安徽大学《深度学习与神经网络》
实验报告 3

学号：	WA2214014	专业：	人工智能	姓名：	杨跃浙
实验日期：	06.01	教师签字：		成绩：	

[实验名称] 卷积神经网络实验

[实验目的]

1．熟悉和掌握卷积核、填充、步幅、通道、汇聚等基本概念和操作

2．熟悉和掌握 LeNet、ResNet 等经典卷积网络的构建、训练和测试

3．熟悉和掌握 BatchNorm

[实验要求]

1．采用 Python 语言基于 PyTorch 深度学习框架进行编程

2．代码可读性强：变量、函数、类等命名可读性强，包含必要的注释

3．提交实验报告要求：

- ➢ 命名方式："学号–姓名–Lab–N"（N 为实验课序号，即：1–6）；
- ➢ 截止时间：下次实验课当晚 23:59；
- ➢ 提交方式：智慧安大–网络教育平台–作业；
- ➢ 按时提交（过时不补）；

[实验内容]

1. 图像卷积

   - 学习、运行和调试参考教材 6.2 小节内容

2. 填充和步幅：

   - 学习、运行和调试参考教材 6.3 小节内容

3. 通道：

   - 学习、运行和调试参考教材 6.4 小节内容

4. 汇聚：

   - 学习、运行和调试参考教材 6.5 小节内容

5. 手工构建 LeNet（**不能直接调用**）实现对 MNIST 数据集的分类

   - 训练集和测试集上的损失曲线、正确率曲线，以及最终正确率
   - 完成 p246 练习题 2
   - 在 LeNet 中加入 BatchNorm 观察运行结果，与加入结果对比分析

6. 残差网络：

   - 学习、运行和调试参考教材 7.6 小节内容

7. 参考资料：

   - **参考教材**：https://zh-v2.d2l.ai/d2l-zh-pytorch.pdf
   - **PyTorch 官方文档**：https://pytorch.org/docs/2.0/；
   - **PyTorch 官方论坛**：https://discuss.pytorch.org/

[实验代码和结果]

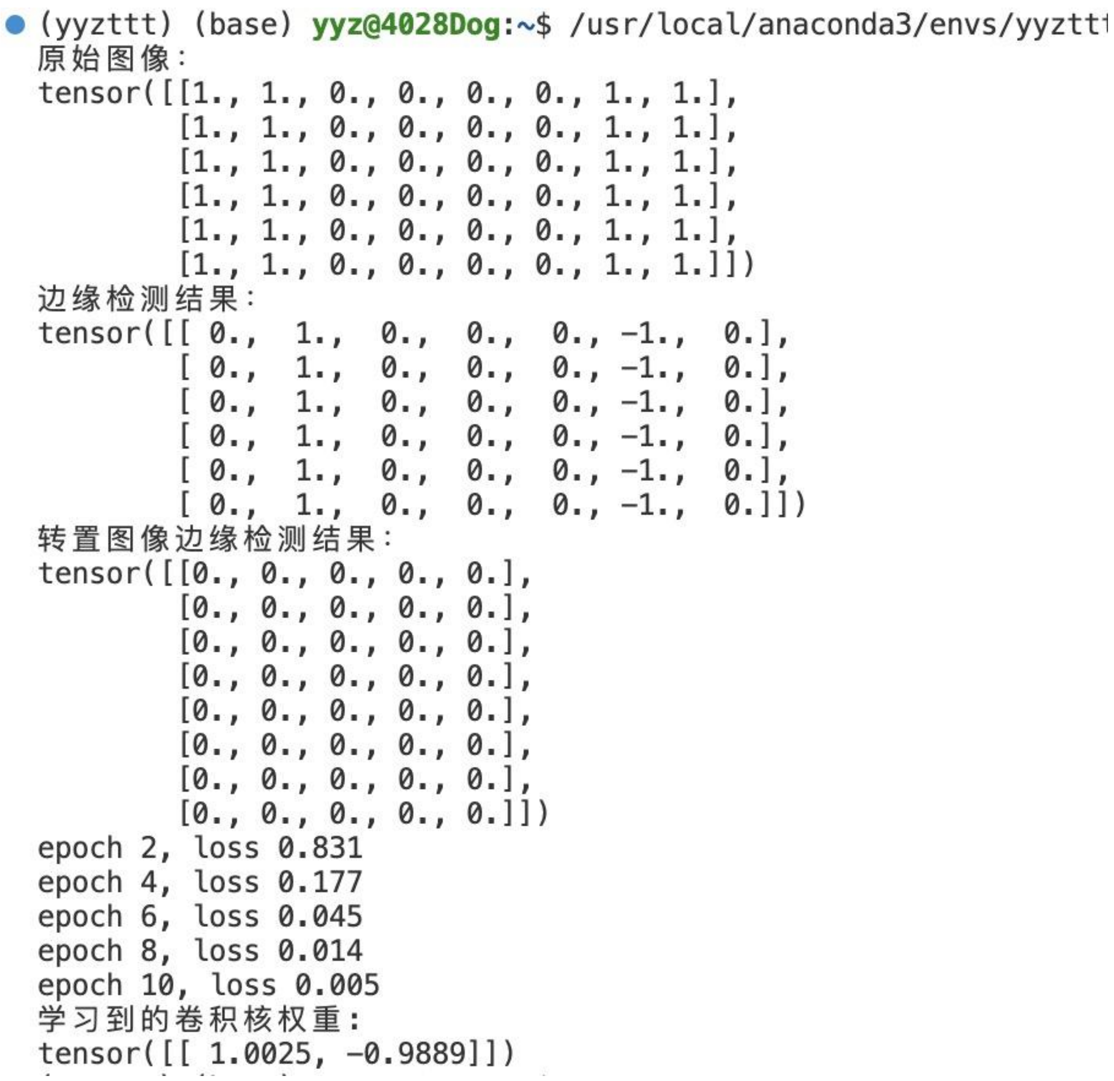
1． 图像卷积

实验代码：

```
import torch
from torch import nn
from d2l import torch as d2l

def corr2d(X, K):
"""计算二维互相关运算"""
h, w = K.shape
Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
for i in range(Y.shape[0]):
for j in range(Y.shape[1]):
Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
return Y

# 实现自定义二维卷积层
class Conv2D(nn.Module):
def __init__(self, kernel_size):
super().__init__()
self.weight = nn.Parameter(torch.rand(kernel_size))
self.bias = nn.Parameter(torch.zeros(1))
def forward(self, x):
return corr2d(x, self.weight) + self.bias

# 边缘检测示例
X = torch.ones((6, 8))
X[:, 2:6] = 0
print("原始图像：")
print(X)

K = torch.tensor([[1.0, -1.0]])
Y = corr2d(X, K)
print("边缘检测结果：")
print(Y)
```

```
# 转置图像进行测试
print("转置图像边缘检测结果: ")
print(corr2d(X.t(), K))

# 从数据学习卷积核
conv2d = nn.Conv2d(1, 1, kernel_size=(1, 2), bias=False)
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2

for i in range(10):
Y_hat = conv2d(X)
l = (Y_hat - Y) ** 2
conv2d.zero_grad()
l.sum().backward()
conv2d.weight.data[:] -= lr * conv2d.weight.grad
if (i + 1) % 2 == 0:
print(f'epoch {i+1}, loss {l.sum():.3f}')

print("学习到的卷积核权重:")
print(conv2d.weight.data.reshape((1, 2)))
```

实验结果:



```
● (yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt
原始图像:
tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])
边缘检测结果:
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
转置图像边缘检测结果:
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
epoch 2, loss 0.831
epoch 4, loss 0.177
epoch 6, loss 0.045
epoch 8, loss 0.014
epoch 10, loss 0.005
学习到的卷积核权重:
tensor([[ 1.0025, -0.9889]])
```

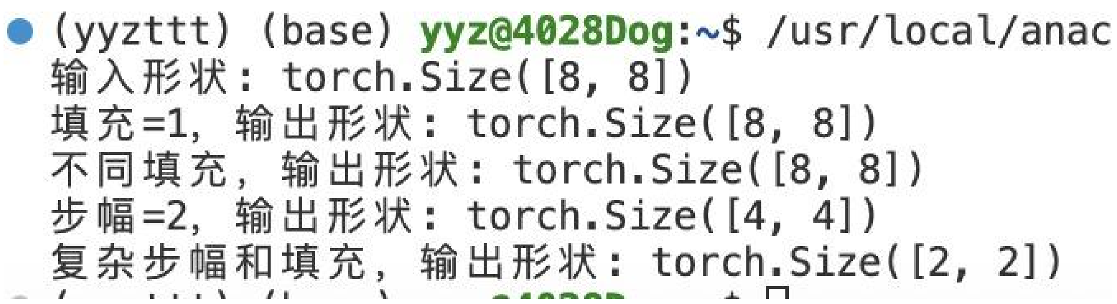
2. 填充和步幅

实验代码:

```
import torch
from torch import nn
from d2l import torch as d2l

# 为了方便起见，定义一个计算卷积层的函数
def comp_conv2d(conv2d, X):
# 这里的（1，1）表示批量大小和通道数都是 1
X = X.reshape((1, 1) + X.shape)
Y = conv2d(X)
# 省略前两个维度：批量大小和通道
return Y.reshape(Y.shape[2:])
```

```python
# 创建输入张量
X = torch.rand(size=(8, 8))
print("输入形状:", X.shape)

# 应用填充以保持输出形状与输入相同
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1)
Y = comp_conv2d(conv2d, X)
print("填充=1, 输出形状:", Y.shape)

# 不同形状的卷积核和不同的填充
conv2d = nn.Conv2d(1, 1, kernel_size=(5, 3), padding=(2, 1))
Y = comp_conv2d(conv2d, X)
print("不同填充, 输出形状:", Y.shape)

# 使用步幅减少输出形状
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
Y = comp_conv2d(conv2d, X)
print("步幅=2, 输出形状:", Y.shape)

# 复杂步幅示例
conv2d = nn.Conv2d(1, 1, kernel_size=(3, 5), padding=(0, 1),
stride=(3, 4))
Y = comp_conv2d(conv2d, X)
print("复杂步幅和填充, 输出形状:", Y.shape)
```

