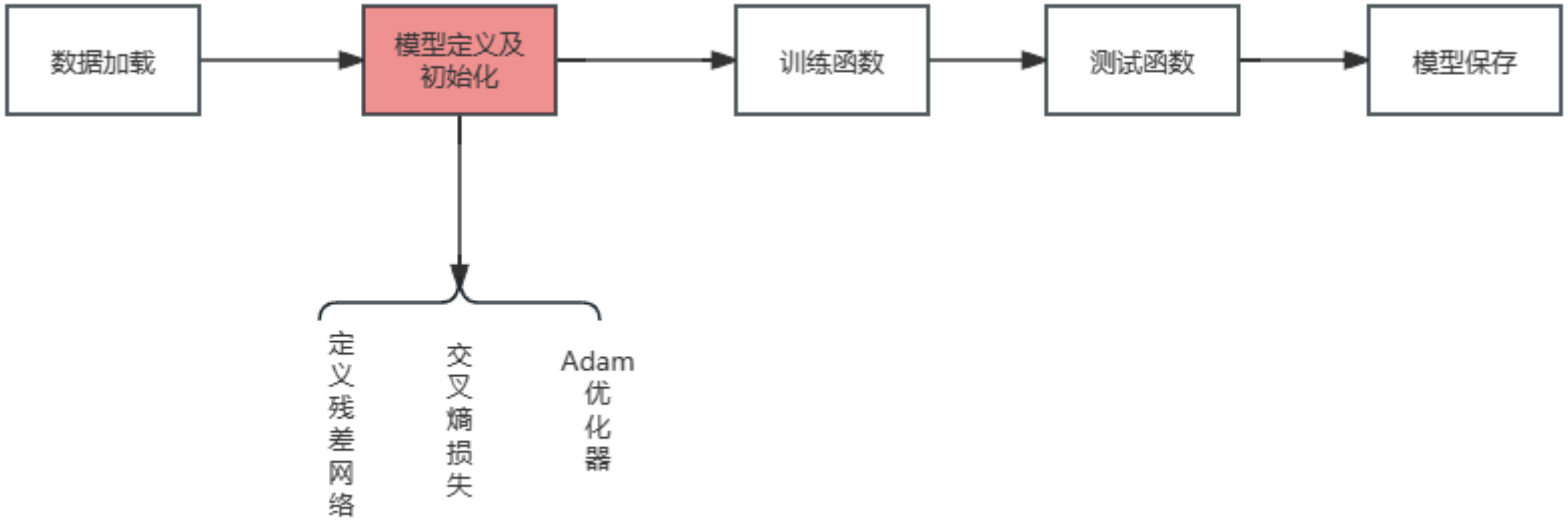
ResNet网络训练mnist手写数字识别模型报告

代码流程



1.数据加载：

- 加载MNIST数据集并创建训练和测试数据加载器。

2.模型定义及初始化：

- 定义ResNet_CNN模型，包括前向传播逻辑。
- 初始化损失函数为交叉熵损失函数。
- 初始化优化器为Adam优化器。

3.训练函数：

- 循环遍历每个训练轮次（epoch）。
- 将模型设置为训练模式。
- 遍历训练数据集，计算损失、进行反向传播和参数更新，并打印训练日志。
- 计算训练集损失和准确率。
- 调用测试函数进行模型测试。

4.测试函数：

- 将模型设置为评估模式。
- 遍历测试数据集，计算损失和预测准确率。

5.模型保存：

- 使用示例输入数据对模型进行跟踪，并保存模型。

各板块作用

main.py

main函数主要是各训练参数的初始化以及模型的训练以及测试过程

参数初始化

```
epochs = 10 #训练轮数
batch_size_train = 64   #训练集批量大小
batch_size_test = 1000  #测试集批量大小
learning_rate = 0.02    #学习率，即步长
log_interval = 10   #每隔多少个batch打印训练日志
random_seed = 1     # 随机种子
torch.manual_seed(random_seed)  #设置随机种子
```

此模型总共训练10轮，每轮每次批处理64个data；学习率设为0.02（避免过小导致陷入局部最优解）；每隔10个batch，及640个data，打印一次训练日志：包括损失值；设置随机种子有助于确保实验结果的一致性，并有助于调试和验证模型。

