



南開大學
Nankai University

深度学习实验报告

实验名称：优化器

姓名：张恒硕

学号：2212266

专业：智能科学与技术

目录

一、 实验目的	4
二、 实验原理	4
1. 优化器	4
(1) SGD (随机梯度下降, Stochastic Gradient Descent)	4
(2) SGD+动量法 (Momentum)	4
(3) Adam (Adaptive Moment Estimation)	4
2. 初始化权重	4
高斯分布随机初始化	5
Xavier 初始化	5
Kaiming 均匀分布初始化 (He 初始化)	5
3. 批量大小 (batch_size)	5
4. 学习率 (Learning Rate, LR) 策略	5
固定学习率 (Fixed Learning Rate)	5
指数衰减 (Exponential Decay)	5
分段学习率 (Step Decay, Piecewise Constant Decay)	5
多项式衰减 (Polynomial Decay)	5
线性衰减 (Linear Decay)	5
余弦衰减 (Cosine Annealing)	6
预热 (Warm-up)	6
三、 实验步骤	6
四、 基础代码	6
五、 调试训练与分析	13
1. SGD	13
(1) 结果展示	13
(2) 初始化权重	14
(3) 批量大小	14
(4) 学习率策略	16
2. SGD+动量法	18
(1) 结果展示	18
(2) 初始化权重	18
(3) 学习率策略	19
(4) 权重衰减	20
(5) Nesterov	21
3. Adam	21
(1) 结果展示	22
(2) 初始化权重	22
(3) 学习率策略	23
(4) AMSGrad	24
(5) AdamW	24
六、 附加题	25
1. 从并行计算角度解释不同批量大小对算法训练速度的影响	25
2. SGD+动量法进一步尝试	25
3. Adam 进一步尝试 1	25

4. Adam 进一步尝试 2.....	25
5. 在卷积神经网络上重复实验	25

一、实验目的

基于实验一的多层感知机模型，调试优化器配置，探索各种优化器的性能。使用 Fashion-mnist 数据集（与要求的 mnist 数据集不同）进行实验。

二、实验原理

优化器是神经网络训练过程中调整模型参数以最小化损失函数的关键组件。

- 参数更新：基于损失函数对模型参数的梯度来更新参数。
- 学习率控制：学习率决定了参数更新的步伐，过大会导致模型无法收敛，过小则需要过多迭代次数。
- 正则化：权重衰减等正则化机制可以防止过拟合，提高模型的泛化能力。

1. 优化器

常用的优化器有 SGD 和 Adam，在它们基础上，又演变出更多具有特定功能的优化器，以下进行简单的介绍。

(1) SGD（随机梯度下降，Stochastic Gradient Descent）

在每次迭代中随机选择一小批样本估计梯度，计算效率高，并因为随机性引入噪声能够逃离局部最小值。其公式如下：

$$\theta_{t+1} = \theta_t - \eta \nabla f_i(\theta_t)$$

(2) SGD+动量法（Momentum）

动量法的加入可以在相关方向上加速 SGD 收敛并抑制振荡。动量法可以抑制梯度变化，进而抑制搜索空间中每个新点的步长。以下图 2 说明了二者的区别。

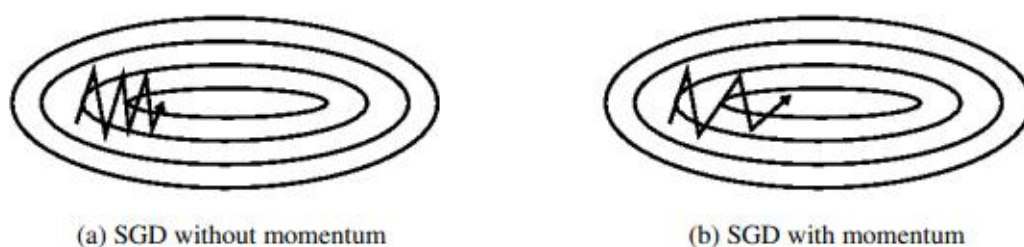


图 2 SGD 有无动量法的差别

(3) Adam (Adaptive Moment Estimation)

作为带有动量项的 RMSprop，利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率，使每一次迭代学习率都有确定范围，变化平稳。其公式如下：

$$m_t = \mu m_{t-1} + (1 - \mu) g_t$$

$$n_t = v n_{t-1} + (1 - v) g_t^2$$

$$\widehat{m}_t = \frac{m_t}{1 - \mu_t}$$

$$\widehat{n}_t = \frac{n_t}{1 - v_t}$$

$$\Delta \theta_t = - \frac{\widehat{m}_t}{\sqrt{\widehat{n}_t} + \varepsilon} \eta$$

2. 初始化权重

合理的初始化权重能减少迭代次数，以下是三种初始化方法：

方法	特点	优点
----	----	----

高斯分布随机初始化	基于正态分布，权重应围绕零均值分布，并具有一定方差。	确保信号在传播中既不会迅速消失也不会爆炸。
Xavier 初始化	确保前向传播和反向传播过程中激活值和梯度的方差保持一致。	对于使用 tanh 或 sigmoid 激活函数的网络表现较好。
Kaiming 均匀分布初始化 (He 初始化)		对于使用 ReLU 及其变体激活函数的网络表现较好。

3. 批量大小 (batch_size)

一次迭代中用于更新模型参数的样本数量，其大小与模型运行速度和收敛效率有关。

4. 学习率 (Learning Rate, LR) 策略

在训练过程中，学习率可以随迭代次数变化，不同的变化方式可能产生不一样的结果。以下是七种常用的学习率策略：

策略	特点	优点	缺点	公式 (实验中使用)
固定学习率 (Fixed Learning Rate)	整个训练过程中不变。	简单易实现，无需额外设置参数。	可能无法适应训练的各阶段，导致收敛慢或不稳定。	$\eta_t = \eta_0$
指数衰减 (Exponential Decay)	按指数函数衰减。	有助于稳定收敛，尤其是训练后期。	对参数敏感。	$\eta_t = 0.9^t \eta_0$
分段学习率 (Step Decay、Piecewise Constant Decay)	周期性衰减。	可以手动设定调节点，便于控制。	需预先确定衰减点。	$\eta_t = \begin{cases} 0.1^x \eta_0 \\ \eta_0 \end{cases}$
多项式衰减 (Polynomial Decay)	按多项式函数衰减。	平滑且可控的衰减曲线，适合长周期训练。	需要调整多个参数。	$\eta_t = \eta_0 (1 - \frac{t}{T})^{0.9}$
线性衰减 (Linear Decay)	线性地从初始学习率衰减到最终学习率。	简单直观，容易理解。	对于不同类型的任务和数据集，效果存在差异。	$\eta_t = \eta_0 (1 - \frac{t}{T})$