实验结果：



```
● (yyzttt) (base) yyz@4028Dog:~$ /usr/local/anac
  输入形状: torch.Size([8, 8])
  填充=1, 输出形状: torch.Size([8, 8])
  不同填充, 输出形状: torch.Size([8, 8])
  步幅=2, 输出形状: torch.Size([4, 4])
  复杂步幅和填充, 输出形状: torch.Size([2, 2])
```

3. 通道

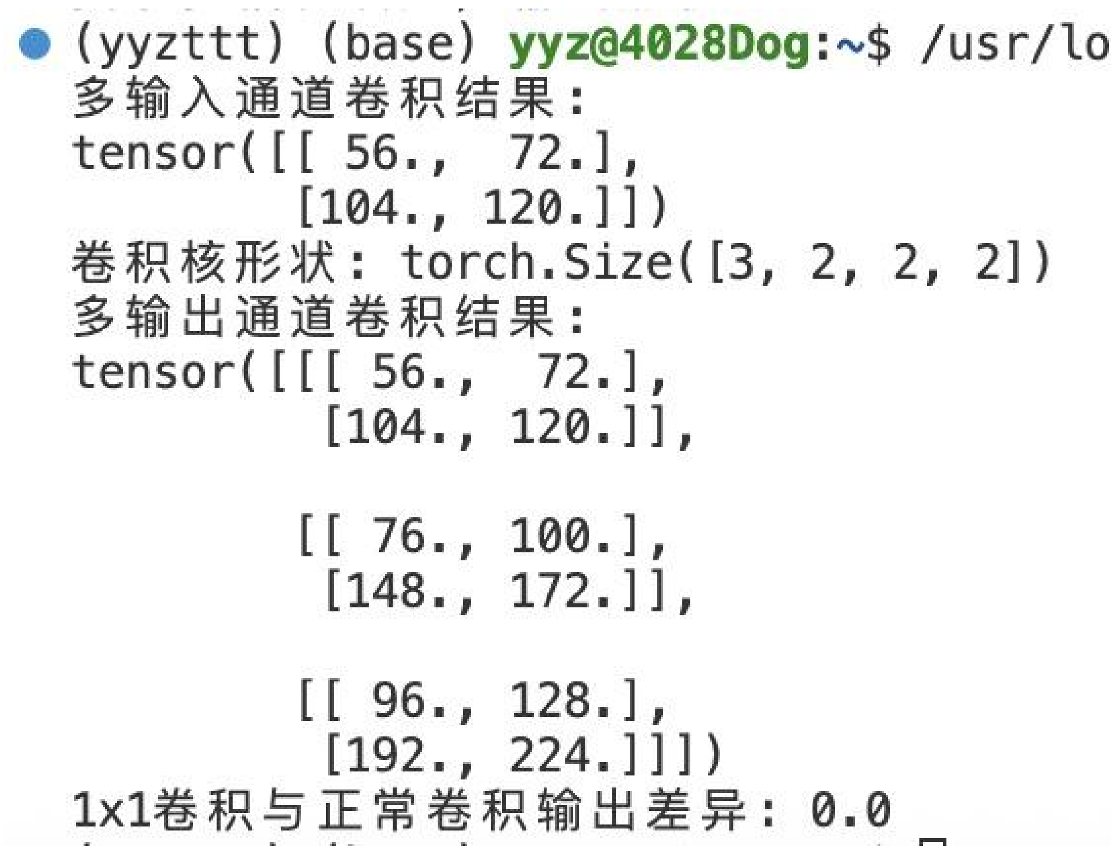
实验代码：

```python
import torch
```

```
from torch import nn
from d2l import torch as d2l

def corr2d(X, K):
h, w = K.shape
Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
for i in range(Y.shape[0]):
for j in range(Y.shape[1]):
Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
return Y

def corr2d_multi_in(X, K):
# 先遍历"X"和"K"的第 0 个维度（通道维度），再把它们加在一起
return sum(d2l.corr2d(x, k) for x, k in zip(X, K))

def corr2d_multi_in_out(X, K):
# 迭代"K"的第 0 个维度，每次都对输入"X"执行互相关运算
# 最后将所有结果都叠加在一起
return torch.stack([corr2d_multi_in(X, k) for k in K], 0)

def corr2d_multi_in_out_1x1(X, K):
c_i, h, w = X.shape
c_o = K.shape[0]
X = X.reshape((c_i, h * w))
K = K.reshape((c_o, c_i))
# 全连接层中的矩阵乘法
Y = torch.matmul(K, X)
return Y.reshape((c_o, h, w))

# 多输入通道示例
X = torch.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0,
7.0, 8.0]],
[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = torch.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0,
4.0]]])

print("多输入通道卷积结果:")
print(corr2d_multi_in(X, K))
```

```
# 多输出通道示例
K = torch.stack((K, K + 1, K + 2), 0)
print("卷积核形状:", K.shape)
print("多输出通道卷积结果:")
print(corr2d_multi_in_out(X, K))

# 1x1 卷积示例
X = torch.normal(0, 1, (3, 3, 3))
K = torch.normal(0, 1, (2, 3, 1, 1))
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
print("1x1 卷积与正常卷积输出差异:", float(torch.abs(Y1 -
Y2).sum()))
```

实验结果:



```
● (yyzttt) (base) yyz@4028Dog:~$ /usr/lo
  多输入通道卷积结果:
  tensor([[ 56.,  72.],
          [104., 120.]])
  卷积核形状: torch.Size([3, 2, 2, 2])
  多输出通道卷积结果:
  tensor([[[ 56.,  72.],
           [104., 120.]],

          [[ 76., 100.],
           [148., 172.]],

          [[ 96., 128.],
           [192., 224.]]])
  1x1卷积与正常卷积输出差异: 0.0
```

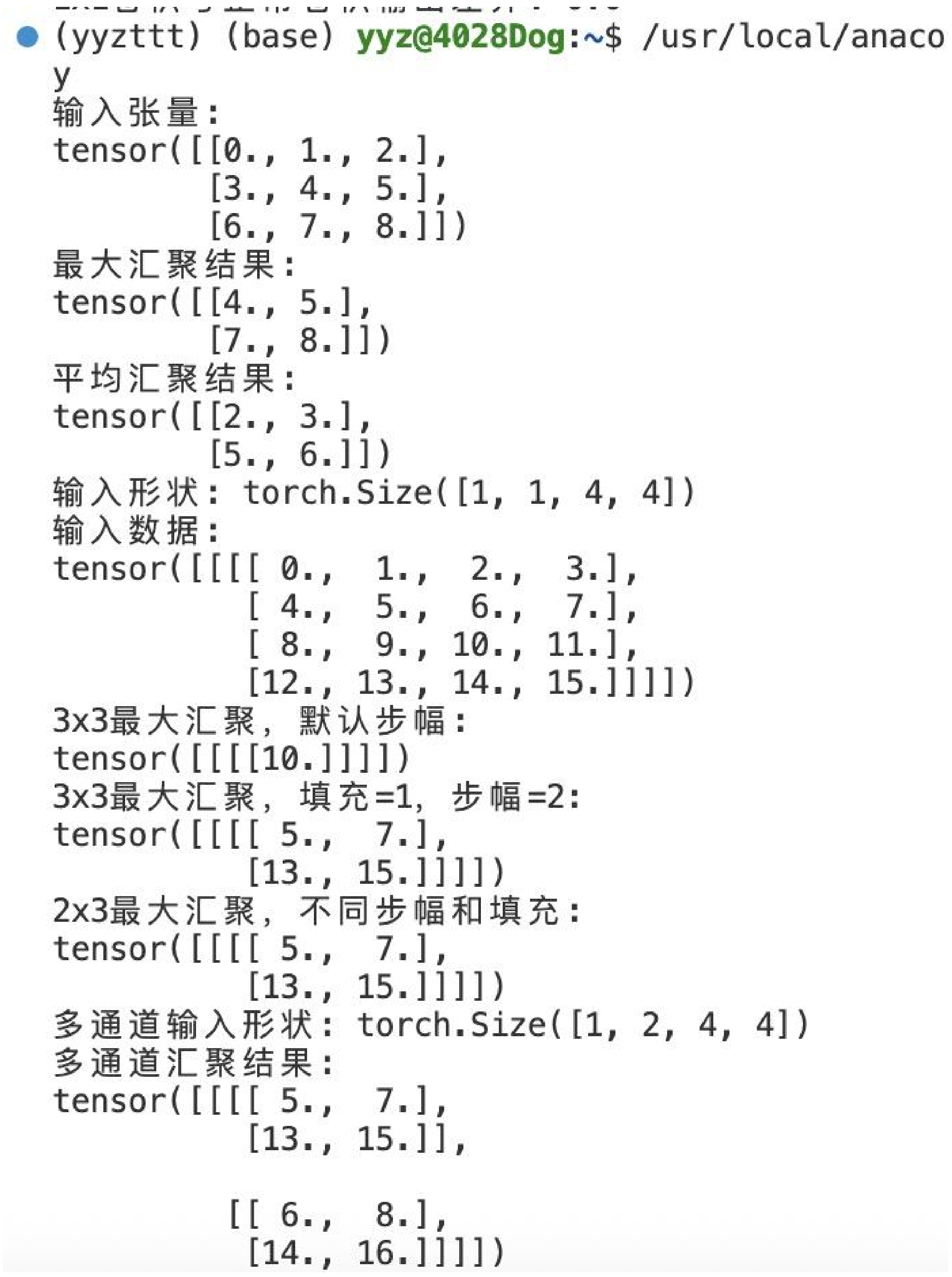
4. 汇聚

实验代码:

```
import torch
from torch import nn
from d2l import torch as d2l

def pool2d(X, pool_size, mode='max'):
p_h, p_w = pool_size
Y = torch.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
for i in range(Y.shape[0]):
for j in range(Y.shape[1]):
if mode == 'max':
Y[i, j] = X[i: i + p_h, j: j + p_w].max()
elif mode == 'avg':
Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
return Y

# 创建输入张量
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0,
8.0]])
print("输入张量:")
print(X)

# 最大汇聚和平均汇聚
print("最大汇聚结果:")
print(pool2d(X, (2, 2)))
print("平均汇聚结果:")
print(pool2d(X, (2, 2), 'avg'))

# 使用 PyTorch 内置汇聚层
X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4,
4))
print("输入形状:", X.shape)
print("输入数据:")
print(X)

# 默认步幅与池化窗口大小相同
pool2d = nn.MaxPool2d(3)
print("3x3 最大汇聚, 默认步幅:")
print(pool2d(X))
```

```
# 指定填充和步幅
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
print("3x3 最大汇聚，填充=1，步幅=2:")
print(pool2d(X))

# 矩形汇聚窗口
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
print("2x3 最大汇聚，不同步幅和填充:")
print(pool2d(X))

# 多通道
X = torch.cat((X, X + 1), 1)
print("多通道输入形状:", X.shape)
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
print("多通道汇聚结果:")
print(pool2d(X))
```