加载数据集

```
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data/', train=True, download=True,
                       transform=torchvision.transforms.ToTensor()),
    batch_size=batch_size_train, shuffle=True)

test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data/', train=False, download=True,
                       transform=torchvision.transforms.ToTensor()),
    batch_size=batch_size_test, shuffle=True)
```

本次训练模型采用的是mnist手写数字数据集，包含60000个训练集和10000个数据集，其为28*28的黑白图像。

初始化模型、损失函数和优化器

```
model = ResNet_CNN().to(device)      #实例化ResNet_CNN网络并移动到设备上
loss_f = nn.CrossEntropyLoss()       #交叉熵损失函数
optimizer = optim.Adam(model.parameters(), lr=learning_rate)    #Adam优化器,性能优于SGD
```

本次训练搭建了RestNet残差网络，并使用交叉熵损失函数以及Adam优化器

训练函数

```
def train(epochs):
    for epoch in range(epochs):      #循环遍历每个训练轮次
        model.train()        #将模型设置为训练模式
```

```
train_loss = 0      #跟踪损失值
correct_train = 0   #正确判断的个数
total_train = 0     #总训练个数
for batch_idx, (data, target) in enumerate(train_loader):   #遍历训练数据集，data是输入图像数据，target是对应的标签
    data, target = data.to(device), target.to(device)   #输入数据和标签移动到GPU上
    optimizer.zero_grad()   #清除之前的梯度，以避免梯度累积
    output = model(data)    #将输入数据传递给模型并得到输出
    loss = loss_f(output, target)   #计算模型输出与实际标签之间的损失
    loss.backward()     #误差反向传播计算梯度
    optimizer.step()    #根据计算的梯度更新参数
    train_loss += loss.item()       #计算训练集损失，正确率
    _, predicted = output.max(1)
    total_train += target.size(0)
    correct_train += predicted.eq(target).sum().item()

    if batch_idx % log_interval == 0:
        print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
            epoch, batch_idx * len(data), len(train_loader.dataset),
            100. * batch_idx / len(train_loader), loss.item()
        ))

train_loss /= len(train_loader.dataset)
train_accuracy = 100. * correct_train / total_train

test_loss, test_accuracy, correct_test = test()

print('\nEpoch: {} Train set: Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)'.format(
    epoch, train_loss, correct_train, total_train, train_accuracy))
print('Test set: Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
    test_loss, correct_test, len(test_loader.dataset), test_accuracy))
```

训练函数是将数据传递给ResNet_CNN模型，遍历训练数据集，在每个批次中计算损失、进行反向传播和参数更新：通过输入数据 data，得到模型的输出 output，然后计算输出与实际标签之间的损失 loss，并通过 loss.backward() 进行误差反向传播，最后根据梯度更新模型参数 optimizer.step()

测试函数

```
def test():
    model.eval()    #禁用 Dropout 和 Batch Normalization 等层的训练模式，设置模型为测试模式
    test_loss = 0   #跟踪测试过程中的损失值和准确率
    correct = 0
    with torch.no_grad():   # 上下文管理器，以避免进行梯度计算，从而加快计算速度
        for data, target in test_loader:    #遍历测试数据集，data是输入图像数据，target是对应的标签
            data, target = data.to(device), target.to(device)   #将输入数据和标签移动到GPU上
            output = model(data)    #将输入数据传递给模型并得到输出
            test_loss += loss_f(output, target).item()  #计算模型输出与实际标签之间的损失
            pred = output.argmax(dim=1, keepdim=True)   #找到每个输入数据的预测标签
            correct += pred.eq(target.view_as(pred)).sum().item()   #将正确预测的数量添加到 correct 变量中

    test_loss /= len(test_loader.dataset)   #计算测试集损失及准确率
    test_accuracy = 100. * correct / len(test_loader.dataset)
    return test_loss, test_accuracy, correct
```

