山东大学

崇新学堂

2025－2026 学年第 1 学期

实 验 报 告

课程名称： 信息基础Ⅱ

实验名称： 基于 LeNet-5 的 Mnist 字符识别

专 业 班 级 崇新 23

学 生 姓 名 杨瑞

实 验 时 间 2025/9/28

一、知识梳理

1. LeNet-5 简介

LeNet-5 是一种典型的非常高效的用来识别手写体数字的卷积神经网络，出自论文 Gradient-Based Learning Applied to Document Recognition，是由 Yann LeCun 提出的，对 MNIST 数据集的识别准确度可达 99.2%。

网络特点：

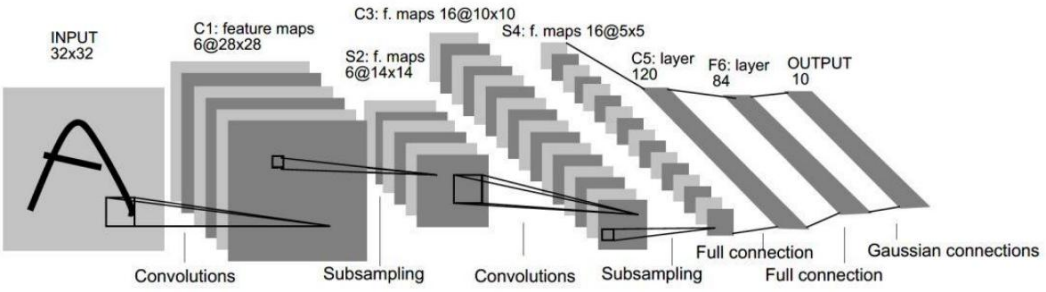
（1）局部感受野：每个卷积神经元只与输入图像的一部分区域相连，从而减少参数数量。

（2）权值共享：同一卷积核在整幅图像上滑动，提取相同类型的特征，进一步减少了参数规模。

（3）池化（下采样）：通过平均池化降低特征图的分辨率，保留主要特征同时减少计算量。

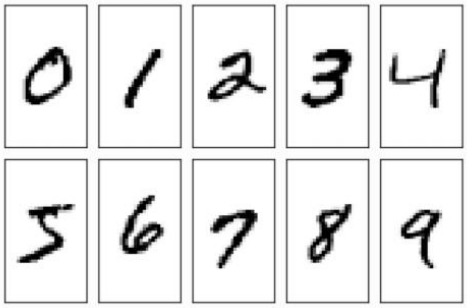
2. LeNet-5 经典结构（原始版本）

- 输入层（Input）：32×32 的灰度图像。
- C1 卷积层：6 个 5×5 卷积核，输出 6 个 28×28 的特征图。
- S2 池化层：6 个 14×14 的特征图（2×2 平均池化）。
- C3 卷积层：16 个 5×5 卷积核，输出 16 个 10×10 的特征图。
- S4 池化层：16 个 5×5 特征图（再次下采样）。
- C5 卷积层：16 个 5×5 的卷积，得到 120 个 1×1 特征图。
- F6 全连接层：84 个神经元。
- 输出层：10 个神经元，分别对应数字 0–9。



3. MNIST 数据集简介

MNIST 数据集包含 60000 张训练样本和 10000 张测试样本，每张图片是 28×28 的灰度手写数字图像，标签为 0–9 十个类别。



二、代码构建过程

在实验中，我使用 PyTorch 实现了 LeNet 网络，主要过程包括：

1. 网络定义

使用 torch.nn.Sequential 搭建网络结构，提供了两种版本：Sigmoid 激活函数版本和 ReLU 激活函数版本。（下方代码选用 Sigmoid 激活函数）

```
net = torch.nn.Sequential(
    Reshape(),
    torch.nn.Conv2d(1, 6, kernel_size=5, padding=2),
    torch.nn.Sigmoid(),
    torch.nn.AvgPool2d(kernel_size=2, stride=2),
    torch.nn.Conv2d(6, 16, kernel_size=5),
    torch.nn.Sigmoid(),
    torch.nn.AvgPool2d(kernel_size=2, stride=2),
    torch.nn.Flatten(),
    torch.nn.Linear(16 * 5 * 5, 120),
    torch.nn.Sigmoid(),
    torch.nn.Linear(120, 84),
    torch.nn.Sigmoid(),
    torch.nn.Linear(84, 10)
)
```

