**Lab2：基于卷积神经网络的MNIST手写体数字识别**

目录

[1. 神经网络架构介绍 1](#_Toc154077188)

[2. 训练结果与分析 3](#_Toc154077189)

[3. 不同架构设计对网络性能的影响 5](#_Toc154077190)

### 神经网络架构介绍

本实验设计的网络是基于经典的ResNet架构。核心组件是残差块（ResidualBlock），它包含两个卷积层，每个卷积层后接批量归一化（BatchNorm2d）和ReLU激活函数。这种设计允许网络学习残差映射，有助于缓解深层网络训练中的梯度消失问题。

# 定义残差块

*class* ResidualBlock(*nn*.*Module*):

*def* *\_\_init\_\_*(self, in\_channels, out\_channels, stride=1, downsample=*None*):

        super(ResidualBlock, *self*).*\_\_init\_\_*()

*self*.conv1 = nn.Conv2d(in\_channels, out\_channels, kernel\_size=3,

                               stride=stride, padding=1, bias=*False*)

*self*.bn1 = nn.BatchNorm2d(out\_channels)

*self*.relu = nn.ReLU(inplace=*True*)

*self*.conv2 = nn.Conv2d(out\_channels, out\_channels, kernel\_size=3,

                               padding=1, bias=*False*)

*self*.bn2 = nn.BatchNorm2d(out\_channels)

*self*.downsample = downsample

*def* forward(self, x):

        residual = x

        out = *self*.conv1(x)

        out = *self*.bn1(out)

        out = *self*.relu(out)

        out = *self*.conv2(out)

        out = *self*.bn2(out)

        if *self*.downsample:

            residual = *self*.downsample(x)

        out += residual

        out = *self*.relu(out)

        return out

在ResNet类中构建了一个初始卷积层（conv），后接批量归一化和ReLU激活函数。然后是两个残差块序列，每个序列由两个残差块组成。通过make\_layer函数实现，这样的设计使得网络能够更深入地提取特征，同时保持了计算的效率。最后，通过一个自适应平均池化层（AdaptiveAvgPool2d）和一个全连接层（fc）完成了10分类的MNIST手写体数字识别任务。

# 定义ResNet网络

*class* ResNet(*nn*.*Module*):

*def* *\_\_init\_\_*(self, block, layers, num\_classes=10):

        super(ResNet, *self*).*\_\_init\_\_*()

*self*.in\_channels = 64

*self*.conv = nn.Conv2d(1, 64, kernel\_size=7, stride=1, padding=3, bias=*False*)

*self*.bn = nn.BatchNorm2d(64)

*self*.relu = nn.ReLU(inplace=*True*)

*self*.layer1 = *self*.make\_layer(block, 64, layers[0])

*self*.layer2 = *self*.make\_layer(block, 128, layers[1], 2)

*self*.avg\_pool = nn.AdaptiveAvgPool2d((1, 1))

*self*.fc = nn.Linear(128, num\_classes)

*def* make\_layer(self, block, out\_channels, blocks, stride=1):

        downsample = *None*

        if (stride != 1) or (*self*.in\_channels != out\_channels):

            downsample = nn.Sequential(

                nn.Conv2d(*self*.in\_channels, out\_channels, kernel\_size=1,

                          stride=stride, bias=*False*),

                nn.BatchNorm2d(out\_channels))

        layers = []

        layers.append(block(*self*.in\_channels, out\_channels, stride, downsample))

*self*.in\_channels = out\_channels

        for i in *range*(1, blocks):

            layers.append(block(out\_channels, out\_channels))

        return nn.Sequential(\*layers)

*def* forward(self, x):

        out = *self*.conv(x)

        out = *self*.bn(out)

        out = *self*.relu(out)

        out = *self*.layer1(out)

        out = *self*.layer2(out)

        out = *self*.avg\_pool(out)

        out = out.view(out.size(0), -1)

        out = *self*.fc(out)