余弦衰减 (Cosine Annealing)	按余弦波形衰减，先快速下降，然后缓慢上升，再下降到最小值。	收敛路径更自然，有助于跳出局部最优解。	需合理设置最小学习率和周期长度。	$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})(1 + \cos(\frac{t\pi}{T}))$
预热 (Warm-up)	训练初期使用非常低的学习率，经若干迭代后逐渐增加到正常水平。	有助于稳定训练过程，特别是对于大规模模型和复杂数据集，可以防止梯度爆炸。	增加了训练初期的复杂性和计算成本。	$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\frac{t}{5}$ <p>, for $t < 5$</p>

三、实验步骤

1. 选择优化器。分别为 SGD（见五、1.）、SGD+动量法（见五、2.）和 Adam（见五、3.）。
2. 分别调节初始化权重，分为 Kaiming 均匀分布初始化、高斯分布随机初始化和 Xavier 初始化。
3. 调节批量大小，进行从 1 到全样本的批量大小的比较（见五、1.（3））。
4. 分别调节学习率策略，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。
5. 尝试其他变种优化器，如 SGD+动量法+权重衰减（见五、2.（4））、SGD+动量法+Nesterov（见五、2.（5））、AMSGrad（见五、3.（4））、AdamW（见五、3.（5））。
6. 对卷积神经网络部分重复以上实验（见六、5.）。

四、基础代码

以下给出代码，各种调试项皆包含在内。网络部分与实验一一致，优化器等调试部分都是调用的相关函数，在代码中进行了注释，不作额外分析。

```
import torch

import torch.nn as nn

from d2l import torch as d2l

import time

import matplotlib.pyplot as plt
```

```
# 初始化权重
```

```
def init_weights(m):
```

```
    if isinstance(m, nn.Linear):
```

```
        # 高斯分布随机初始化
```

```
        nn.init.normal_(m.weight, mean=0.0, std=0.01)
```

```
        # Xavier 初始化
```

```
        # nn.init.xavier_uniform_(m.weight)
```

```
    if m.bias is not None:
```

```
        nn.init.zeros_(m.bias)
```

```
# 预热
```

```
def warmup_lr_scheduler(optimizer, warmup_iters, warmup_factor):
```

```
    def f(x):
```

```
        if x >= warmup_iters:
```

```
            return 1
```

```
        alpha = float(x) / warmup_iters
```

```
        return warmup_factor * (1 - alpha) + alpha
```

```
    return torch.optim.lr_scheduler.LambdaLR(optimizer, f)
```

```
# 训练函数
```

```
def train(net, train_iter, test_iter, num_epochs, loss):
```

```
    start_time = time.time()
```

```
net.to(device)

# 初始化模型参数

params = [p for p in net.parameters() if p.requires_grad]

# 优化器

# SGD

optimizer = torch.optim.SGD(params, lr=learning_rate)

# SGD+动量

# optimizer = torch.optim.SGD(params, lr=learning_rate, momentum=0.9)

# SGD+动量+权重衰减

# optimizer = torch.optim.SGD(params, lr=learning_rate, momentum=0.9,
weight_decay=0.0001)

# SGD+动量+nesterov

# optimizer = torch.optim.SGD(params, lr=learning_rate, momentum=0.9,
nesterov=True)

# Adam

# optimizer = torch.optim.Adam(params, lr=learning_rate)

# Adam+amsgrad

# optimizer = torch.optim.Adam(params, lr=learning_rate, amsgrad=True)

# Adamw

# optimizer = torch.optim.AdamW(params, lr=learning_rate)

# 学习率策略
```



```

# 固定学习率：循环末尾不更新学习率

# 指数衰减

# scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.9)

# 分段学习率

# scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[30, 80],
gamma=0.1)

# 多项式衰减

# lr_lambda = lambda epoch: (1 - float(epoch) / num_epochs) ** 0.9

# scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda)

# 线性衰减

# scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda=lambda
epoch: 1 - epoch / num_epochs)

# 余弦衰减

# scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=num_epochs)

# 预热

warmup_epochs = 5

# scheduler = warmup_lr_scheduler(optimizer, warmup_epochs,
warmup_factor=0.001)

train_loss_list, train_acc_list, test_acc_list = [], [], []

for epoch in range(num_epochs):

```

```
# 训练

net.train()

train_loss, correct, total_samples = 0.0, 0, 0

for X, y in train_iter:

    X, y = X.to(device), y.to(device)

    optimizer.zero_grad()

    y_hat = net(X)

    l = loss(y_hat, y).mean()

    l.backward()

    optimizer.step()

    train_loss += l.item() * y.shape[0]

    _, predicted = torch.max(y_hat.data, 1)

    total_samples += y.size(0)

    correct += (predicted == y).sum().item()

avg_train_loss = train_loss / total_samples

train_acc = correct / total_samples

# 测试

net.eval()

with torch.no_grad():

    test_correct, test_total = 0, 0

    for X, y in test_iter:

        X, y = X.to(device), y.to(device)
```

```

        y_hat = net(X)

        _, predicted = torch.max(y_hat.data, 1)

        test_total += y.size(0)

        test_correct += (predicted == y).sum().item()

    test_acc = test_correct / test_total

    # 学习率变化

    # scheduler.step()

    # 结果

    train_loss_list.append(avg_train_loss)

    train_acc_list.append(train_acc)

    test_acc_list.append(test_acc)

    print(f'Epoch[{epoch + 1}/{num_epochs}], Loss:{avg_train_loss:.4f}, Train
Acc:{train_acc:.4f}, Test Acc:{test_acc:.4f}')

    best_test_acc = max(test_acc_list)

    print(f'最优验证正确率:{best_test_acc:.4f}')

    # 训练耗时

    end_time = time.time()

    training_time = end_time - start_time

    # 绘图

    fig = plt.figure(figsize=(12, 6))

    ax1 = fig.add_subplot(111)

    ax2 = ax1.twinx()

```

```

epochs = range(1, num_epochs + 1)

ax1.plot(epochs, train_loss_list, 'bo-', label="Train Loss")

ax1.set_xlabel('Epochs')

ax1.set_ylabel('Loss')

ax1.set_xlim(1, num_epochs)

ax1.set_ylim(0, max(train_loss_list) * 1.1)

ax1.legend(loc='upper left')

ax2.plot(epochs, train_acc_list, 'r+-', label="Train Acc")

ax2.plot(epochs, test_acc_list, 'g+-', label="Test Acc")

ax2.set_ylabel('Accuracy')

ax2.set_ylim(0, 1.1)

ax2.legend(loc='upper right')

plt.title('Training Progress')

plt.show()

return training_time