实验结果:



```
● (yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaco
  y
  输入张量:
  tensor([[0., 1., 2.],
          [3., 4., 5.],
          [6., 7., 8.]])
  最大汇聚结果:
  tensor([[4., 5.],
          [7., 8.]])
  平均汇聚结果:
  tensor([[2., 3.],
          [5., 6.]])
  输入形状: torch.Size([1, 1, 4, 4])
  输入数据:
  tensor([[[[ 0.,  1.,  2.,  3.],
            [ 4.,  5.,  6.,  7.],
            [ 8.,  9., 10., 11.],
            [12., 13., 14., 15.]]]])
  3x3最大汇聚, 默认步幅:
  tensor([[[[10.]]]])
  3x3最大汇聚, 填充=1, 步幅=2:
  tensor([[[[ 5.,  7.],
            [13., 15.]]]])
  2x3最大汇聚, 不同步幅和填充:
  tensor([[[[ 5.,  7.],
            [13., 15.]]]])
  多通道输入形状: torch.Size([1, 2, 4, 4])
  多通道汇聚结果:
  tensor([[[[ 5.,  7.],
            [13., 15.]],

           [[ 6.,  8.],
            [14., 16.]]]])
```

5. 手工构建 LeNet（不能直接调用）实现对 MNIST 数据集的分类

实验代码:

```
import torch
from torch import nn
```

```
import torch.nn.functional as F
import matplotlib.pyplot as plt
from torchvision import transforms
from torchvision.datasets import FashionMNIST
from torch.utils.data import DataLoader

plt.ioff() # 关闭交互式模式以防止图表显示

# 基础 LeNet 模型
class LeNet(nn.Module):
def __init__(self):
super(LeNet, self).__init__()
# 第一个卷积层
self.conv1 = nn.Conv2d(1, 6, kernel_size=5, padding=2)
self.sigmoid1 = nn.Sigmoid()
self.avgpool1 = nn.AvgPool2d(kernel_size=2, stride=2)
# 第二个卷积层
self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
self.sigmoid2 = nn.Sigmoid()
self.avgpool2 = nn.AvgPool2d(kernel_size=2, stride=2)
# 全连接层
self.flatten = nn.Flatten()
self.fc1 = nn.Linear(16 * 5 * 5, 120)
self.sigmoid3 = nn.Sigmoid()
self.fc2 = nn.Linear(120, 84)
self.sigmoid4 = nn.Sigmoid()
self.fc3 = nn.Linear(84, 10)
def forward(self, x):
x = self.avgpool1(self.sigmoid1(self.conv1(x)))
x = self.avgpool2(self.sigmoid2(self.conv2(x)))
x = self.flatten(x)
x = self.sigmoid3(self.fc1(x))
x = self.sigmoid4(self.fc2(x))
x = self.fc3(x)
return x

# 带 BatchNorm 的 LeNet
class LeNetBN(nn.Module):
def __init__(self):
```

```
super(LeNetBN, self).__init__()
# 第一个带 BN 的卷积层
self.conv1 = nn.Conv2d(1, 6, kernel_size=5, padding=2)
self.bn1 = nn.BatchNorm2d(6)
self.sigmoid1 = nn.Sigmoid()
self.avgpool1 = nn.AvgPool2d(kernel_size=2, stride=2)
# 第二个带 BN 的卷积层
self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
self.bn2 = nn.BatchNorm2d(16)
self.sigmoid2 = nn.Sigmoid()
self.avgpool2 = nn.AvgPool2d(kernel_size=2, stride=2)
# 带 BN 的全连接层
self.flatten = nn.Flatten()
self.fc1 = nn.Linear(16 * 5 * 5, 120)
self.bn3 = nn.BatchNorm1d(120)
self.sigmoid3 = nn.Sigmoid()
self.fc2 = nn.Linear(120, 84)
self.bn4 = nn.BatchNorm1d(84)
self.sigmoid4 = nn.Sigmoid()
self.fc3 = nn.Linear(84, 10)
def forward(self, x):
x = self.avgpool1(self.sigmoid1(self.bn1(self.conv1(x))))
x = self.avgpool2(self.sigmoid2(self.bn2(self.conv2(x))))
x = self.flatten(x)
x = self.sigmoid3(self.bn3(self.fc1(x)))
x = self.sigmoid4(self.bn4(self.fc2(x)))
x = self.fc3(x)
return x

# 练习题 2: 改进的 LeNet 模型
class ImprovedLeNet(nn.Module):
def __init__(self):
super(ImprovedLeNet, self).__init__()
# 1. 修改卷积窗口大小
# 2. 增加输出通道数
# 3. 使用 ReLU 激活函数
# 4. 增加一个卷积层
# 第一个卷积层
```

```
self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
self.bn1 = nn.BatchNorm2d(32)
self.relu1 = nn.ReLU()
self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
# 第二个卷积层
self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
self.bn2 = nn.BatchNorm2d(64)
self.relu2 = nn.ReLU()
self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
# 第三个卷积层（新增）
self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
self.bn3 = nn.BatchNorm2d(128)
self.relu3 = nn.ReLU()
self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
# 全连接层
self.flatten = nn.Flatten()
self.fc1 = nn.Linear(128 * 3 * 3, 256)
self.bn4 = nn.BatchNorm1d(256)
self.relu4 = nn.ReLU()
self.dropout1 = nn.Dropout(0.5)
self.fc2 = nn.Linear(256, 128)
self.bn5 = nn.BatchNorm1d(128)
self.relu5 = nn.ReLU()
self.dropout2 = nn.Dropout(0.3)
self.fc3 = nn.Linear(128, 10)
def forward(self, x):
x = self.pool1(self.relu1(self.bn1(self.conv1(x))))
x = self.pool2(self.relu2(self.bn2(self.conv2(x))))
x = self.pool3(self.relu3(self.bn3(self.conv3(x))))
x = self.flatten(x)
x = self.dropout1(self.relu4(self.bn4(self.fc1(x))))
x = self.dropout2(self.relu5(self.bn5(self.fc2(x))))
x = self.fc3(x)
return x

# 训练函数
def train_model(net, train_iter, test_iter, num_epochs, lr,
device, scheduler=None):
```

```
def init_weights(m):
if type(m) == nn.Linear or type(m) == nn.Conv2d:
nn.init.xavier_uniform_(m.weight)
net.apply(init_weights)
print('train on', device)
net.to(device)
optimizer = torch.optim.SGD(net.parameters(), lr=lr,
momentum=0.9, weight_decay=5e-4)
loss = nn.CrossEntropyLoss()
# 记录训练和测试精度、损失
train_loss_history = []
train_acc_history = []
test_loss_history = []
test_acc_history = []
for epoch in range(num_epochs):
# 训练
net.train()
train_loss_sum, train_acc_sum, n = 0.0, 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)
l.backward()
optimizer.step()
train_loss_sum += l.item()
train_acc_sum += (y_hat.argmax(dim=1) == y).sum().item()
n += y.size(0)
train_loss = train_loss_sum / len(train_iter)
train_acc = train_acc_sum / n
train_loss_history.append(train_loss)
train_acc_history.append(train_acc)
# 测试
net.eval()
test_loss_sum, test_acc_sum, n = 0.0, 0.0, 0
with torch.no_grad():
for X, y in test_iter:
X, y = X.to(device), y.to(device)
```