        return out

# 实例化网络

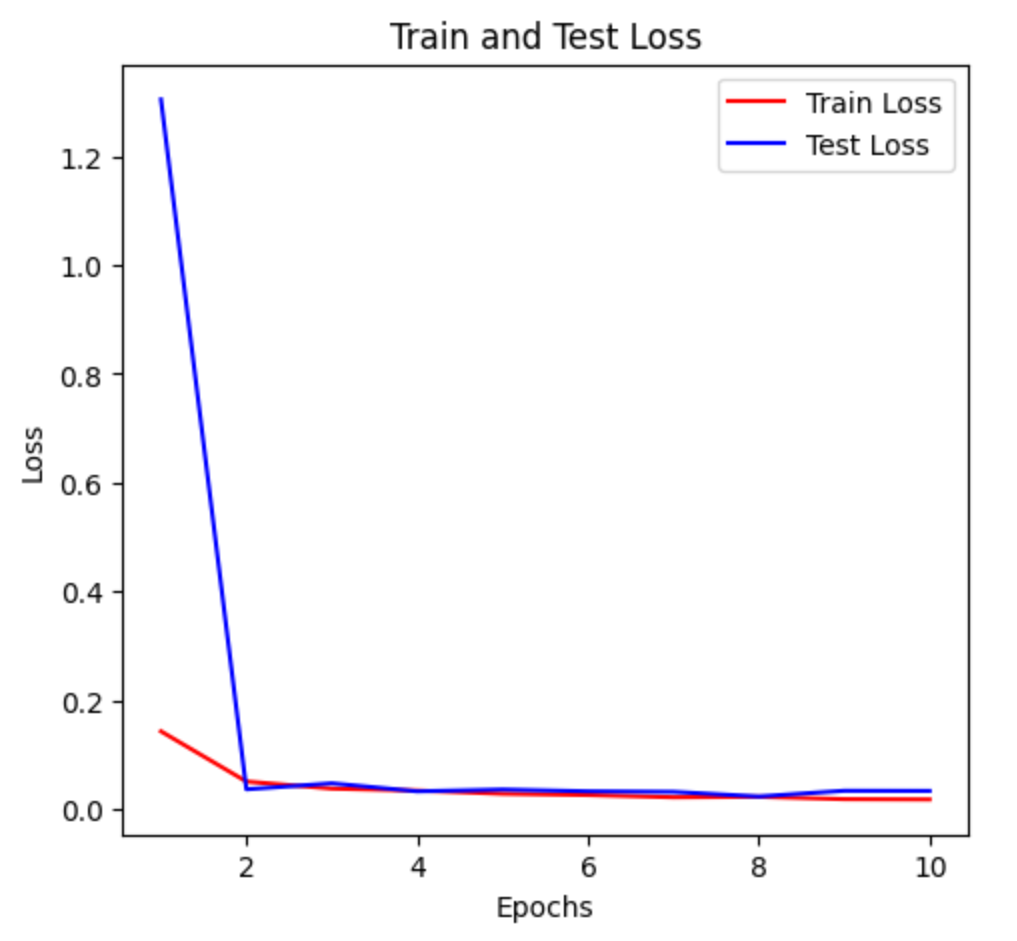
model = ResNet(ResidualBlock, [2, 2, 2, 2])

### 2. 训练结果与分析

**表：每轮 mini-batch 训练后模型在训练集和测试集上的损失和准确率**

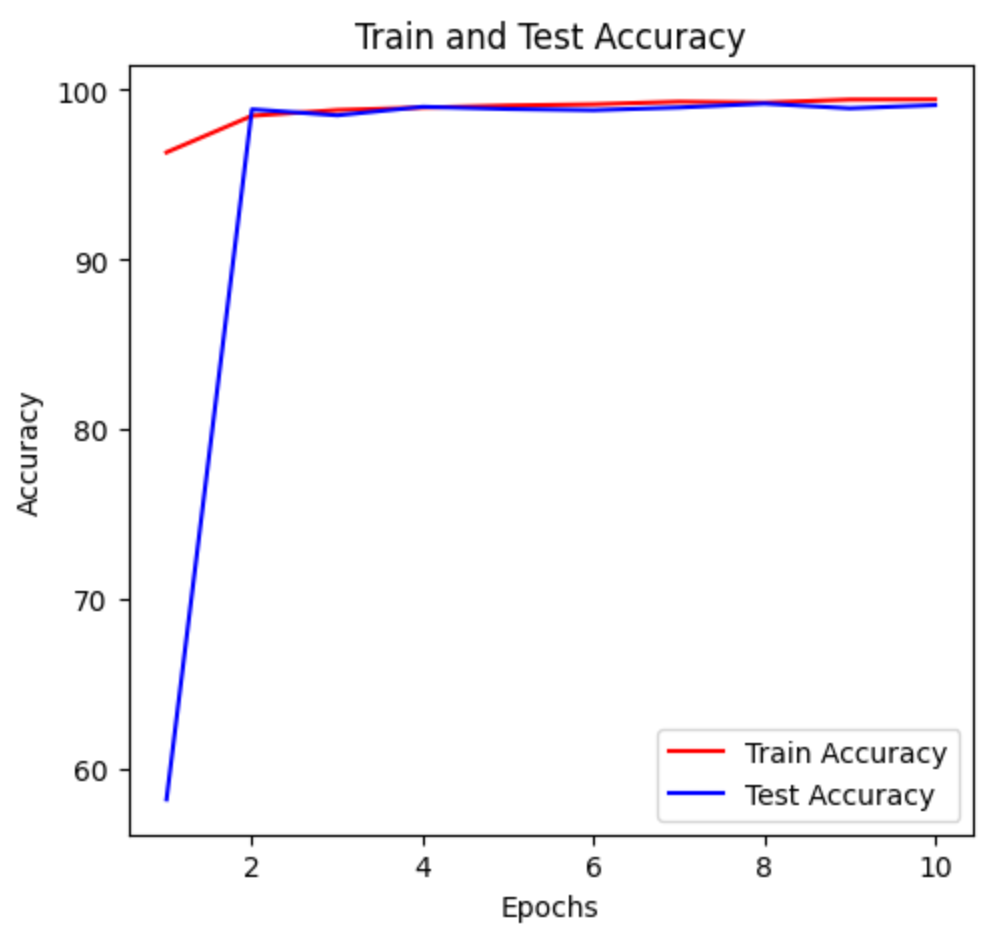
| **Epoch** | **Train Loss** | **Train Accuracy** | **Test Loss** | **Test Accuracy** |
| --- | --- | --- | --- | --- |
| 1/10 | 0.1431 | 96.33% | 1.3042 | 58.24% |
| 2/10 | 0.0508 | 98.48% | 0.0365 | 98.86% |
| 3/10 | 0.0378 | 98.82% | 0.0475 | 98.51% |
| 4/10 | 0.0341 | 98.96% | 0.0328 | 99.01% |
| 5/10 | 0.0283 | 99.09% | 0.0361 | 98.88% |
| 6/10 | 0.0260 | 99.16% | 0.0328 | 98.81% |
| 7/10 | 0.0219 | 99.31% | 0.0318 | 98.97% |
| 8/10 | 0.0227 | 99.26% | 0.0233 | 99.20% |
| 9/10 | 0.0184 | 99.44% | 0.0338 | 98.92% |
| 10/10 | 0.0179 | 99.44% | 0.0336 | 99.11% |