测试函数用于在训练过程中对模型进行测试。在测试函数中，首先通过调用 model.eval() 将模型设置为评估模式，禁用了一些特定的层（如Dropout和Batch Normalization）的训练模式，以确保测试过程中的一致性
然后，测试数据集中的每个样本都被输入到模型中，得到模型的输出。对于每个样本，将计算模型输出与实际标签之间的损失，并统计模型正确预测的数量。最后，根据损失和正确预测的数量计算出测试集的平均损失和准确率

clss.py

clss函数主要完成了残差块的构建以及ResNet_CNN的搭建

残差模块

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        # 第一个卷积层，用于变换输入的通道数和尺寸
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        # 第一个批归一化层
        self.bn1 = nn.BatchNorm2d(out_channels)
        # ReLU 激活函数
        self.relu = nn.ReLU(inplace=True)
        # 第二个卷积层，用于恢复通道数
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        # 第二个批归一化层
        self.bn2 = nn.BatchNorm2d(out_channels)
        # 如果步长不为 1 或输入和输出的通道数不同，定义一个 downsample 层来匹配维度
        self.downsample = None
        if stride != 1 or in_channels != out_channels:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )
    #前向传播过程
    def forward(self, x):
        residual = x
        out = self.conv1(x)      #第一个卷积层的前向传播
        out = self.bn1(out)      #第一个批归一化层的前向传播
        out = self.relu(out)     #ReLU 激活函数
        out = self.conv2(out)    #第二个卷积层的前向传播
        out = self.bn2(out)      #第二个批归一化层的前向传播
        #如果定义了downsample层，进行维度匹配
        if self.downsample is not None:
            residual = self.downsample(x)
```

```
        out += residual      #将残差加到输出上
        out = self.relu(out)    #ReLU 激活函数
        return out
```

1. 残差学习原理：
   残差学习的核心思想是学习输入和输出之间的残差（即差值），而不是直接学习原始映射。这种设计可以解决深度网络训练过程中的梯度消失和梯度爆炸问题，使得网络更容易训练。

2. ResidualBlock 的结构：
   ResidualBlock 包含两个卷积层（conv1 和 conv2）、两个批归一化层（bn1 和 bn2）、一个 ReLU 激活函数以及一个 downsample 层（根据需要可能包含一个卷积层和一个批归一化层）。
   在前向传播过程中，输入 x 首先通过 conv1、bn1 和 ReLU 得到中间特征，然后经过 conv2 和 bn2 得到最终特征。
   如果输入和输出的通道数不同或者步长不为 1，会利用 downsample 层进行维度匹配。

3. ResidualBlock 的作用：
   通过残差相加的方式，可以捕获输入和输出之间的差异信息，有助于网络学习到更有效的特征表示。
   可以有效地构建深层网络，提升网络的表达能力和泛化能力。
   由于残差块的结构简单明了，易于训练和优化。

总的来说，ResidualBlock 作为深度网络中的基本模块，在深度卷积神经网络中起着至关重要的作用，能够帮助解决训练深层网络时遇到的一系列问题，同时提升网络的性能和效果。

ResNet_CNN

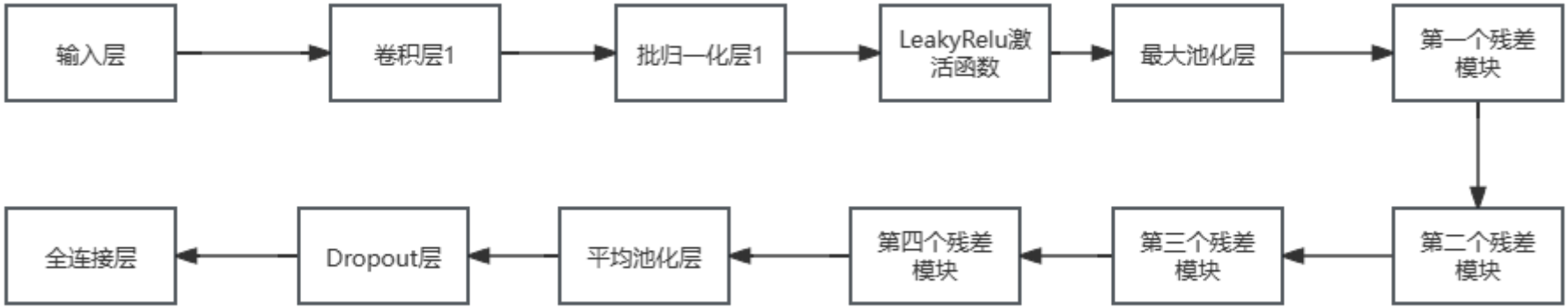
```
class ResNet_CNN(nn.Module):
    def __init__(self, num_classes=10):
        super(ResNet_CNN, self).__init__()
        self.in_channels = 64   #默认初始输入通道数为64
        #第一个卷积层
        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1, bias=False)
        #第一个批归一化层
        self.bn1 = nn.BatchNorm2d(64)
        #LeakyReLU 激活函数，有助于解决ReLU的神经元死亡问题
```