```

主函数

```
if __name__ == "__main__":
```

```
    # 参数
```

```
    num_epochs = 10
```

```
    learning_rate = 0.1 # (SGD)为0.1, Adam为0.001
```

```
    batch_size = 256
```

```

# 数据

train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)

# 设备

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 模型

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

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)

)

# net.apply(init_weights)

# 损失函数

loss = nn.CrossEntropyLoss(reduction='mean')

# 训练

training_time = train(net, train_iter, test_iter, num_epochs, loss)

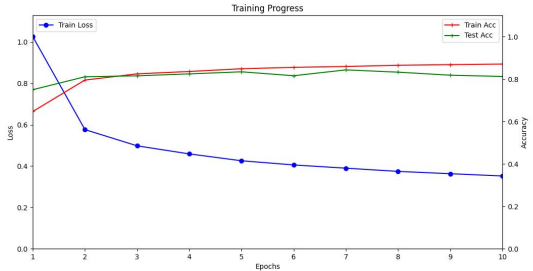
print(f"代码运行时间:{training_time}秒")

```

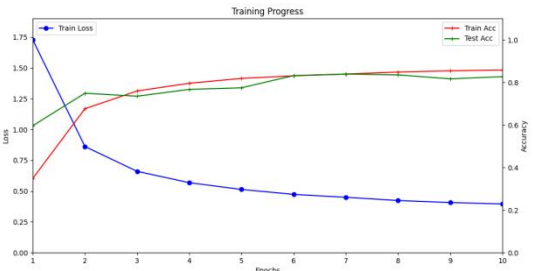
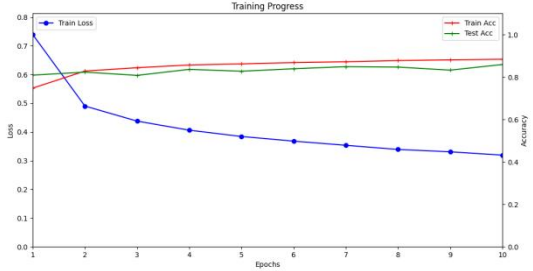
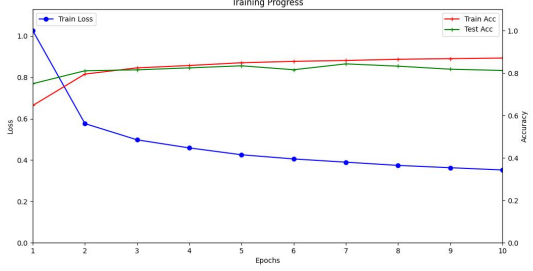
五、调试训练与分析

1. SGD

(1) 结果展示

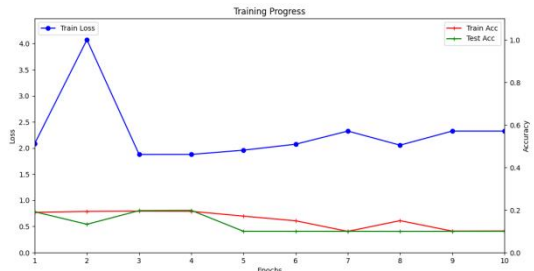
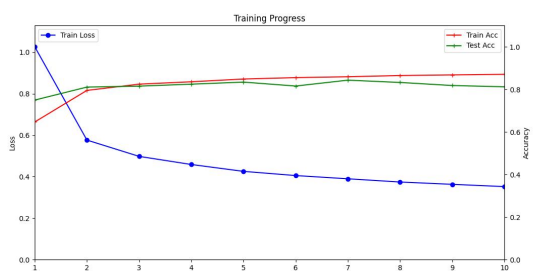
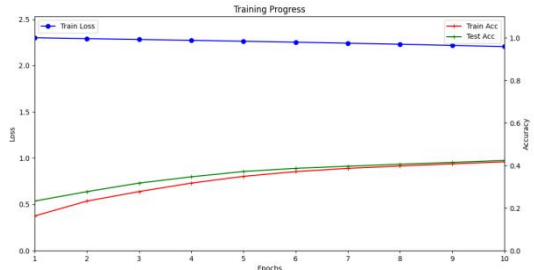
SGD+kaiming 均匀分布初始化 +batchsize=256+固定学习率		结果展示
测试正确率	0.8430	
运行时间 (s)	69.69544982910156	

(2) 初始化权重

统一配置：SGD+batchsize=256+固定学习率		
高斯分布随机初始化		结果展示
测试正确率	0.8388	
运行时间 (s)	68.88738799095154	
Xavier 初始化		结果展示
测试正确率	0.8589	
运行时间 (s)	67.4884262084961	
Kaiming 均匀分布初始化		结果展示
测试正确率	0.8430	
运行时间 (s)	69.69544982910156	

分析：经比较发现，三者的运行时间基本一致，正确率差别也不大。高斯分布随机初始化的正确率相较最低，因为它没有后两者在前、后传播过程中保持一致的特性。Xavier 初始化略优于 Kaiming 均匀分布初始化，这可能是由于它更适配 sigmoid 激活函数。

(3) 批量大小

统一配置：SGD+kaiming 均匀分布初始化+固定学习率		
batchsize=1		结果展示
测试正确率	0.1983	
运行时间 (s)	863.4770958423615	
batchsize=256		结果展示
测试正确率	0.8430	
运行时间 (s)	69.69544982910156	
全样本 (batchsize=60000)		结果展示
测试正确率	0.4237	
运行时间 (s)	84.75873708724976	
分析：以 1 为批量进行训练，效率极低，且难以收敛（单一样本不能很好代表整体数据分布），容易过拟合。而全样本训练，会稳定而缓慢的收敛，可以提供更准确的估计，但缺乏正则化效果，容易过拟合，且需求较大内存。选用适宜的批量大小，能快速收敛到较好的结果。		

仿照 11.5.4 节最后一个图绘制了下图图 5，并总结效果如下表表 1：

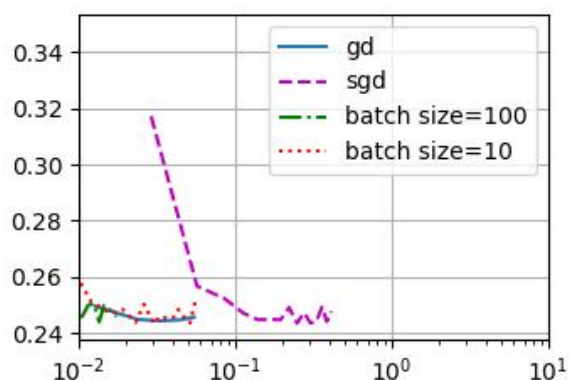


图 5 调试批量大小

表 1 调试批量大小

项目	批量大小	收敛速度	最终损失值
GD (Gradient Descent)	整个数据集	较慢	较低
SGD (Stochastic Gradient Descent)	1	较快	较高且波动较大
Mini-Batch SGD1	100	较快	较低且波动较小
Mini-Batch SGD2	10	较快	略大且波动较大