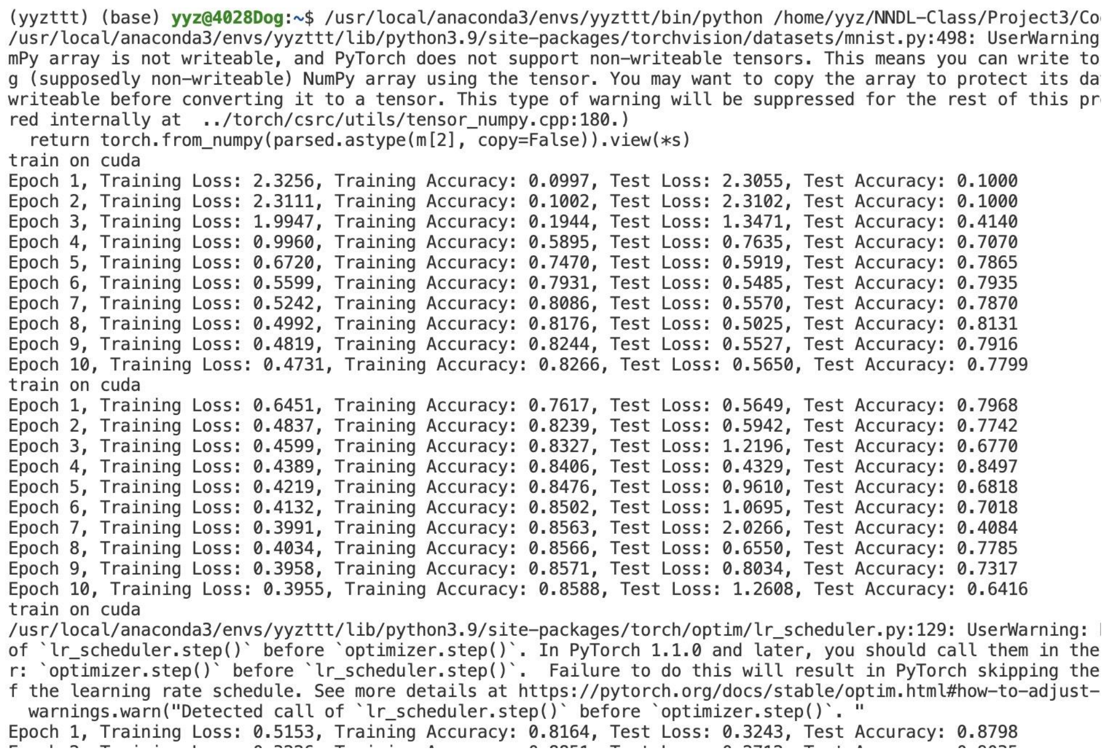
```
y_hat = net(X)
l = loss(y_hat, y)
test_loss_sum += l.item()
test_acc_sum += (y_hat.argmax(dim=1) == y).sum().item()
n += y.size(0)
test_loss = test_loss_sum / len(test_iter)
test_acc = test_acc_sum / n
test_loss_history.append(test_loss)
test_acc_history.append(test_acc)
# 如果提供了学习率调度器，则更新学习率
if scheduler:
scheduler.step()
print(f'Epoch {epoch+1}, Training Loss: {train_loss:.4f},
Training Accuracy: {train_acc:.4f}, '
f'Test Loss: {test_loss:.4f}, Test Accuracy:
{test_acc:.4f}')

return train_loss_history, train_acc_history,
test_loss_history, test_acc_history

# 加载数据集
transform = transforms.Compose([transforms.ToTensor()])
data_root = './NNDL-Class/Project1/Data'
train_dataset = FashionMNIST(root=data_root, train=True,
transform=transform, download=True)
test_dataset = FashionMNIST(root=data_root, train=False,
transform=transform, download=True)

batch_size = 256
train_iter = DataLoader(train_dataset,
batch_size=batch_size, shuffle=True)
test_iter = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

# 设置设备
device = torch.device("cuda" if torch.cuda.is_available()
else "cpu")

# 训练原始 LeNet 模型
```

```
lr, num_epochs = 0.9, 10
net = LeNet()
train_loss, train_acc, test_loss, test_acc = train_model(
net, train_iter, test_iter, num_epochs, lr, device)

# 训练带 BatchNorm 的 LeNet
net_bn = LeNetBN()
train_loss_bn, train_acc_bn, test_loss_bn, test_acc_bn =
train_model(
net_bn, train_iter, test_iter, num_epochs, lr, device)

# 训练改进的 LeNet（练习题 2）
lr_improved = 0.05
improved_net = ImprovedLeNet()
scheduler = torch.optim.lr_scheduler.StepLR(
torch.optim.SGD(improved_net.parameters(), lr=lr_improved,
momentum=0.9, weight_decay=5e-4),
step_size=3, gamma=0.5)
train_loss_improved, train_acc_improved,
test_loss_improved, test_acc_improved = train_model(
improved_net, train_iter, test_iter, num_epochs,
lr_improved, device, scheduler)

# 绘制三个模型的对比图
plt.figure(figsize=(15, 10))

# 损失曲线对比
plt.subplot(2, 2, 1)
plt.plot(range(1, num_epochs+1), train_loss, 'b-',
label='LeNet Training Loss')
plt.plot(range(1, num_epochs+1), train_loss_bn, 'r--',
label='LeNet+BN Training Loss')
plt.plot(range(1, num_epochs+1), train_loss_improved, 'g-.',
label='Improved LeNet Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Comparison')
plt.legend()
```

```python
plt.subplot(2, 2, 2)
plt.plot(range(1, num_epochs+1), test_loss, 'b-',
label='LeNet Test Loss')
plt.plot(range(1, num_epochs+1), test_loss_bn, 'r--',
label='LeNet+BN Test Loss')
plt.plot(range(1,num_epochs+1),test_loss_improved,'g-.',
label='Improved LeNet Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Test Loss Comparison')
plt.legend()

# 准确率曲线对比
plt.subplot(2, 2, 3)
plt.plot(range(1, num_epochs+1), train_acc, 'b-',
label='LeNet Training Accuracy')
plt.plot(range(1, num_epochs+1), train_acc_bn, 'r--',
label='LeNet+BN Training Accuracy')
plt.plot(range(1,num_epochs+1),train_acc_improved,'g-.',
label='Improved LeNet Training Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training Accuracy Comparison')
plt.legend()

plt.subplot(2, 2, 4)
plt.plot(range(1, num_epochs+1), test_acc, 'b-',
label='LeNet Test Accuracy')
plt.plot(range(1, num_epochs+1), test_acc_bn, 'r--',
label='LeNet+BN Test Accuracy')
plt.plot(range(1, num_epochs+1), test_acc_improved, 'g-.',
label='Improved LeNet Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Test Accuracy Comparison')
plt.legend()

plt.tight_layout()
plt.savefig('./NNDL-Class/Project3/Result/lenet_all_comparison.png')

# 打印最终准确率对比
print("\nModel Performance Comparison:")
print("-" * 50)
print(f"Original LeNet Final Test Accuracy: {test_acc[-1]:.4f}")
print(f"LeNet with BatchNorm Final Test Accuracy: {test_acc_bn[-1]:.4f}")
print(f"Improved LeNet Final Test Accuracy: {test_acc_improved[-1]:.4f}")
```

实验结果：

第一次：

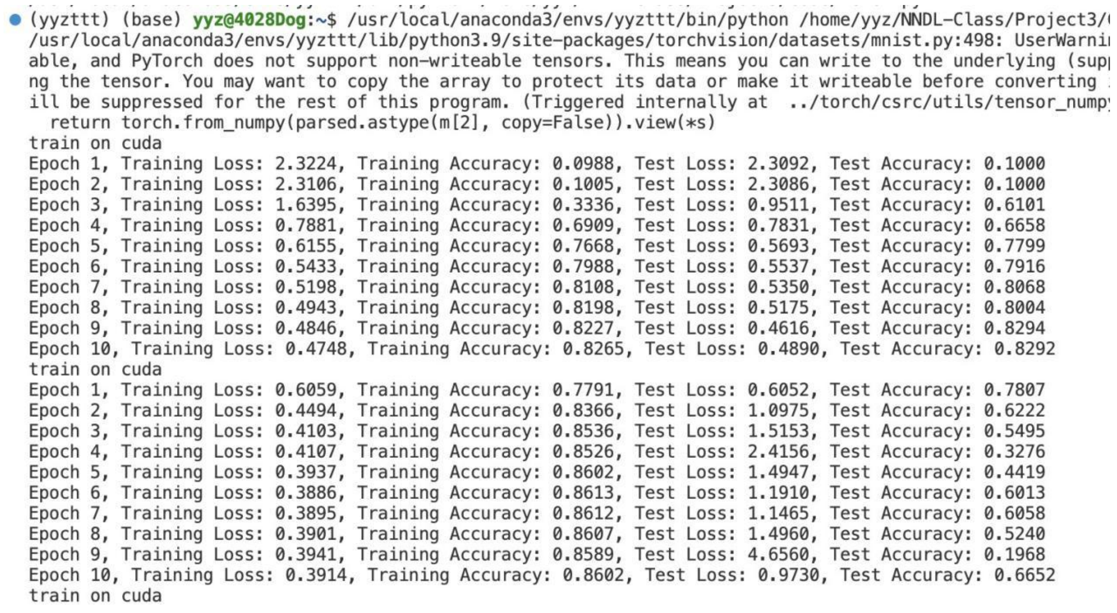


```
(yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt/bin/python /home/yyz/NNDL-Class/Project3/Co
/usr/local/anaconda3/envs/yyzttt/lib/python3.9/site-packages/torchvision/datasets/mnist.py:498: UserWarning
mPy array is not writeable, and PyTorch does not support non-writeable tensors. This means you can write to
g (supposedly non-writeable) NumPy array using the tensor. You may want to copy the array to protect its da
writeable before converting it to a tensor. This type of warning will be suppressed for the rest of this pr
red internally at  ../torch/csrc/utils/tensor_numpy.cpp:180.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
train on cuda
Epoch 1, Training Loss: 2.3256, Training Accuracy: 0.0997, Test Loss: 2.3055, Test Accuracy: 0.1000
Epoch 2, Training Loss: 2.3111, Training Accuracy: 0.1002, Test Loss: 2.3102, Test Accuracy: 0.1000
Epoch 3, Training Loss: 1.9947, Training Accuracy: 0.1944, Test Loss: 1.3471, Test Accuracy: 0.4140
Epoch 4, Training Loss: 0.9960, Training Accuracy: 0.5895, Test Loss: 0.7635, Test Accuracy: 0.7070
Epoch 5, Training Loss: 0.6720, Training Accuracy: 0.7470, Test Loss: 0.5919, Test Accuracy: 0.7865
Epoch 6, Training Loss: 0.5599, Training Accuracy: 0.7931, Test Loss: 0.5485, Test Accuracy: 0.7935
Epoch 7, Training Loss: 0.5242, Training Accuracy: 0.8086, Test Loss: 0.5570, Test Accuracy: 0.7870
Epoch 8, Training Loss: 0.4992, Training Accuracy: 0.8176, Test Loss: 0.5025, Test Accuracy: 0.8131
Epoch 9, Training Loss: 0.4819, Training Accuracy: 0.8244, Test Loss: 0.5527, Test Accuracy: 0.7916
Epoch 10, Training Loss: 0.4731, Training Accuracy: 0.8266, Test Loss: 0.5650, Test Accuracy: 0.7799
train on cuda
Epoch 1, Training Loss: 0.6451, Training Accuracy: 0.7617, Test Loss: 0.5649, Test Accuracy: 0.7968
Epoch 2, Training Loss: 0.4837, Training Accuracy: 0.8239, Test Loss: 0.5942, Test Accuracy: 0.7742
Epoch 3, Training Loss: 0.4599, Training Accuracy: 0.8327, Test Loss: 1.2196, Test Accuracy: 0.6770
Epoch 4, Training Loss: 0.4389, Training Accuracy: 0.8406, Test Loss: 0.4329, Test Accuracy: 0.8497
Epoch 5, Training Loss: 0.4219, Training Accuracy: 0.8476, Test Loss: 0.9610, Test Accuracy: 0.6818
Epoch 6, Training Loss: 0.4132, Training Accuracy: 0.8502, Test Loss: 1.0695, Test Accuracy: 0.7018
Epoch 7, Training Loss: 0.3991, Training Accuracy: 0.8563, Test Loss: 2.0266, Test Accuracy: 0.4084
Epoch 8, Training Loss: 0.4034, Training Accuracy: 0.8566, Test Loss: 0.6550, Test Accuracy: 0.7785
Epoch 9, Training Loss: 0.3958, Training Accuracy: 0.8571, Test Loss: 0.8034, Test Accuracy: 0.7317
Epoch 10, Training Loss: 0.3955, Training Accuracy: 0.8588, Test Loss: 1.2608, Test Accuracy: 0.6416
train on cuda
/usr/local/anaconda3/envs/yyzttt/lib/python3.9/site-packages/torch/optim/lr_scheduler.py:129: UserWarning:
of `lr_scheduler.step()` before `optimizer.step()`. In PyTorch 1.1.0 and later, you should call them in the
r: `optimizer.step()` before `lr_scheduler.step()`.  Failure to do this will result in PyTorch skipping the
f the learning rate schedule. See more details at https://pytorch.org/docs/stable/optim.html#how-to-adjust-
  warnings.warn("Detected call of `lr_scheduler.step()` before `optimizer.step()`. "
Epoch 1, Training Loss: 0.5153, Training Accuracy: 0.8164, Test Loss: 0.3243, Test Accuracy: 0.8798
```