在训练过程中，观察到使用ReLU激活函数的网络在训练集上的损失迅速下降，并在测试集上展现出良好的泛化能力。如上表和图1所示，模型的测试损失在第一个epoch后迅速减小，并与训练损失趋于一致。这表明模型在训练数据上没有过拟合，并且具有较好的泛化能力。



**图1：使用ReLU激活函数的网络在训练过程中的损失变化**

图2展示了相应的准确率变化，可以看到训练准确率和测试准确率都很高，且随着训练的进行，两者都达到了90%以上，最终稳定在了一个较高的水平（接近100%）。



**图2：使用ReLU激活函数的网络在训练过程中的准确率变化**

最终，在训练集上的最高准确率为99.44%，在测试集上的最高准确率为99.11%。该网络基本能够有效地对MNIST手写体数字进行识别。

### 3. 不同架构设计对网络性能的影响

为了探索不同架构设计对网络性能的影响，我尝试了将激活函数由ReLU改为Sigmoid，并对训练结果进行了对比。相关修改如下：

# 定义残差块

*class* ResidualBlock(*nn*.*Module*):

*def* *\_\_init\_\_*(self, in\_channels, out\_channels, stride=1, downsample=*None*):

        super(ResidualBlock, *self*).*\_\_init\_\_*()

*...*

*self*.sigmoid = nn.Sigmoid()

*...*

*def* forward(self, x):

  ...

        out = *self*.sigmoid(out)

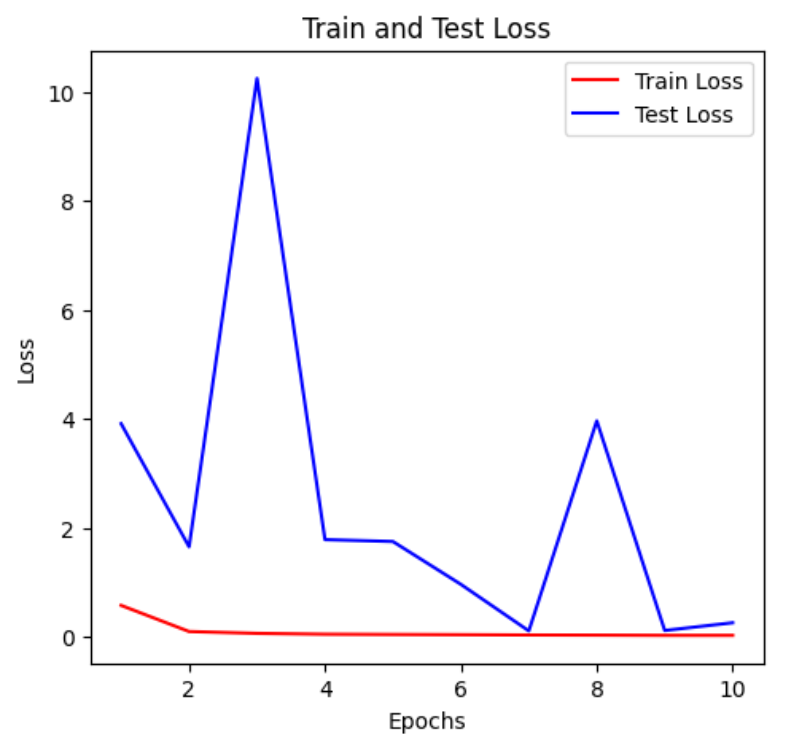
        ...