经比较，批量大小 100 时最优。

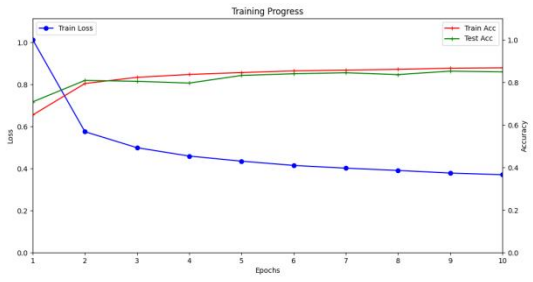
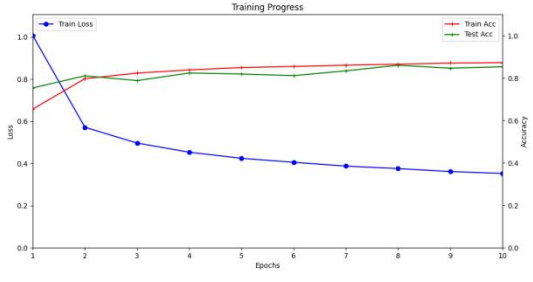
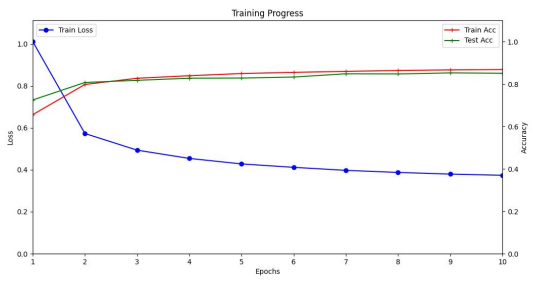
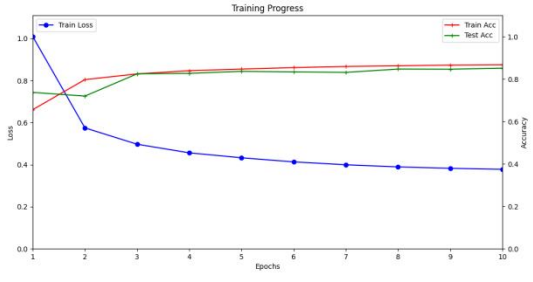
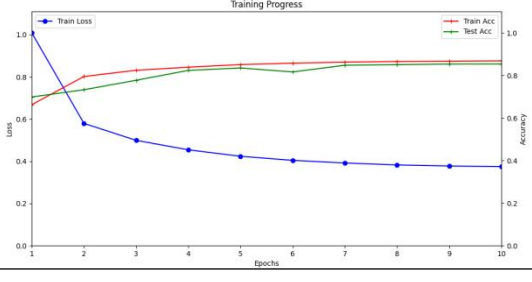
从并行角度总结如下表：

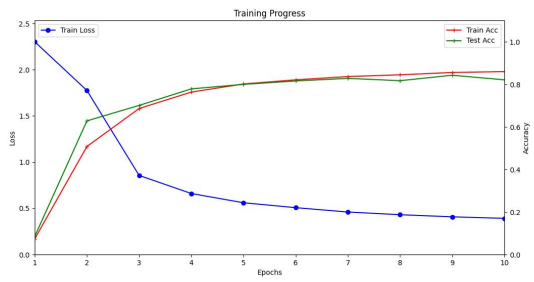
项目	小批量	大批量
硬件利用率（并行性）	利用率高	利用率低
通信开销	小	大
效果	可能导致梯度估计的方差较大，从而导致损失函数的变化路径更加波动	更准确的梯度估计，带来更平滑的损失下降路径
内存要求	第	高
收敛行为	随机性有助于跳出局部最优解，减慢收敛速度	更为稳定的梯度估计，有助于更快地收敛到一个解，但可能更容易陷入局部最优解

(4) 学习率策略

统一配置：SGD+kaiming 均匀分布初始化+batchsize=256

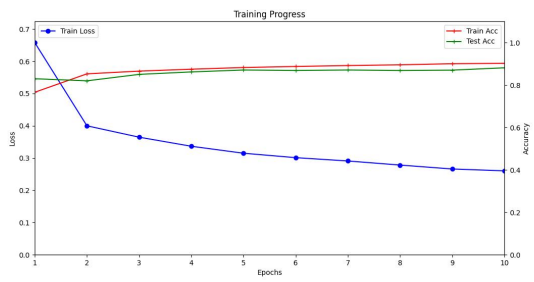
固定学习率		结果展示
测试正确率	0.8430	
运行时间 (s)	69.69544982910156	

指数衰减		结果展示
测试正确率	0.8526	
运行时间 (s)	67.94014978408813	
分段学习率		结果展示
测试正确率	0.8602	
运行时间 (s)	67.32852792739868	
多项式衰减		结果展示
测试正确率	0.8523	
运行时间 (s)	66.33056902885437	
线性衰减		结果展示
测试正确率	0.8510	
运行时间 (s)	66.07158517837524	
余弦衰减		结果展示
测试正确率	0.8534	
运行时间 (s)	67.5896897315979	

预热		结果展示
测试正确率	0.8426	
运行时间 (s)	67.12294793128967	
分析：对于当前任务来说，以上各策略的优点无法完美体现，结果与耗时都大抵相当。但是由展示的曲线可以发现，尤其是在训练初期，不同的策略还是有一定的差异的。		