Original LeNet：

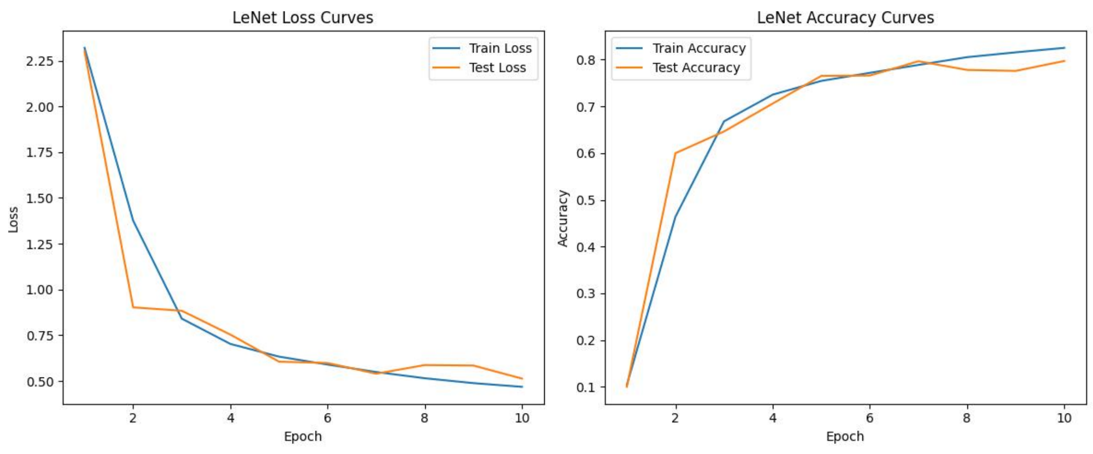
```
Epoch 1, Training Loss: 2.3256, Training Accuracy: 0.0997, Test Loss: 2.3055, Test Accuracy: 0.1000
Epoch 2, Training Loss: 2.3111, Training Accuracy: 0.1002, Test Loss: 2.3102, Test Accuracy: 0.1000
Epoch 3, Training Loss: 1.9947, Training Accuracy: 0.1944, Test Loss: 1.3471, Test Accuracy: 0.4140
Epoch 4, Training Loss: 0.9960, Training Accuracy: 0.5895, Test Loss: 0.7635, Test Accuracy: 0.7070
Epoch 5, Training Loss: 0.6720, Training Accuracy: 0.7470, Test Loss: 0.5919, Test Accuracy: 0.7865
Epoch 6, Training Loss: 0.5599, Training Accuracy: 0.7931, Test Loss: 0.5485, Test Accuracy: 0.7935
Epoch 7, Training Loss: 0.5242, Training Accuracy: 0.8086, Test Loss: 0.5570, Test Accuracy: 0.7870
Epoch 8, Training Loss: 0.4992, Training Accuracy: 0.8176, Test Loss: 0.5025, Test Accuracy: 0.8131
Epoch 9，Training Loss：0.4819，Training Accuracy：0.8244，Test Loss：0.5527，Test Accuracy：0.7916
Epoch 10，Training Loss：0.4731，Training Accuracy：0.8266，Test Loss：0.5650，Test Accuracy：0.7799
LeNet with BatchNorm：
Epoch 1，Training Loss：0.6451，Training Accuracy：0.7617，Test Loss：0.5649，Test Accuracy：0.7968
Epoch 2，Training Loss：0.4837，Training Accuracy：0.8239，Test Loss：0.5942，Test Accuracy：0.7742
Epoch 3，Training Loss：0.4599，Training Accuracy：0.8327，Test Loss：1.2196，Test Accuracy：0.6770
Epoch 4，Training Loss：0.4389，Training Accuracy：0.8406，Test Loss：0.4329，Test Accuracy：0.8497
Epoch 5，Training Loss：0.4219，Training Accuracy：0.8476，Test Loss：0.9610，Test Accuracy：0.6818
Epoch 6，Training Loss：0.4132，Training Accuracy：0.8502，Test Loss：1.0695，Test Accuracy：0.7018
Epoch 7，Training Loss：0.3991，Training Accuracy：0.8563，Test Loss：2.0266，Test Accuracy：0.4084
Epoch 8，Training Loss：0.4034，Training Accuracy：0.8566，Test Loss：0.6550，Test Accuracy：0.7785
Epoch 9，Training Loss：0.3958，Training Accuracy：0.8571，Test Loss：0.8034，Test Accuracy：0.7317
Epoch 10，Training Loss：0.3955，Training Accuracy：0.8588，Test Loss：1.2608，Test Accuracy：0.6416
Improved LeNet：
Epoch 1，Training Loss：0.5153，Training Accuracy：0.8164，Test Loss：0.3243，Test Accuracy：0.8798
Epoch 2，Training Loss：0.3226，Training Accuracy：0.8851，Test Loss：0.2712，Test Accuracy：0.9035
Epoch 3，Training Loss：0.2738，Training Accuracy：0.9023，Test Loss：0.3028，Test Accuracy：0.8903
Epoch 4，Training Loss：0.2468，Training Accuracy：0.9108，Test Loss：0.2753，Test Accuracy：0.9014
Epoch 5，Training Loss：0.2264，Training Accuracy：0.9194，Test Loss：0.2414，Test Accuracy：0.9107
Epoch 6，Training Loss：0.2139，Training Accuracy：0.9244，Test Loss：0.2445，Test Accuracy：0.9095
Epoch 7，Training Loss：0.1960，Training Accuracy：0.9302，Test Loss：0.2362，Test Accuracy：0.9162
Epoch 8，Training Loss：0.1877，Training Accuracy：0.9319，Test Loss：0.2397，Test Accuracy：0.9121
Epoch 9，Training Loss：0.1731，Training Accuracy：0.9385，Test Loss：0.2788，Test Accuracy：0.8975
Epoch 10，Training Loss：0.1688，Training Accuracy：0.9397，Test Loss：0.2449，Test Accuracy：0.9159

Model Performance Comparison：
——————————————————————————————————————————————————
Original LeNet Final Test Accuracy：0.7799
LeNet with BatchNorm Final Test Accuracy：0.6416
Improved LeNet Final Test Accuracy：0.9159
```