```
        self.relu = nn.LeakyReLU(negative_slope=0.01, inplace=True)
        #最大池化层
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        #构建不同层的残差块
        self.layer1 = self._make_layer(64, 3)    #构建了一个包含3个残差块的残差块组，每个残差块的输入通道数为64，输出通道数也为64
        self.layer2 = self._make_layer(128, 4, stride=2)    #构建了一个包含4个残差块的残差块组，每个残差块的输入通道数为64（由上一层的输出
        self.layer3 = self._make_layer(256, 6, stride=2)    #构建了一个包含6个残差块的残差块组，每个残差块的输入通道数为128（由上一层的输出
        self.layer4 = self._make_layer(512, 3, stride=2)    #构建了一个包含3个残差块的残差块组，每个残差块的输入通道数为256（由上一层的输出
        #全局平均池化层
        self.global_avgpool = nn.AdaptiveAvgPool2d((1, 1))
        #Dropout 层，处理过拟合
        self.dropout = nn.Dropout(0.3)
        #全连接层
        self.fc = nn.Linear(512, num_classes)

    #根据输入的输出通道数和残差块数量构建一个由多个残差块构成的层
    def _make_layer(self, out_channels, blocks, stride=1):
        layers = []
        layers.append(ResidualBlock(self.in_channels, out_channels, stride))
        self.in_channels = out_channels
        for _ in range(1, blocks):
            layers.append(ResidualBlock(out_channels, out_channels))
        return nn.Sequential(*layers)

    #前向传播过程
    def forward(self, x):
        x = self.conv1(x)  #第一个卷积层的前向传播
        x = self.bn1(x)    #第一个批归一化层的前向传播
        x = self.relu(x)   #ReLU 激活函数
        x = self.maxpool(x)  #最大池化层
        x = self.layer1(x)  #第一个残差块组
        x = self.layer2(x)  #第二个残差块组
        x = self.layer3(x)  #第三个残差块组
```