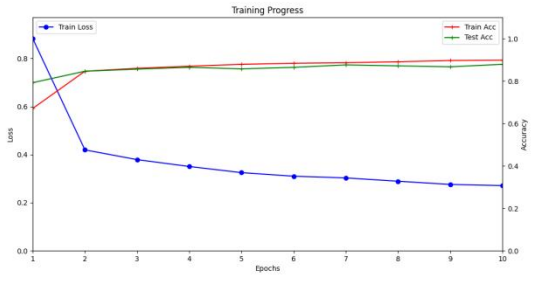

2. SGD+动量法

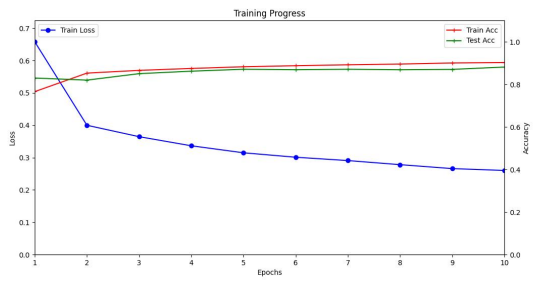
(1) 结果展示

SGD+动量法+kaiming 均匀分布初始化+batchsize=256+固定学习率		结果展示																																												
测试正确率	0.8811	 <p>The graph titled 'Training Progress' displays three metrics over 10 epochs. The left y-axis represents 'Loss' (0.0 to 0.7), and the right y-axis represents 'Accuracy' (0.0 to 1.0). The x-axis represents 'Epochs' (1 to 10). The 'Train Loss' (blue line with diamond markers) starts at approximately 0.65 and decreases steadily to about 0.25. The 'Train Acc' (red line with square markers) starts at approximately 0.50 and increases to about 0.85. The 'Test Acc' (green line with triangle markers) starts at approximately 0.50 and increases to about 0.80.</p> <table border="1"><thead><tr><th>Epoch</th><th>Train Loss</th><th>Train Acc</th><th>Test Acc</th></tr></thead><tbody><tr><td>1</td><td>0.65</td><td>0.50</td><td>0.50</td></tr><tr><td>2</td><td>0.40</td><td>0.55</td><td>0.52</td></tr><tr><td>3</td><td>0.37</td><td>0.57</td><td>0.54</td></tr><tr><td>4</td><td>0.34</td><td>0.58</td><td>0.56</td></tr><tr><td>5</td><td>0.32</td><td>0.59</td><td>0.58</td></tr><tr><td>6</td><td>0.30</td><td>0.60</td><td>0.59</td></tr><tr><td>7</td><td>0.29</td><td>0.61</td><td>0.60</td></tr><tr><td>8</td><td>0.28</td><td>0.62</td><td>0.61</td></tr><tr><td>9</td><td>0.27</td><td>0.63</td><td>0.62</td></tr><tr><td>10</td><td>0.26</td><td>0.64</td><td>0.63</td></tr></tbody></table>	Epoch	Train Loss	Train Acc	Test Acc	1	0.65	0.50	0.50	2	0.40	0.55	0.52	3	0.37	0.57	0.54	4	0.34	0.58	0.56	5	0.32	0.59	0.58	6	0.30	0.60	0.59	7	0.29	0.61	0.60	8	0.28	0.62	0.61	9	0.27	0.63	0.62	10	0.26	0.64	0.63
Epoch	Train Loss		Train Acc	Test Acc																																										
1	0.65	0.50	0.50																																											
2	0.40	0.55	0.52																																											
3	0.37	0.57	0.54																																											
4	0.34	0.58	0.56																																											
5	0.32	0.59	0.58																																											
6	0.30	0.60	0.59																																											
7	0.29	0.61	0.60																																											
8	0.28	0.62	0.61																																											
9	0.27	0.63	0.62																																											
10	0.26	0.64	0.63																																											
运行时间 (s)	67.64800548553467																																													

分析：加了动量法的 SGD 结果明显更好，这可能是因为动量法带来的明确下降方向，使迭代更具效率。

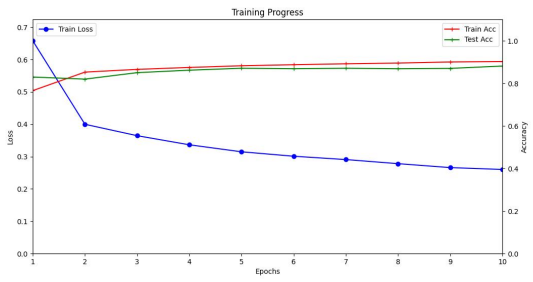
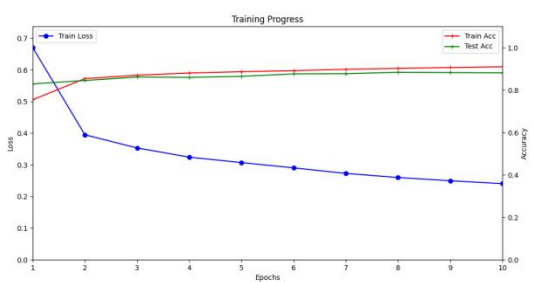
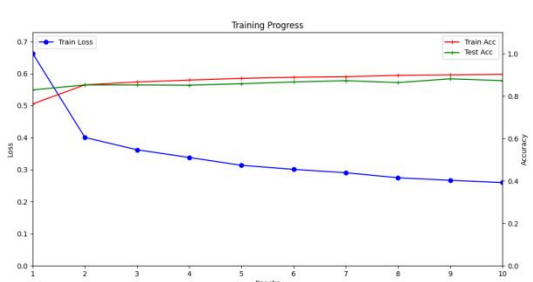

(2) 初始化权重

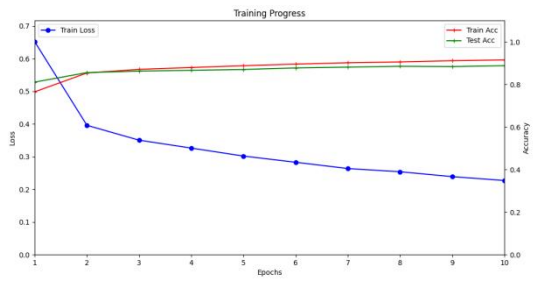
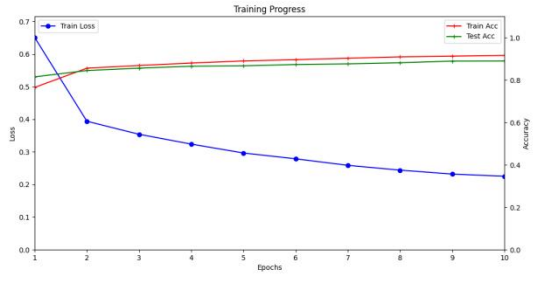
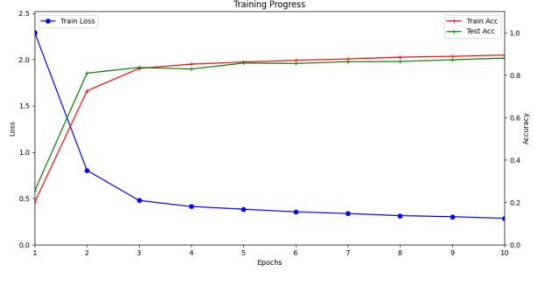
统一配置：SGD+动量法+batchsize=256+固定学习率		
高斯分布随机初始化		结果展示
测试正确率	0.8787	
运行时间 (s)	67.6887309551239	
Xavier 初始化		结果展示
测试正确率	0.8802	
运行时间 (s)	67.46522569656372	

Kaiming 均匀分布初始化		结果展示																																												
测试正确率	0.8811	 <table border="1"><caption>Training Progress Data</caption><thead><tr><th>Epochs</th><th>Train Loss</th><th>Train Acc</th><th>Test Acc</th></tr></thead><tbody><tr><td>1</td><td>0.65</td><td>0.50</td><td>0.55</td></tr><tr><td>2</td><td>0.41</td><td>0.56</td><td>0.54</td></tr><tr><td>3</td><td>0.37</td><td>0.57</td><td>0.56</td></tr><tr><td>4</td><td>0.34</td><td>0.58</td><td>0.57</td></tr><tr><td>5</td><td>0.32</td><td>0.59</td><td>0.58</td></tr><tr><td>6</td><td>0.30</td><td>0.59</td><td>0.58</td></tr><tr><td>7</td><td>0.29</td><td>0.59</td><td>0.58</td></tr><tr><td>8</td><td>0.28</td><td>0.59</td><td>0.58</td></tr><tr><td>9</td><td>0.27</td><td>0.59</td><td>0.58</td></tr><tr><td>10</td><td>0.26</td><td>0.59</td><td>0.58</td></tr></tbody></table>	Epochs	Train Loss	Train Acc	Test Acc	1	0.65	0.50	0.55	2	0.41	0.56	0.54	3	0.37	0.57	0.56	4	0.34	0.58	0.57	5	0.32	0.59	0.58	6	0.30	0.59	0.58	7	0.29	0.59	0.58	8	0.28	0.59	0.58	9	0.27	0.59	0.58	10	0.26	0.59	0.58
Epochs	Train Loss		Train Acc	Test Acc																																										
1	0.65	0.50	0.55																																											
2	0.41	0.56	0.54																																											
3	0.37	0.57	0.56																																											
4	0.34	0.58	0.57																																											
5	0.32	0.59	0.58																																											
6	0.30	0.59	0.58																																											
7	0.29	0.59	0.58																																											
8	0.28	0.59	0.58																																											
9	0.27	0.59	0.58																																											
10	0.26	0.59	0.58																																											
运行时间 (s)	67.64800548553467																																													
分析：结果同 SGD，不做赘述。																																														