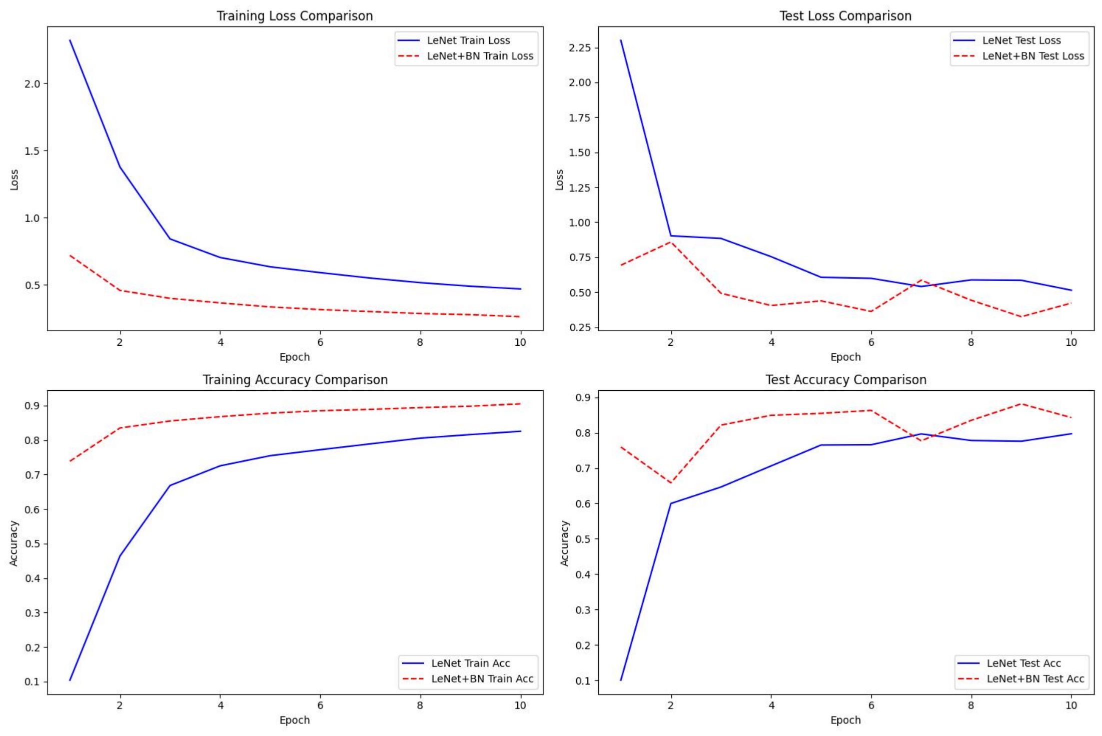
第二次：

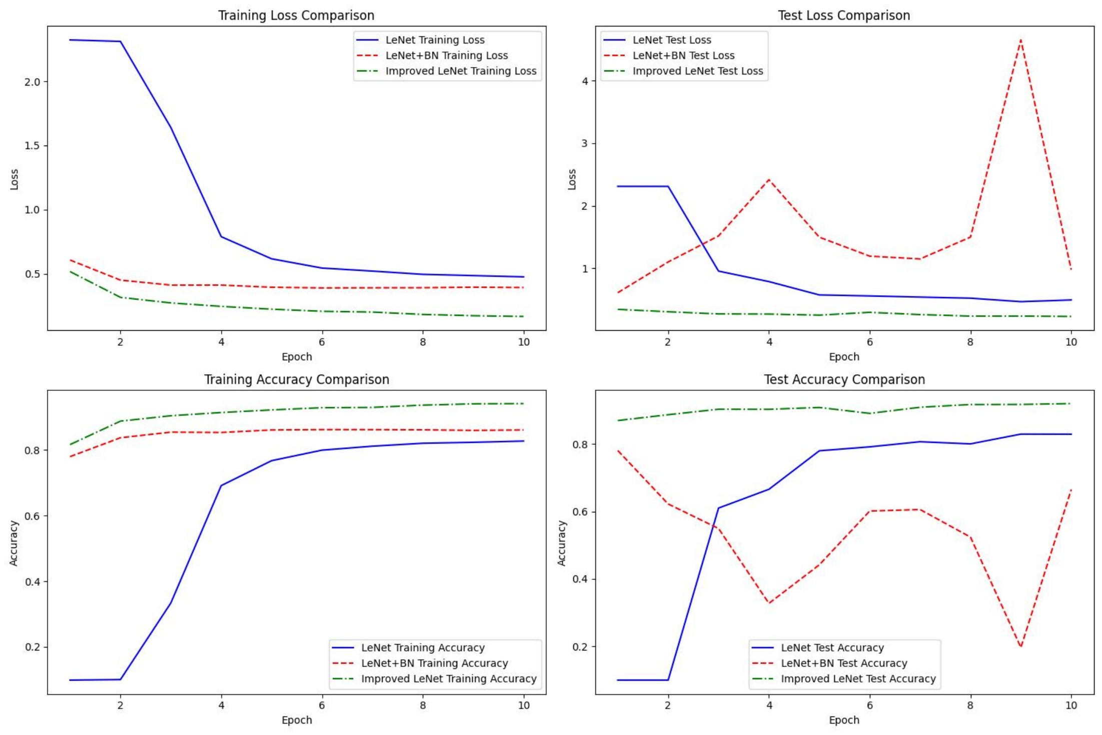


```
● (yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt/bin/python /home/yyz/NNDL-Class/Project3/
  /usr/local/anaconda3/envs/yyzttt/lib/python3.9/site-packages/torchvision/datasets/mnist.py:498: UserWarni
  able, and PyTorch does not support non-writeable tensors. This means you can write to the underlying (sup
  ng the tensor. You may want to copy the array to protect its data or make it writeable before converting
  ill be suppressed for the rest of this program. (Triggered internally at  ../torch/csrc/utils/tensor_nump
    return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
  train on cuda
  Epoch 1, Training Loss: 2.3224, Training Accuracy: 0.0988, Test Loss: 2.3092, Test Accuracy: 0.1000
  Epoch 2, Training Loss: 2.3106, Training Accuracy: 0.1005, Test Loss: 2.3086, Test Accuracy: 0.1000
  Epoch 3, Training Loss: 1.6395, Training Accuracy: 0.3336, Test Loss: 0.9511, Test Accuracy: 0.6101
  Epoch 4, Training Loss: 0.7881, Training Accuracy: 0.6909, Test Loss: 0.7831, Test Accuracy: 0.6658
  Epoch 5, Training Loss: 0.6155, Training Accuracy: 0.7668, Test Loss: 0.5693, Test Accuracy: 0.7799
  Epoch 6, Training Loss: 0.5433, Training Accuracy: 0.7988, Test Loss: 0.5537, Test Accuracy: 0.7916
  Epoch 7, Training Loss: 0.5198, Training Accuracy: 0.8108, Test Loss: 0.5350, Test Accuracy: 0.8068
  Epoch 8, Training Loss: 0.4943, Training Accuracy: 0.8198, Test Loss: 0.5175, Test Accuracy: 0.8004
  Epoch 9, Training Loss: 0.4846, Training Accuracy: 0.8227, Test Loss: 0.4616, Test Accuracy: 0.8294
  Epoch 10, Training Loss: 0.4748, Training Accuracy: 0.8265, Test Loss: 0.4890, Test Accuracy: 0.8292
  train on cuda
  Epoch 1, Training Loss: 0.6059, Training Accuracy: 0.7791, Test Loss: 0.6052, Test Accuracy: 0.7807
  Epoch 2, Training Loss: 0.4494, Training Accuracy: 0.8366, Test Loss: 1.0975, Test Accuracy: 0.6222
  Epoch 3, Training Loss: 0.4103, Training Accuracy: 0.8536, Test Loss: 1.5153, Test Accuracy: 0.5495
  Epoch 4, Training Loss: 0.4107, Training Accuracy: 0.8526, Test Loss: 2.4156, Test Accuracy: 0.3276
  Epoch 5, Training Loss: 0.3937, Training Accuracy: 0.8602, Test Loss: 1.4947, Test Accuracy: 0.4419
  Epoch 6, Training Loss: 0.3886, Training Accuracy: 0.8613, Test Loss: 1.1910, Test Accuracy: 0.6013
  Epoch 7, Training Loss: 0.3895, Training Accuracy: 0.8612, Test Loss: 1.1465, Test Accuracy: 0.6058
  Epoch 8, Training Loss: 0.3901, Training Accuracy: 0.8607, Test Loss: 1.4960, Test Accuracy: 0.5240
  Epoch 9, Training Loss: 0.3941, Training Accuracy: 0.8589, Test Loss: 4.6560, Test Accuracy: 0.1968
  Epoch 10, Training Loss: 0.3914, Training Accuracy: 0.8602, Test Loss: 0.9730, Test Accuracy: 0.6652
  train on cuda
```

```
Original LeNet：
Epoch 1，Training Loss：2.3224，Training Accuracy：0.0988，Test Loss：2.3092，Test Accuracy：0.1000
Epoch 2，Training Loss：2.3106，Training Accuracy：0.1005，Test Loss：2.3086，Test Accuracy：0.1000
Epoch 3，Training Loss：1.6395，Training Accuracy：0.3336，Test Loss：0.9511，Test Accuracy：0.6101
Epoch 4，Training Loss：0.7881，Training Accuracy：0.6909，Test Loss：0.7831，Test Accuracy：0.6658
Epoch 5，Training Loss：0.6155，Training Accuracy：0.7668，Test Loss：0.5693，Test Accuracy：0.7799
Epoch 6， Training Loss： 0.5433， Training Accuracy： 0.7988， Test Loss： 0.5537， Test Accuracy： 0.7916
Epoch 7， Training Loss： 0.5198， Training Accuracy： 0.8108， Test Loss： 0.5350， Test Accuracy： 0.8068
Epoch 8， Training Loss： 0.4943， Training Accuracy： 0.8198， Test Loss： 0.5175， Test Accuracy： 0.8004
Epoch 9， Training Loss： 0.4846， Training Accuracy： 0.8227， Test Loss： 0.4616， Test Accuracy： 0.8294
Epoch 10， Training Loss： 0.4748， Training Accuracy： 0.8265， Test Loss： 0.4890， Test Accuracy： 0.8292
LeNet with BatchNorm：
Epoch 1， Training Loss： 0.6059， Training Accuracy： 0.7791， Test Loss： 0.6052， Test Accuracy： 0.7807
Epoch 2， Training Loss： 0.4494， Training Accuracy： 0.8366， Test Loss： 1.0975， Test Accuracy： 0.6222
Epoch 3， Training Loss： 0.4103， Training Accuracy： 0.8536， Test Loss： 1.5153， Test Accuracy： 0.5495
Epoch 4， Training Loss： 0.4107， Training Accuracy： 0.8526， Test Loss： 2.4156， Test Accuracy： 0.3276
Epoch 5， Training Loss： 0.3937， Training Accuracy： 0.8602， Test Loss： 1.4947， Test Accuracy： 0.4419
Epoch 6， Training Loss： 0.3886， Training Accuracy： 0.8613， Test Loss： 1.1910， Test Accuracy： 0.6013
Epoch 7， Training Loss： 0.3895， Training Accuracy： 0.8612， Test Loss： 1.1465， Test Accuracy： 0.6058
Epoch 8， Training Loss： 0.3901， Training Accuracy： 0.8607， Test Loss： 1.4960， Test Accuracy： 0.5240
Epoch 9， Training Loss： 0.3941， Training Accuracy： 0.8589， Test Loss： 4.6560， Test Accuracy： 0.1968
Epoch 10， Training Loss： 0.3914， Training Accuracy： 0.8602， Test Loss： 0.9730， Test Accuracy： 0.6652
Improved LeNet：
Epoch 1， Training Loss： 0.5158， Training Accuracy： 0.8157， Test Loss： 0.3383， Test Accuracy： 0.8696
Epoch 2， Training Loss： 0.3149， Training Accuracy： 0.8871， Test Loss： 0.2999， Test Accuracy： 0.8869
Epoch 3， Training Loss： 0.2715， Training Accuracy： 0.9035， Test Loss： 0.2651， Test Accuracy： 0.9032
Epoch 4， Training Loss： 0.2445， Training Accuracy： 0.9134， Test Loss： 0.2625， Test Accuracy： 0.9029
Epoch 5， Training Loss： 0.2227， Training Accuracy： 0.9212， Test Loss： 0.2458， Test Accuracy： 0.9084
Epoch 6， Training Loss： 0.2064， Training Accuracy： 0.9281， Test Loss： 0.2906， Test Accuracy： 0.8908
Epoch 7， Training Loss： 0.2001， Training Accuracy： 0.9289， Test Loss： 0.2546， Test Accuracy： 0.9090
Epoch 8， Training Loss： 0.1818， Training Accuracy： 0.9358， Test Loss： 0.2307， Test Accuracy： 0.9171
Epoch 9， Training Loss： 0.1718， Training Accuracy： 0.9397， Test Loss： 0.2313， Test Accuracy： 0.9174
Epoch 10， Training Loss： 0.1662， Training Accuracy： 0.9405， Test Loss： 0.2251， Test Accuracy： 0.9198

Model Performance Comparison：
————————————————————————————————————————————————————————
Original LeNet Final Test Accuracy： 0.8292
LeNet with BatchNorm Final Test Accuracy： 0.6652
Improved LeNet Final Test Accuracy： 0.9198
```