```
        x = self.layer4(x)  #第四个残差块组
        x = self.global_avgpool(x)  #全局平均池化
        x = x.view(x.size(0), -1)   #展平特征向量
        x = self.dropout(x)          #Dropout
        x = self.fc(x)               #全连接层
        return x
```

ResNet_CNN为训练模型的主要网络架构，是整个深度学习的原理框架

网络结构



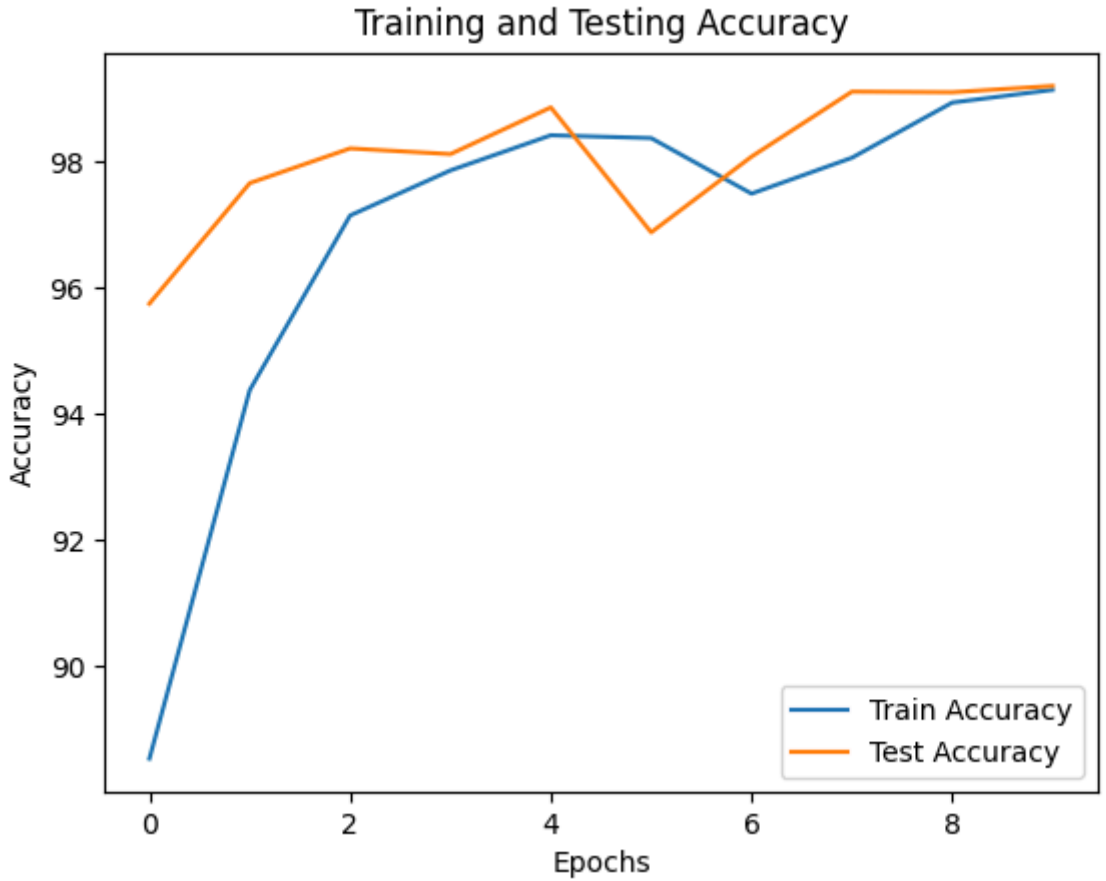
- 输入层：这里的输入是单通道的图像数据（例如灰度图像），所以输入通道数为 1。
- 第一个卷积层和批归一化层：使用了一个 3x3 的卷积核对输入图像进行特征提取，输出通道数为 64，并接入了批归一化层和 LeakyReLU 激活函数。
- 最大池化层：对特征图进行下采样，减小特征图尺寸。
- 多个残差块组成的层：包括四个残差块组，每个组内包含若干个残差块。每个残差块内部的结构是类似的，包括两个 3x3 的卷积层和批归一化层，同时第一个残差块会进行降采样（strided convolution）以匹配输入和输出的维度。这些残差块的作用是学习特征表示，并且通过跳跃连接（skip

connection）来解决梯度消失的问题。

- 全局平均池化层：对最后一个残差块组的输出特征图进行全局平均池化，将特征图大小压缩为 1x1。
- Dropout 层：用于防止过拟合，按照一定的概率丢弃部分神经元的输出。
- 全连接层：最后通过一个全连接层将特征图映射到分类数目的维度上，输出最终的分类结果。

结果

在训练集上的准确度可达到99%



最终的准确率：训练集：Accuracy: 59483/60000 (99.14%)，测试集：Accuracy: 9920/10000 (99.20%)

Loss:

Train Epoch: 9 [0/60000 (0%)] Loss: 0.016021
Train Epoch: 9 [640/60000 (1%)] Loss: 0.029153
Train Epoch: 9 [1280/60000 (2%)] Loss: 0.050220
Train Epoch: 9 [1920/60000 (3%)] Loss: 0.033309
Train Epoch: 9 [2560/60000 (4%)] Loss: 0.005529
Train Epoch: 9 [3200/60000 (5%)] Loss: 0.048378
Train Epoch: 9 [3840/60000 (6%)] Loss: 0.018423
Train Epoch: 9 [4480/60000 (7%)] Loss: 0.011928

Train Epoch: 9 [5120/60000 (9%)] Loss: 0.000334
Train Epoch: 9 [5760/60000 (10%)] Loss: 0.009953
Train Epoch: 9 [6400/60000 (11%)] Loss: 0.009144
Train Epoch: 9 [7040/60000 (12%)] Loss: 0.123446
Train Epoch: 9 [7680/60000 (13%)] Loss: 0.009328
Train Epoch: 9 [8320/60000 (14%)] Loss: 0.000589
Train Epoch: 9 [8960/60000 (15%)] Loss: 0.004237
Train Epoch: 9 [9600/60000 (16%)] Loss: 0.019387
Train Epoch: 9 [10240/60000 (17%)] Loss: 0.006603
Train Epoch: 9 [10880/60000 (18%)] Loss: 0.067053
Train Epoch: 9 [11520/60000 (19%)] Loss: 0.020456
Train Epoch: 9 [12160/60000 (20%)] Loss: 0.009906
Train Epoch: 9 [12800/60000 (21%)] Loss: 0.030066
Train Epoch: 9 [13440/60000 (22%)] Loss: 0.042518
Train Epoch: 9 [14080/60000 (23%)] Loss: 0.071778
Train Epoch: 9 [14720/60000 (25%)] Loss: 0.101711
Train Epoch: 9 [15360/60000 (26%)] Loss: 0.027915