(3) 学习率策略

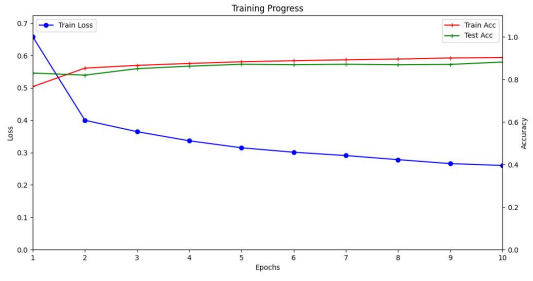
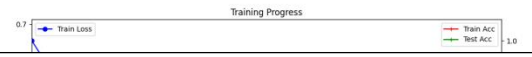
统一配置：SGD+动量法+kaiming 均匀分布初始化+batchsize=256

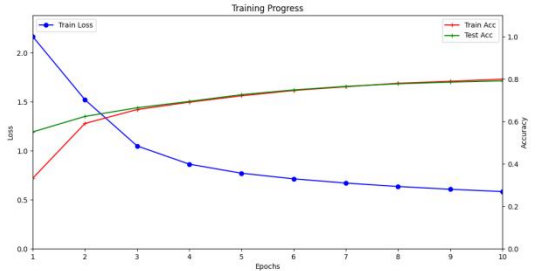
固定学习率		结果展示
测试正确率	0.8811	
运行时间 (s)	67.64800548553467	
指数衰减		结果展示
测试正确率	0.8835	
运行时间 (s)	67.57790231704712	
分段学习率		结果展示
测试正确率	0.8812	
运行时间 (s)	68.08002805709839	
多项式衰减		结果展示
测试正确率	0.8921	
运行时间 (s)	68.34606003761292	

线性衰减		结果展示
测试正确率	0.8876	
运行时间 (s)	67.54336619377136	
余弦衰减		结果展示
测试正确率	0.8895	
运行时间 (s)	67.78437638282776	
预热		结果展示
测试正确率	0.8797	
运行时间 (s)	67.29184865951538	
分析：结果同 SGD，不做赘述。		

(4) 权重衰减

权重衰减 (Weight Decay) 作为一种正则化方式，在损失函数中添加参数范数惩罚项来防止过拟合。

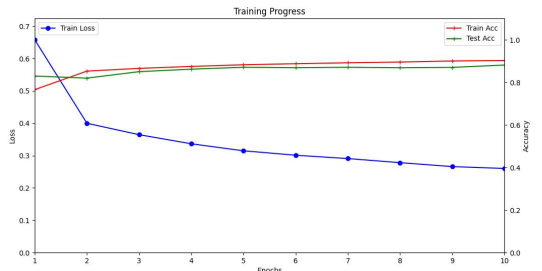
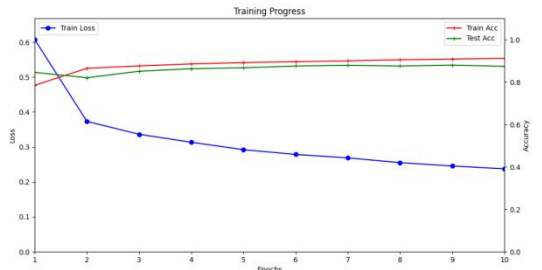
不加权重衰减		结果展示
测试正确率	0.8811	
运行时间 (s)	67.64800548553467	
加权重衰减 (weight_decay=0.0001)		结果展示
测试正确率	0.8759	

运行时间 (s)	67.42794966697693	
加权重衰减 (weight_decay=0.001)		结果展示
测试正确率	0.7914	
运行时间 (s)	67.11317253112793	

分析：不加权重衰减前模型也未过拟合，其加入影响不大。在一定程度上，权重衰减的加入使测试与训练更加贴合。不同大小的权重衰减因子也会对结果产生影响。较小的权重衰减因子，正则化力度较小，使模型复杂，能学习到更精细的数据特征，但有过拟合的风险，适用于大型数据集或特征丰富、复杂的任务。较大的权重衰减因子会强烈地抑制大权重值，使模型倾向于选择更简单的解决方案，减弱表达复杂函数的能力，但同时可以有效减少过拟合，提高模型的泛化能力。

(5) Nesterov

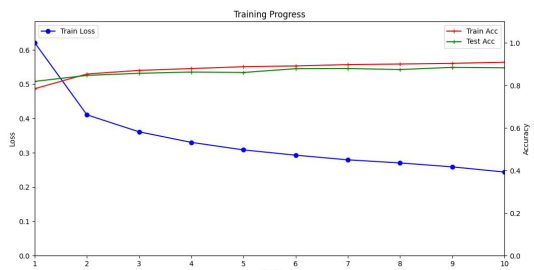
Nesterov（加速梯度，Nesterov Accelerated Gradient, NAG）是一种改进动量梯度下降法。其先根据当前动量进行试探性更新，然后计算在这个新位置上的梯度，而不是像标准动量那样在当前位置计算梯度。其减少了振荡，能更有效地收敛到最小值。

不加 Nesterov		结果展示
测试正确率	0.8811	
运行时间 (s)	67.64800548553467	
加 Nesterov		结果展示
测试正确率	0.8796	
运行时间 (s)	67.88639378547668	
分析：在一定程度上，Nesterov 的加入甚至比不加入要差，这可能和学习率或动量参数的不当设置有关，也可能与初始化权重有关。		

3. Adam



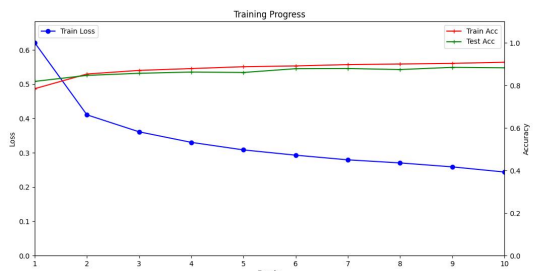
(注意：与前两个使用不同的学习率)