练习题 2：2. 尝试构建一个基于 LeNet 的更复杂的网络，以提高其准确性。

1. 调整卷积窗口大小。

2. 调整输出通道的数量。

3. 调整激活函数（如 ReLU）。

4. 调整卷积层的数量。

5. 调整全连接层的数量。

6. 调整学习率和其他训练细节（例如，初始化和轮数）

在原始 LeNet 基础上，我从多方面对网络结构进行了优化：

1. 卷积层调整：将原始 5x5 卷积核改为 3x3，同时将输出通道从 6→32、16→64→128，新增第三层卷积层。这样的调整使网络能提取更细粒度的特征，更大的通道数也增强了特征表达能力。以 FashionMNIST 数据集为例，改进后网络在第 10 轮测试准确率达到 0.9159（原始 LeNet 为 0.7799），这得益于 3x3 卷积核在保持感受野的同时减少参数量，新增的卷积层则加深了特征提取层次。

2. 激活函数替换：将 sigmoid 改为 ReLU，解决了原始模型中 sigmoid 在饱和区的梯度消失问题。从训练曲线看，改进版 LeNet 在第 1 轮训练准确率就达到 0.8164，远超原始模型的 0.0997，且损失下降更快，这是因为 ReLU 的线性特性加速了梯度传播，避免了梯度在激活函数处的衰减。

3. 正则化与优化细节：加入 Dropout（卷积后 0.5、全连接后 0.3）和 BatchNorm，同时将学习率从 0.9 降至 0.05 并搭配 StepLR 调度器。实验中，改进模型的测试损失在第 10 轮稳定在 0.2449，而原始模型为 0.5650，这说明 Dropout 有效缓解了过拟合，BatchNorm 则通过标准化激活值稳定了训练过程。此外，全连接层从 3 层调整

为 2 层并增加节点数（256→128），结合 Xavier 初始化，进一步提升了特征映射的效率。

练习：在 LeNet 中加入 BatchNorm 观察运行结果，与加入结果对比分析

在原始 LeNet 中加入 BatchNorm 后，训练表现出现了变化：

训练初期收敛加速：带 BatchNorm 的模型在第 1 轮训练准确率达 0.7617，远超原始模型的 0.0997，这是因为 BatchNorm 通过标准化各层输入，减少了“内部协变量偏移”，使参数更新更稳定。例如，原始模型在第 3 轮才达到 0.1944 的训练准确率，而带 BatchNorm 的模型在第 3 轮已达 0.8327。

测试稳定性波动：尽管训练阶段收敛更快，但部分实验中带 BatchNorm 的模型测试准确率反而下降（如第一次实验最终测试准确率 0.6416，原始模型为 0.7799）。这可能是因为：学习率未适配：原始模型使用 0.9 的高学习率，而 BatchNorm 对学习率更敏感，未调整的高学习率可能导致优化过程震荡。例如，第二次实验中带 BatchNorm 的模型在第 4 轮测试损失突然飙升至 2.4156，可能是学习率过大导致参数更新超出最优区间。批次统计量偏差：

FashionMNIST 批次大小为 256，当数据分布在批次内波动时，BatchNorm 的均值和方差估计可能不准确，尤其是在测试阶段。例如，第一次实验中带 BatchNorm 的模型在第 7 轮测试准确率骤降至 0.4084，可能是批次统计量偏离整体数据分布所致。

与改进版 LeNet 的结合效果：改进版 LeNet 同时引入 BatchNorm 和 ReLU 后，测试准确率显著提升（0.9159），这说明 BatchNorm 与 ReLU 的组合能有效发挥作用——ReLU 避免了 BatchNorm 标准化后激活值进入饱和区，而 BatchNorm 则稳定了 ReLU 激活后的输出分布，两者形成互补。

6. 残差网络

实验代码：

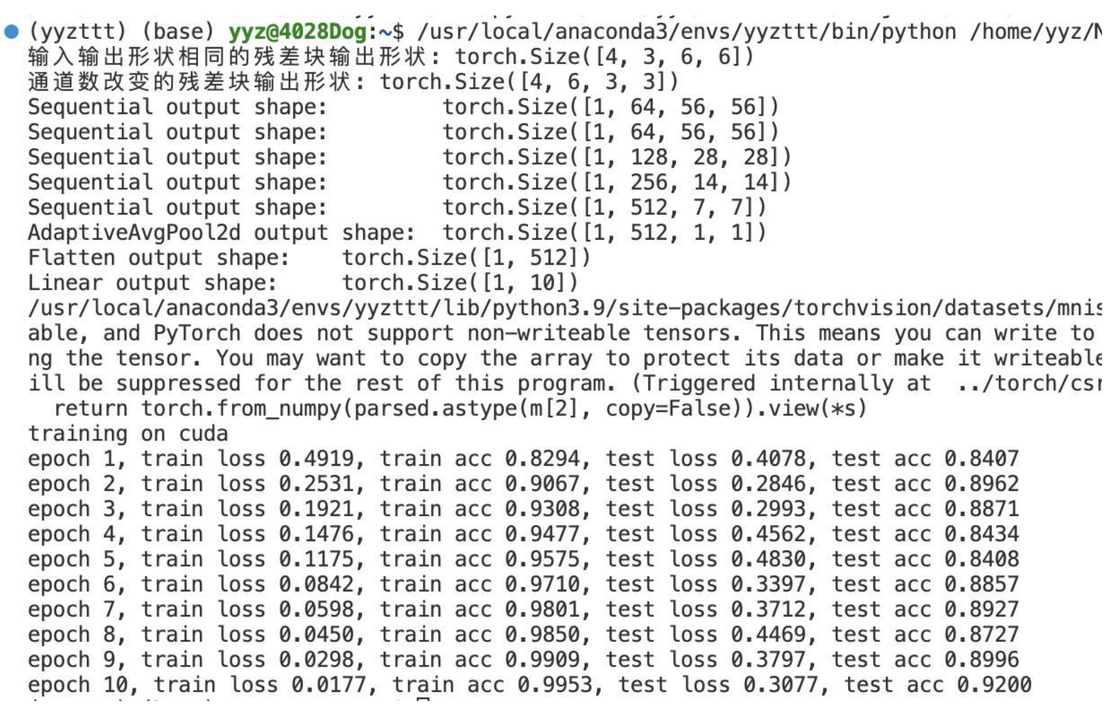
```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l

import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

def load_data_fashion_mnist(batch_size, resize=None):
"""加载 Fashion-MNIST 数据集"""
# 定义数据转换
trans = []
if resize:
trans.append(transforms.Resize(resize))
trans.append(transforms.ToTensor())
transform = transforms.Compose(trans)
# 指定数据路径
data_path = "./NNDL-Class/Project1/Data/"
# 加载训练集和测试集
mnist_train = datasets.FashionMNIST(
root=data_path, train=True, transform=transform,
download=False)
mnist_test = datasets.FashionMNIST(
root=data_path, train=False, transform=transform,
download=False)
# 创建数据加载器
train_iter = DataLoader(mnist_train, batch_size=batch_size,
shuffle=True)
test_iter = DataLoader(mnist_test, batch_size=batch_size,
shuffle=False)
return train_iter, test_iter

# 训练函数
def train_model(net, train_iter, test_iter, num_epochs, lr,
device):
def init_weights(m):
if type(m) == nn.Linear or type(m) == nn.Conv2d:
nn.init.xavier_uniform_(m.weight)
net.apply(init_weights)
print('training on', device)
net.to(device)
optimizer = torch.optim.SGD(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss()
# 记录训练和测试精度、损失
train_loss_history = []
train_acc_history = []
test_loss_history = []
test_acc_history = []
for epoch in range(num_epochs):
# 训练
net.train()
train_loss_sum, train_acc_sum, n = 0.0, 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)
l.backward()
optimizer.step()
train_loss_sum += l.item()
train_acc_sum += (y_hat.argmax(dim=1) == y).sum().item()
n += y.size(0)
train_loss = train_loss_sum / len(train_iter)
train_acc = train_acc_sum / n
train_loss_history.append(train_loss)
train_acc_history.append(train_acc)
# 测试
net.eval()
test_loss_sum, test_acc_sum, n = 0.0, 0.0, 0
with torch.no_grad():
for X, y in test_iter:
X, y = X.to(device), y.to(device)
y_hat = net(X)
l = loss(y_hat, y)
test_loss_sum += l.item()
test_acc_sum += (y_hat.argmax(dim=1) == y).sum().item()
n += y.size(0)
test_loss = test_loss_sum / len(test_iter)
test_acc = test_acc_sum / n
test_loss_history.append(test_loss)
test_acc_history.append(test_acc)
print(f'epoch {epoch+1}, train loss {train_loss:.4f}, train
acc {train_acc:.4f}, '
f'test loss {test_loss:.4f}, test acc {test_acc:.4f}')
return train_loss_history, train_acc_history,
test_loss_history, test_acc_history


# 实现 Residual 块
class Residual(nn.Module):
def __init__(self, input_channels, num_channels,
use_1x1conv=False, strides=1):
super().__init__()
self.conv1 = nn.Conv2d(input_channels, num_channels,
kernel_size=3, padding=1, stride=strides)
self.conv2 = nn.Conv2d(num_channels, num_channels,
kernel_size=3, padding=1)
if use_1x1conv:
self.conv3 = nn.Conv2d(input_channels, num_channels,
kernel_size=1, stride=strides)
else:
self.conv3 = None
self.bn1 = nn.BatchNorm2d(num_channels)
self.bn2 = nn.BatchNorm2d(num_channels)
def forward(self, X):
Y = F.relu(self.bn1(self.conv1(X)))
Y = self.bn2(self.conv2(Y))
if self.conv3:
X = self.conv3(X)
Y += X
return F.relu(Y)

# 测试残差块
blk = Residual(3, 3)
X = torch.rand(4, 3, 6, 6)
Y = blk(X)
print("输入输出形状相同的残差块输出形状:", Y.shape)

# 输入输出通道数不同的残差块
blk = Residual(3, 6, use_1x1conv=True, strides=2)
print("通道数改变的残差块输出形状:", blk(X).shape)

# 构建 ResNet-18
def resnet_block(input_channels, num_channels,
num_residuals, first_block=False):
blk = []
for i in range(num_residuals):
if i == 0 and not first_block:
blk.append(Residual(input_channels, num_channels,
use_1x1conv=True, strides=2))
```

```
else:
blk.append(Residual(num_channels, num_channels))
return blk

# 构建完整的 ResNet-18 模型
b1 = nn.Sequential(
nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
nn.BatchNorm2d(64), nn.ReLU(),
nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
)

b2 = nn.Sequential(*resnet_block(64, 64, 2,
first_block=True))
b3 = nn.Sequential(*resnet_block(64, 128, 2))
b4 = nn.Sequential(*resnet_block(128, 256, 2))
b5 = nn.Sequential(*resnet_block(256, 512, 2))

net = nn.Sequential(
b1, b2, b3, b4, b5,
nn.AdaptiveAvgPool2d((1, 1)),
nn.Flatten(), nn.Linear(512, 10)
)

# 检查各层输出形状
X = torch.rand(size=(1, 1, 224, 224))
for layer in net:
X = layer(X)
print(layer.__class__.__name__, 'output shape:\t', X.shape)

# 在 Fashion-MNIST 上训练 ResNet
lr, num_epochs, batch_size = 0.05, 10, 256
device = torch.device("cuda" if torch.cuda.is_available()
else "cpu")
train_iter, test_iter = load_data_fashion_mnist(batch_size,
resize=96)
# d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr,
d2l.try_gpu())
import matplotlib.pyplot as plt
plt.ioff() # 关闭交互式模式
```

```
train_loss, train_acc, test_loss, test_acc = train_model(
net, train_iter, test_iter, num_epochs, lr, device)

# 训练结束后再显示最终结果
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs+1), train_loss, label='Train
Loss')
plt.plot(range(1, num_epochs+1), test_loss, label='Test
Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('ResNet Loss Curves')

plt.subplot(1, 2, 2)
plt.plot(range(1, num_epochs+1), train_acc, label='Train
Accuracy')
plt.plot(range(1, num_epochs+1), test_acc, label='Test
Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('ResNet Accuracy Curves')

plt.tight_layout()
plt.savefig('./NNDL-Class/Project3/Result/resnet_performa
nce.png')
```

