小, 收敛值也会变小, 使模型的效果更好。然而超出一定范围, 再增加便不会有明显的改善。这个阈值主要与待分类数据集的复杂程度有关。

2. 过拟合现象

以上展示的各种结果中, 过拟合现象可能不太明显, 以下对基础模型迭代 100 次来观察。

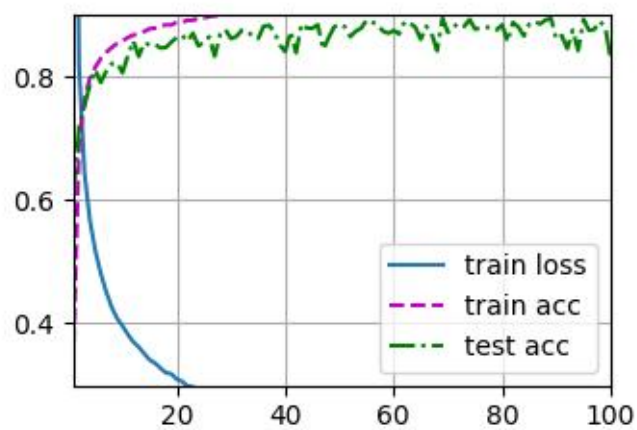


图 7-2 100 次

可以发现，当训练准确率在较高处收敛后，验证准确率一直在其下跳动，这是十分明显的过拟合现象。

3. 在损失函数中增加 l_2 正则化

见五、6.。

4. dropout

见五、7.。

5. batch normalization

见五、8.。

6. 亦或问题 (XOR.py)

要求：

$$h = \text{relu}(wx + b)$$

$$y = w^T h + b$$

- 隐藏层使用 ReLU 激活函数；
- 输出层不使用激活函数；
- 使用均方误差损失函数；
- 使用随机梯度法训练网络。

代码如下：

```
import torch

import torch.nn as nn

from torch import optim

# 神经网络

class XORNet(nn.Module):

    def __init__(self):
```

```

        super(XORNet, self).__init__()

        self.hidden = nn.Linear(2, 5)

        self.output = nn.Linear(5, 1)

    def forward(self, x):

        x = torch.relu(self.hidden(x))

        x = self.output(x)

        return x

# 训练数据

X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)

y = torch.tensor([[0], [1], [1], [0]], dtype=torch.float32)

# 训练配置

model = XORNet()

criterion = nn.MSELoss() # 均方误差损失函数

optimizer = optim.SGD(model.parameters(), lr=0.1) # 随机梯度下降优化器

# 训练

num_epochs = 10000

for epoch in range(num_epochs):

    model.train()

    optimizer.zero_grad()

    outputs = model(X)

```

```
loss = criterion(outputs, y)

loss.backward()

optimizer.step()

if (epoch + 1) % 1000 == 0:

    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')


# 交互

def get_user_input():

    x1 = int(input("请输入第一个数字（0 或 1）："))

    x2 = int(input("请输入第二个数字（0 或 1）："))

    input_tensor = torch.tensor([[x1, x2]], dtype=torch.float32)

    return input_tensor


# 预测

def predict_xor(model, input_tensor):

    model.eval()

    with torch.no_grad():

        prediction = model(input_tensor)

        rounded_prediction = torch.round(prediction).int().item()

    return rounded_prediction


# 主函数
```

```
input_tensor = get_user_input()

prediction = predict_xor(model, input_tensor)

print(f'预测结果: {prediction}')
```

训练过程中，损失为 0（仅展示整千次的损失），其最终结果如下：

```
请输入第一个数字（0或1）：0
请输入第二个数字（0或1）：0
预测结果：0
```

```
请输入第一个数字（0或1）：1
请输入第二个数字（0或1）：1
预测结果：0
```

```
请输入第一个数字（0或1）：0
请输入第二个数字（0或1）：1
预测结果：1
```

```
请输入第一个数字（0或1）：1
请输入第二个数字（0或1）：0
预测结果：1
```