(1) 结果展示

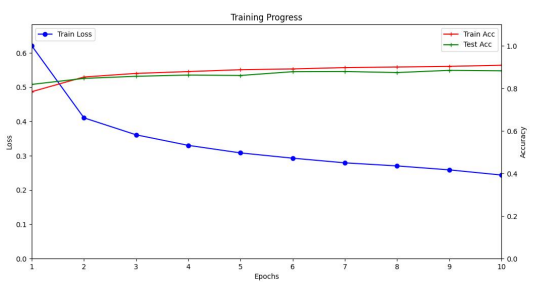
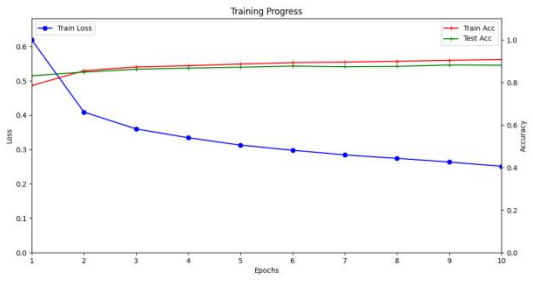
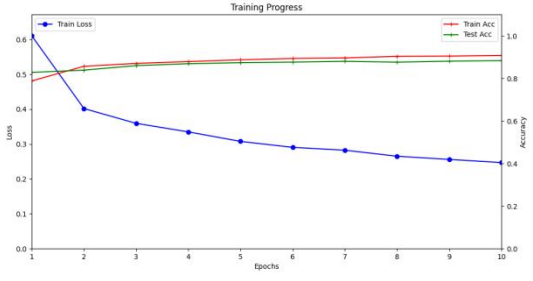
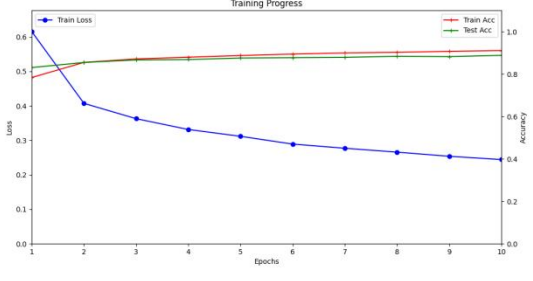
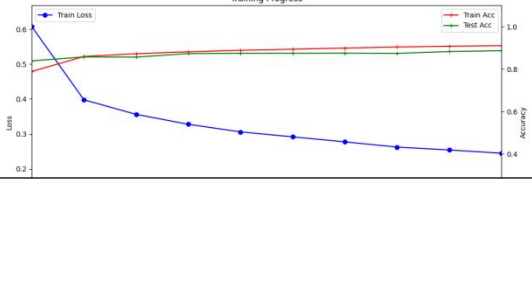
Adam+kaiming 均匀分布初始化+batchsize=256+固定学习率		结果展示																																												
测试正确率	0.8844	 <p>The graph, titled 'Training Progress', plots three metrics over 10 epochs. The left y-axis represents 'Loss' (0.0 to 0.6), and the right y-axis represents 'Accuracy' (0.0 to 1.0). The x-axis is 'Epochs' (1 to 10). Train Loss (blue line with diamond markers) starts at approximately 0.62 and decreases steadily to about 0.24. Train ACC (red line with square markers) starts at approximately 0.48 and increases to about 0.86. Test ACC (green line with circle markers) starts at approximately 0.52 and increases to about 0.85.</p> <table border="1"><thead><tr><th>Epochs</th><th>Train Loss</th><th>Train ACC</th><th>Test ACC</th></tr></thead><tbody><tr><td>1</td><td>0.62</td><td>0.48</td><td>0.52</td></tr><tr><td>2</td><td>0.41</td><td>0.53</td><td>0.53</td></tr><tr><td>3</td><td>0.36</td><td>0.55</td><td>0.54</td></tr><tr><td>4</td><td>0.33</td><td>0.56</td><td>0.55</td></tr><tr><td>5</td><td>0.31</td><td>0.57</td><td>0.56</td></tr><tr><td>6</td><td>0.29</td><td>0.58</td><td>0.57</td></tr><tr><td>7</td><td>0.28</td><td>0.58</td><td>0.57</td></tr><tr><td>8</td><td>0.27</td><td>0.58</td><td>0.57</td></tr><tr><td>9</td><td>0.26</td><td>0.58</td><td>0.57</td></tr><tr><td>10</td><td>0.24</td><td>0.86</td><td>0.85</td></tr></tbody></table>	Epochs	Train Loss	Train ACC	Test ACC	1	0.62	0.48	0.52	2	0.41	0.53	0.53	3	0.36	0.55	0.54	4	0.33	0.56	0.55	5	0.31	0.57	0.56	6	0.29	0.58	0.57	7	0.28	0.58	0.57	8	0.27	0.58	0.57	9	0.26	0.58	0.57	10	0.24	0.86	0.85
Epochs	Train Loss		Train ACC	Test ACC																																										
1	0.62	0.48	0.52																																											
2	0.41	0.53	0.53																																											
3	0.36	0.55	0.54																																											
4	0.33	0.56	0.55																																											
5	0.31	0.57	0.56																																											
6	0.29	0.58	0.57																																											
7	0.28	0.58	0.57																																											
8	0.27	0.58	0.57																																											
9	0.26	0.58	0.57																																											
10	0.24	0.86	0.85																																											
运行时间 (s)	68.0420184135437																																													

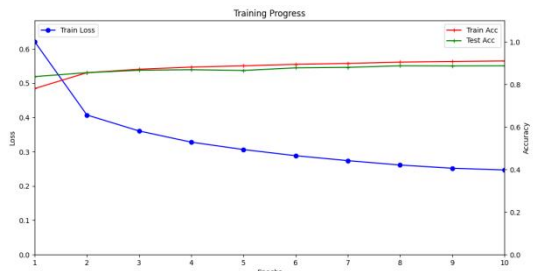
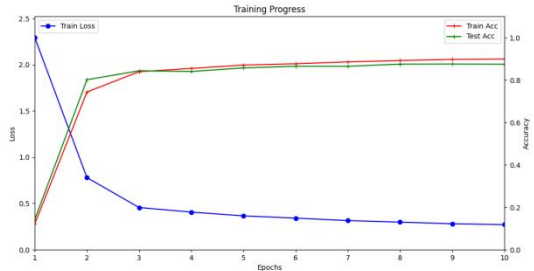
分析：与不加动量的 SGD 相比，Adam 的结果明显更好。这可能是因为变化平稳加快了收敛。		
--	--	--