Train Epoch: 9 [16000/60000 (27%)] Loss: 0.005118
Train Epoch: 9 [16640/60000 (28%)] Loss: 0.002895
Train Epoch: 9 [17280/60000 (29%)] Loss: 0.005298
Train Epoch: 9 [17920/60000 (30%)] Loss: 0.001113
Train Epoch: 9 [18560/60000 (31%)] Loss: 0.001437
Train Epoch: 9 [19200/60000 (32%)] Loss: 0.014853
Train Epoch: 9 [19840/60000 (33%)] Loss: 0.007448
Train Epoch: 9 [20480/60000 (34%)] Loss: 0.000999
Train Epoch: 9 [21120/60000 (35%)] Loss: 0.022029
Train Epoch: 9 [21760/60000 (36%)] Loss: 0.008698
Train Epoch: 9 [22400/60000 (37%)] Loss: 0.001441
Train Epoch: 9 [23040/60000 (38%)] Loss: 0.016153

Train Epoch: 9 [23680/60000 (39%)] Loss: 0.049975
Train Epoch: 9 [24320/60000 (41%)] Loss: 0.000421
Train Epoch: 9 [24960/60000 (42%)] Loss: 0.000638
Train Epoch: 9 [25600/60000 (43%)] Loss: 0.022134
Train Epoch: 9 [26240/60000 (44%)] Loss: 0.021366
Train Epoch: 9 [26880/60000 (45%)] Loss: 0.000758
Train Epoch: 9 [27520/60000 (46%)] Loss: 0.005418
Train Epoch: 9 [28160/60000 (47%)] Loss: 0.002853
Train Epoch: 9 [28800/60000 (48%)] Loss: 0.006523
Train Epoch: 9 [29440/60000 (49%)] Loss: 0.034167
Train Epoch: 9 [30080/60000 (50%)] Loss: 0.003085
Train Epoch: 9 [30720/60000 (51%)] Loss: 0.015242
Train Epoch: 9 [31360/60000 (52%)] Loss: 0.006528
Train Epoch: 9 [32000/60000 (53%)] Loss: 0.009880
Train Epoch: 9 [32640/60000 (54%)] Loss: 0.006986
Train Epoch: 9 [33280/60000 (55%)] Loss: 0.007908
Train Epoch: 9 [33920/60000 (57%)] Loss: 0.005435
Train Epoch: 9 [34560/60000 (58%)] Loss: 0.036836
Train Epoch: 9 [35200/60000 (59%)] Loss: 0.025095
Train Epoch: 9 [35840/60000 (60%)] Loss: 0.005947
Train Epoch: 9 [36480/60000 (61%)] Loss: 0.008368
Train Epoch: 9 [37120/60000 (62%)] Loss: 0.029567
Train Epoch: 9 [37760/60000 (63%)] Loss: 0.041588
Train Epoch: 9 [38400/60000 (64%)] Loss: 0.028201
Train Epoch: 9 [39040/60000 (65%)] Loss: 0.002879
Train Epoch: 9 [39680/60000 (66%)] Loss: 0.014525
Train Epoch: 9 [40320/60000 (67%)] Loss: 0.056336
Train Epoch: 9 [40960/60000 (68%)] Loss: 0.002734
Train Epoch: 9 [41600/60000 (69%)] Loss: 0.036621

Train Epoch: 9 [42240/60000 (70%)] Loss: 0.005161