        return out

训练结果如下表和图3所示，当使用Sigmoid激活函数时，网络在训练集上的损失也呈下降趋势，但在测试集上的损失波动较大（非常大），尤其是在第3个epoch时，测试损失飙升。

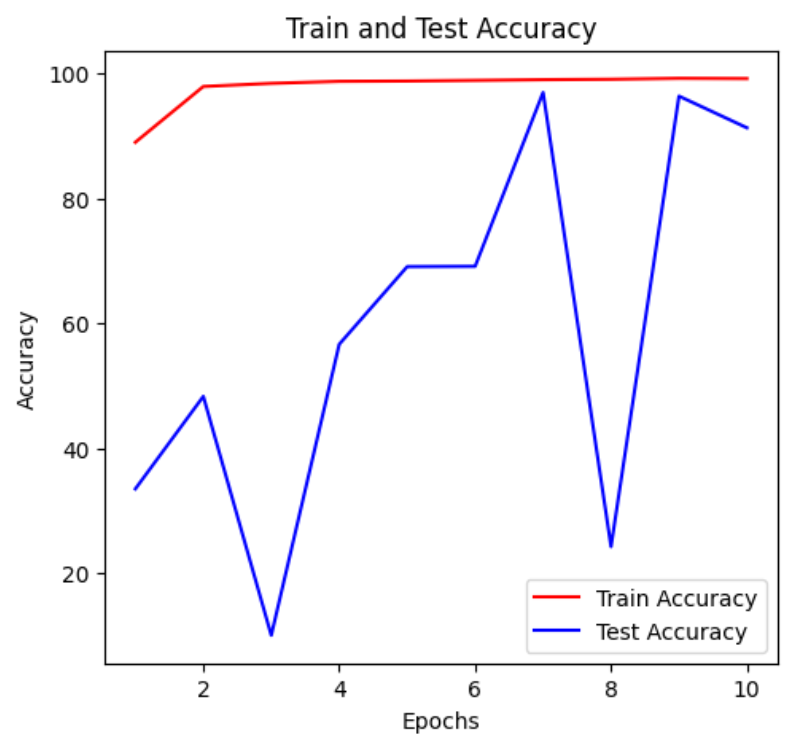
**表：每轮 mini-batch 训练后模型在训练集和测试集上的损失和准确率**

| **Epoch** | **Train Loss** | **Train Accuracy** | **Test Loss** | **Test Accuracy** |
| --- | --- | --- | --- | --- |
| 1/10 | 0.5778 | 88.96% | 3.9099 | 33.53% |
| 2/10 | 0.0986 | 97.88% | 1.6521 | 48.32% |
| 3/10 | 0.0647 | 98.38% | 10.2429 | 10.10% |
| 4/10 | 0.0494 | 98.70% | 1.7840 | 56.65% |
| 5/10 | 0.0445 | 98.75% | 1.7491 | 69.07% |
| 6/10 | 0.0402 | 98.86% | 0.9635 | 69.12% |
| 7/10 | 0.0350 | 98.97% | 0.1095 | 96.93% |
| 8/10 | 0.0322 | 99.03% | 3.9623 | 24.29% |
| 9/10 | 0.0283 | 99.18% | 0.1175 | 96.34% |
| 10/10 | 0.0283 | 99.14% | 0.2595 | 91.27% |



**图3：使用Sigmoid激活函数的网络在训练过程中的损失变化**

图4则显示了Sigmoid激活函数下的准确率变化。与ReLU激活函数相比，Sigmoid网络在测试集上的准确率明显较低，并且波动更大。这可能是由于Sigmoid函数在输出值接近0或1时梯度非常小，导致梯度消失问题。



**图4：使用Sigmoid激活函数的网络在训练过程中的准确率变化**

综上所述，ReLU激活函数由于其线性非饱和形式，在训练深层网络时能够更有效地避免梯度消失问题，从而使模型训练更加稳定，性能更优。相反，Sigmoid函数因其饱和性质，在训练中容易造成梯度消失，进而影响网络的学习速度和性能，这与本实验的实验结果一致。因此，在本实验中，ReLU作为激活函数的选择明显优于Sigmoid函数。