(2) 初始化权重

统一配置：Adam+动量法+batchsize=256+固定学习率		
高斯分布随机初始化		结果展示
测试正确率	0.8807	
运行时间 (s)	68.67198491096497	
Xavier 初始化		结果展示
测试正确率	0.8850	
运行时间 (s)	68.35895538330078	
Kaiming 均匀分布初始化		结果展示
测试正确率	0.8844	
运行时间 (s)	68.0420184135437	
分析：结果同 SGD，不做赘述。		

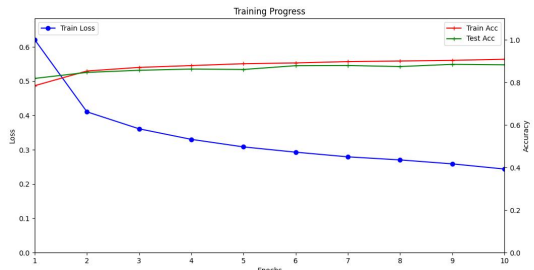
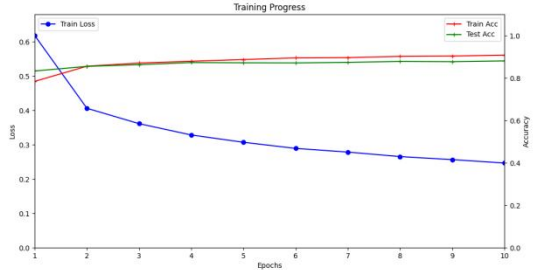
(3) 学习率策略

统一配置：Adam+动量法+kaiming 均匀分布初始化+batchsize=256		
固定学习率		结果展示
测试正确率	0.8844	
运行时间 (s)	68.0420184135437	
指数衰减		结果展示
测试正确率	0.8815	
运行时间 (s)	67.03829288482666	
分段学习率		结果展示
测试正确率	0.8833	
运行时间 (s)	67.07558917999268	
多项式衰减		结果展示
测试正确率	0.8879	
运行时间 (s)	68.37529373168945	
线性衰减		结果展示
测试正确率	0.8862	
运行时间 (s)	68.27660632133484	

余弦衰减		结果展示
测试正确率	0.8871	
运行时间 (s)	68.24571871757507	
预热		结果展示
测试正确率	0.8752	
运行时间 (s)	68.60260462760925	
分析：结果同 SGD，不做赘述。		

(4) AMSGrad

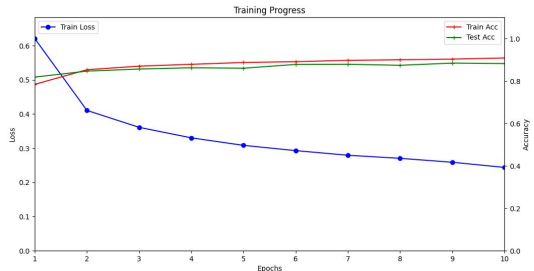
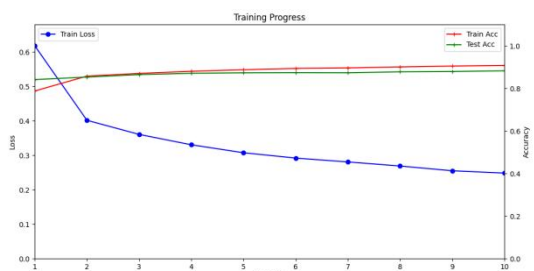
AMSGrad (Amortized Mean Squared Gradient) 是一种改进版 Adam 优化算法，旨在解决后者可能收敛到非最优解的问题。其修正 Adam 中指数移动更新规则，确保了学习率不会单调递减，从而有助于模型跳出局部极小值或鞍点。

不加 AMSGrad		结果展示
测试正确率	0.8844	
运行时间 (s)	68.0420184135437	
加 AMSGrad		结果展示
测试正确率	0.8806	
运行时间 (s)	68.61086177825928	
分析：本问题没有收敛到非最优解的情况，因此 AMSGrad 的使用并无效果。		

(5) AdamW

AdamW 结合了 Adam 的自适应学习率和权重衰减，在更新步骤中明确应用权

重衰减，更有效地防止过拟合，并提高模型的泛化能力。

不加 AdamW		结果展示
测试正确率	0.8844	
运行时间 (s)	68.0420184135437	
加 AdamW		结果展示
测试正确率	0.8823	
运行时间 (s)	68.41013073921204	
分析：本问题没有过拟合，因此 AdamW 的使用并无效果。		

Adam 的权重衰减通过在损失函数中添加 L2 惩罚项实现，其权重衰减因子会乘以学习率并作用于梯度。而 AdamW 的权重衰减直接应用于权重的更新步骤，不依赖于学习率，对所有参数公平一致地施加。

六、附加题

1. 从并行计算角度解释不同批量大小对算法训练速度的影响

见五、1. (3) 表格（最后一个）。

2. SGD+动量法进一步尝试

见五、2. (4) (5)。

3. Adam 进一步尝试 1

见五、3. (4)。

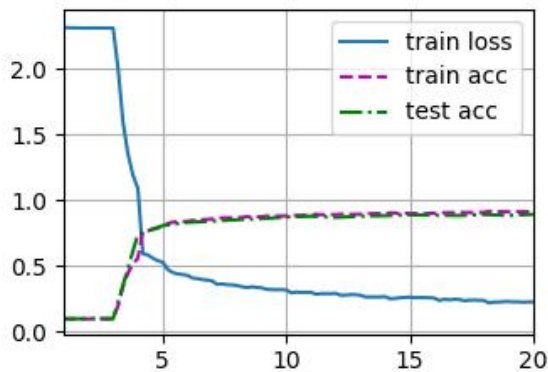
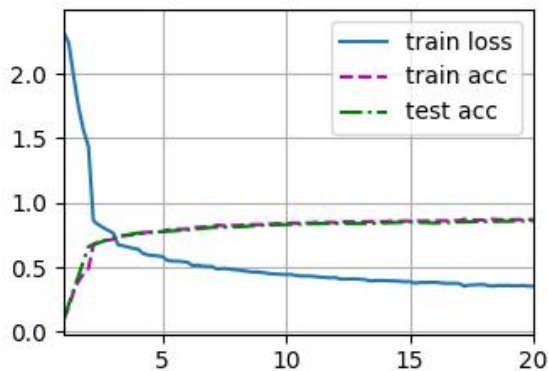
4. Adam 进一步尝试 2

见五、3. (5)。

5. 在卷积神经网络上重复实验

由于以上实验项目过于庞大，这里只针对卷积神经网络和三种优化器进行实验，代码基于实验二代码。

SGD		结果展示
测试正确率	0.843	
运行时间 (s)	102.03244996070862	

SGD+动量法		结果展示
测试正确率	0.895	
运行时间 (s)	100.99193739891052	
Adam		结果展示
测试正确率	0.861	
运行时间 (s)	102.93955135345459	
分析：卷积神经网络的平均耗时要远超多层感知机，而结果优于后者。三种优化器对卷积神经网络的优化效果与对多层感知机的优化效果大体一致。		