Train Epoch: 9 [42880/60000 (71%)] Loss: 0.004798
Train Epoch: 9 [43520/60000 (72%)] Loss: 0.058016
Train Epoch: 9 [44160/60000 (74%)] Loss: 0.015572
Train Epoch: 9 [44800/60000 (75%)] Loss: 0.003977
Train Epoch: 9 [45440/60000 (76%)] Loss: 0.001879
Train Epoch: 9 [46080/60000 (77%)] Loss: 0.010654
Train Epoch: 9 [46720/60000 (78%)] Loss: 0.011698
Train Epoch: 9 [47360/60000 (79%)] Loss: 0.005480
Train Epoch: 9 [48000/60000 (80%)] Loss: 0.007211
Train Epoch: 9 [48640/60000 (81%)] Loss: 0.017509
Train Epoch: 9 [49280/60000 (82%)] Loss: 0.011700
Train Epoch: 9 [49920/60000 (83%)] Loss: 0.010549
Train Epoch: 9 [50560/60000 (84%)] Loss: 0.029972
Train Epoch: 9 [51200/60000 (85%)] Loss: 0.003858
Train Epoch: 9 [51840/60000 (86%)] Loss: 0.007529
Train Epoch: 9 [52480/60000 (87%)] Loss: 0.002951
Train Epoch: 9 [53120/60000 (88%)] Loss: 0.273703
Train Epoch: 9 [53760/60000 (90%)] Loss: 0.186130
Train Epoch: 9 [54400/60000 (91%)] Loss: 0.055897
Train Epoch: 9 [55040/60000 (92%)] Loss: 0.000448
Train Epoch: 9 [55680/60000 (93%)] Loss: 0.001074
Train Epoch: 9 [56320/60000 (94%)] Loss: 0.003084
Train Epoch: 9 [56960/60000 (95%)] Loss: 0.066254
Train Epoch: 9 [57600/60000 (96%)] Loss: 0.005516
Train Epoch: 9 [58240/60000 (97%)] Loss: 0.007536
Train Epoch: 9 [58880/60000 (98%)] Loss: 0.014476
Train Epoch: 9 [59520/60000 (99%)] Loss: 0.002425

其他尝试

最开始使用的网络架构中，为了提高训练速度，将第一个卷积层设为$7*7$，准确率比较低，只有85%左右。修改为$3*3$后但未加dropout层，模型出现了过拟合，在测试集上的准确率为95%左右，而在训练集上可达到99%。

改进

为了加速训练和提高模型性能，我在每一个残差块后加了一个批归一化操作：

```
for _ in range(1, blocks):
            layers.append(ResidualBlock(out_channels, out_channels)) # 逐层增加通道数
            layers.append(nn.BatchNorm2d(out_channels))  # 在每个残差块后添加BatchNorm2d层
        return nn.Sequential(*layers)
```

并且，我观察到先前的Loss很不稳定，有可能是学习率过大的影响。因此，我将学习率设为0.001
此时模型在最终训练集的准确率为99.59%，测试集的准确率为99.36%（最高在epoch=7出现了99.41%）

最后一个epoch的loss：

接着，我将epochs提升至30，达到了此网络的最高准确度：

```
Epoch: 29 Train set: Average loss: 0.0001, Accuracy: 59950/60000 (99.92%)
Test set: Accuracy: 9944/10000 (99.44%)

root@autodl-container-acb611a452-23167602:~/autodl-tmp#
```