2. 数据处理

使用 torchvision.datasets.MNIST 下载数据集，并采用 transforms.Normalize 对图像进行标准化。

```
batch_size = 256
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))  # MNIST 数据集的均值和标准差
])

# 加载 MNIST 数据
train_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform
)
test_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=False,
    download=True,
    transform=transform
)
train_iter = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_iter = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

3. 训练过程

损失函数：CrossEntropyLoss；优化器：SGD；权重初始化：Xavier 初始化；训练时输出训练损失、准确率以及测试准确率。

```
def train(net, train_iter, test_iter, num_epochs, lr, device):
    #初始化权重
    def init_weights(m):
        if type(m) == torch.nn.Linear or type(m) == torch.nn.Conv2d:
            torch.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) #优化器：随机梯度下降（SGD）
    loss = nn.CrossEntropyLoss()
```

```
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer, num_batches = d2l.Timer(), len(train_iter)
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3) #训练损失、正确预测数和总样本数
        net.train()
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            optimizer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            #更新模型参数
            optimizer.step()
            # 计算当前 batch 的损失和准确率
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
            timer.stop()
            train_l = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            #绘图
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (train_l, train_acc, None))
        test_acc = evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    # 训练损失值、训练准确率、测试准确率
    print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
    # 每秒处理的样本数量
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(device)}')
```

三、实验结果

1. Sigmoid 激活函数版本

- 学习率设置为 lr = 0.9，训练 10 个 epoch。
- 结果：训练较为稳定，训练速度为 22742.3examples/sec 测试集准确率可以达到约 98%。

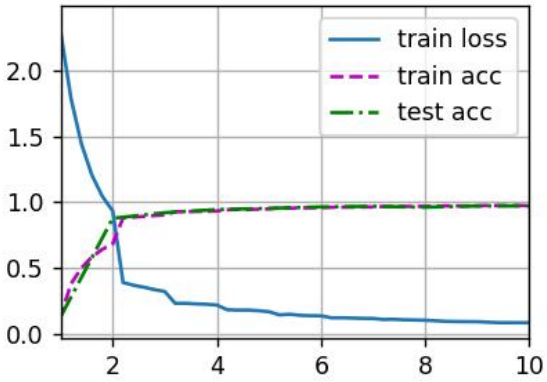


图 1 训练中 Loss 与准确率变化图像（Sigmoid）

```
loss 0.076, train acc 0.977, test acc 0.978
22742.3 examples/sec on cuda:0
```

图 2 训练结果输出（Sigmoid）

2. ReLU 激活函数版本

- 初始学习率设置于 Sigmoid 相同时（lr=0.9），模型效果很差，准确率明显下降。

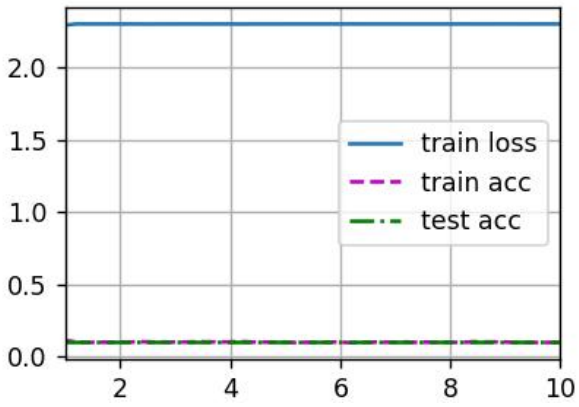


图 3 训练中 Loss 与准确率变化图像（ReLU 学习率=0.9）