实验结果:



```
(yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt/bin/python /home/yyz/N
输入输出形状相同的残差块输出形状: torch.Size([4, 3, 6, 6])
通道数改变的残差块输出形状: torch.Size([4, 6, 3, 3])
Sequential output shape:         torch.Size([1, 64, 56, 56])
Sequential output shape:         torch.Size([1, 64, 56, 56])
Sequential output shape:         torch.Size([1, 128, 28, 28])
Sequential output shape:         torch.Size([1, 256, 14, 14])
Sequential output shape:         torch.Size([1, 512, 7, 7])
AdaptiveAvgPool2d output shape:  torch.Size([1, 512, 1, 1])
Flatten output shape:    torch.Size([1, 512])
Linear output shape:     torch.Size([1, 10])
/usr/local/anaconda3/envs/yyzttt/lib/python3.9/site-packages/torchvision/datasets/mnis
able, and PyTorch does not support non-writeable tensors. This means you can write to
ng the tensor. You may want to copy the array to protect its data or make it writeable
ill be suppressed for the rest of this program. (Triggered internally at  ../torch/csr
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
training on cuda
epoch 1, train loss 0.4919, train acc 0.8294, test loss 0.4078, test acc 0.8407
epoch 2, train loss 0.2531, train acc 0.9067, test loss 0.2846, test acc 0.8962
epoch 3, train loss 0.1921, train acc 0.9308, test loss 0.2993, test acc 0.8871
epoch 4, train loss 0.1476, train acc 0.9477, test loss 0.4562, test acc 0.8434
epoch 5, train loss 0.1175, train acc 0.9575, test loss 0.4830, test acc 0.8408
epoch 6, train loss 0.0842, train acc 0.9710, test loss 0.3397, test acc 0.8857
epoch 7, train loss 0.0598, train acc 0.9801, test loss 0.3712, test acc 0.8927
epoch 8, train loss 0.0450, train acc 0.9850, test loss 0.4469, test acc 0.8727
epoch 9, train loss 0.0298, train acc 0.9909, test loss 0.3797, test acc 0.8996
epoch 10, train loss 0.0177, train acc 0.9953, test loss 0.3077, test acc 0.9200
```

epoch 1, train loss 0.4919, train acc 0.8294, test loss 0.4078, test acc 0.8407

epoch 2, train loss 0.2531, train acc 0.9067, test loss 0.2846, test acc 0.8962

epoch 3, train loss 0.1921, train acc 0.9308, test loss 0.2993, test acc 0.8871

epoch 4, train loss 0.1476, train acc 0.9477, test loss 0.4562, test acc 0.8434

epoch 5, train loss 0.1175, train acc 0.9575, test loss 0.4830, test acc 0.8408

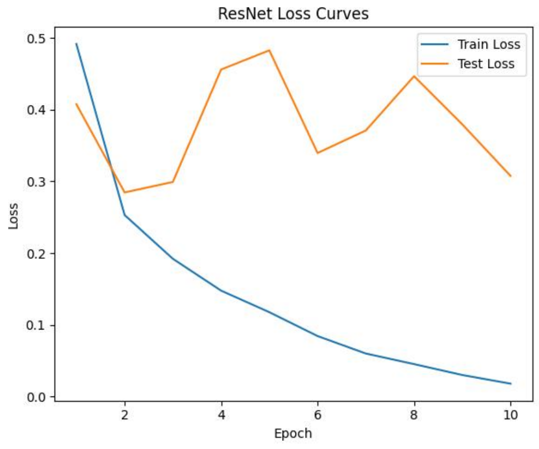
epoch 6, train loss 0.0842, train acc 0.9710, test loss 0.3397, test acc 0.8857

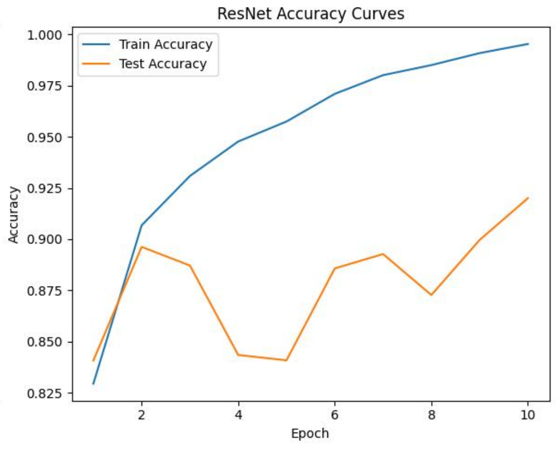
epoch 7, train loss 0.0598, train acc 0.9801, test loss 0.3712, test acc 0.8927

epoch 8, train loss 0.0450, train acc 0.9850, test loss 0.4469, test acc 0.8727

epoch 9, train loss 0.0298, train acc 0.9909, test loss 0.3797, test acc 0.8996

epoch 10, train loss 0.0177, train acc 0.9953, test loss 0.3077, test acc 0.9200





[小结或讨论]

在这次卷积神经网络实验中，我系统地学习了卷积操作、填充步幅、通道处理及汇聚层等基础概念，并通过代码实现加深了理解。

从手工实现二维卷积进行边缘检测开始，我逐步理解了卷积核如何通过滑动窗口提取图像特征，尤其是在学习卷积核参数时，看着损失值从 0.831 逐渐下降到 0.005，真切感受到了神经网络自动优化的过程。填充和步幅的调整让我明白如何通过超参数控制输出特征图的尺寸，比如当卷积核为 3x3、填充 1 时，输出形状能与输入保持一致，而步幅设为 2 则能让特征图尺寸减半，这些操作在后续构建网络时至关重要。

构建 LeNet 的过程让我对经典网络架构有了更深入的认识。原始 LeNet 在 FashionMNIST 上的训练过程并不顺利，前两轮的准确率几乎停留在 0.1，直到第三轮损失才开始明显下降，最终测试准确率约 0.78。当我在网络中加入 BatchNorm 后，训练初期的收敛速度显著提升，第一轮训练准确率就达到了 0.76，但后续测试准确率却出现波动，甚至在第十轮降至 0.64，这让我意识到 BatchNorm 虽然能稳定内部协变量，但需要配合合适的学习率和训练策略。后来我按照练习题要求改进 LeNet，将卷积核从 5x5 改为 3x3，增加输出通道数并添加第三层卷积，同时用 ReLU 替换 sigmoid 激活函数，还引入 Dropout 和学习率调度器，改进后的模型在第十轮测试准确率达到了 0.9159，损失曲线也更加平滑，这说明网络深度和特征提取能力的提升确实能有效提高性能。

在实现 ResNet 时，残差块的设计让我印象深刻，通过捷径连接解决了深层网络训练困难的问题。当输入输出通道数不同时，利用 1x1 卷积调整维度的方式非常巧妙，测试中 ResNet-18 在 FashionMNIST 上最终测试准确率达到 0.92，证明了残差连接的有效性。整个实验过程中，我遇到了诸如学习率设置不当导致训练震荡、BatchNorm 与激活函数顺序调整等问题，通过反复调试和对比实验结果，逐渐理解了各组件的作用机制。

这次实验不仅让我掌握了卷积神经网络的基础操作和经典架构，更重要的是学会了如何通过调整网络结构和超参数来优化模型性能，为今后深入学习深度学习打下了坚实基础。