```
loss 2.304, train acc 0.096, test acc 0.100
62979.1 examples/sec on cuda:0
```

图 4 训练结果输出（ReLU 学习率=0.9）

- 将学习率调整为 lr=0.01 后，收敛情况明显改善，测试集准确率达到 96% 左右，比 Sigmoid 稍低，但拟合速度更快。

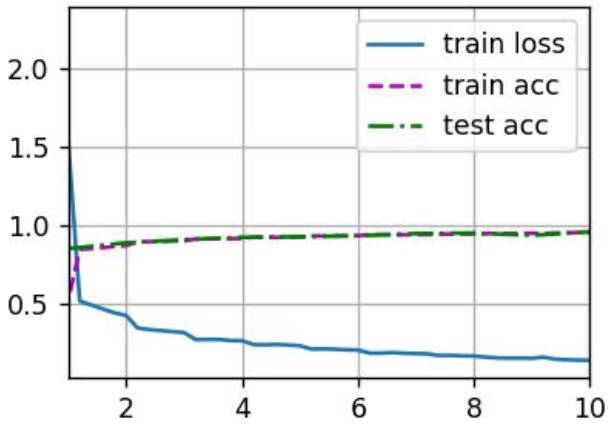


图 5 训练中 Loss 与准确率变化图像（ReLU 学习率=0.01）

```
loss 0.146, train acc 0.956, test acc 0.961
22882.0 examples/sec on cuda:0
```

图 6 训练结果输出（ReLU 学习率=0.01）

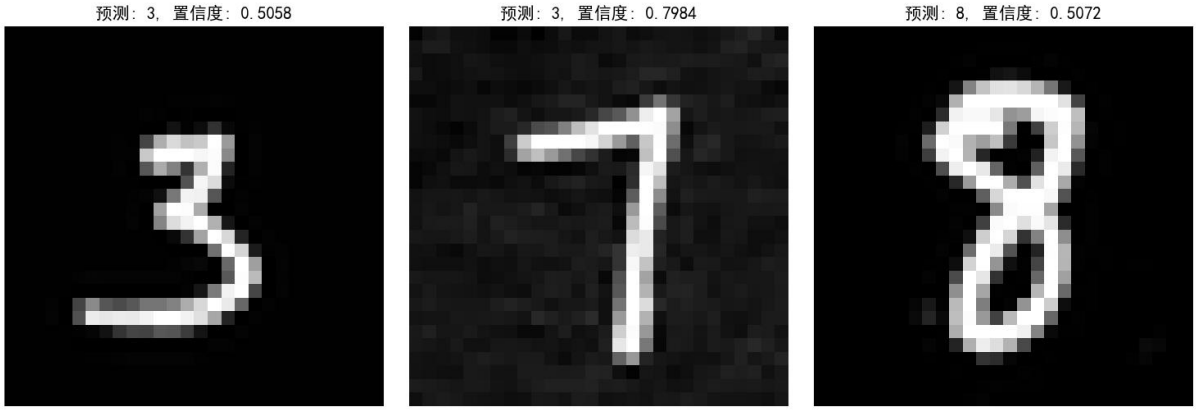
对比实验说明：

- Sigmoid 在本实验中表现稳定，但存在梯度消失问题，训练速度较慢。
- ReLU 在合适学习率下表现优异，训练速度更快，但对学习率更敏感。

四、附加实验

将训练好的模型文件 lenet5_weight.pth 保存到代码目录中，构建 test.py 文件，制作三张手写数字图片保存在代码目录的 img 文件夹中。

运行 test.py，利用模型文件对我自己制作的手写数字图片进行识别。



五、实验总结

1. 本实验实现了经典的 LeNet 网络，并在 MNIST 数据集上进行了训练和测试。

2. 通过实验对比了不同激活函数（Sigmoid vs ReLU）对模型性能的影响：

- Sigmoid 训练稳定，但存在收敛慢的问题。

- ReLU 在合适学习率下效果良好。

3. 由此可以得出结论：激活函数和学习率的选择对模型训练效果有重要影响。在现代卷积神经网络中，ReLU 通常优于 Sigmoid，但需要搭配合适的学习率和优化器。