山东大学

崇新学堂

2023－2024 学年第一学期

实 验 报 告

课程名称: 信息基础 II

专 业 班 级 崇新学堂 21 级

学 生 姓 名 刘浩

个 人 学 号 202120120312

实 验 名 称 基于 LeNet-5 的 MNIST 字符识别

实验二：基于 LeNet-5 的 MNIST 字符识别

一、 实验要求

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。要求自己编程实现网络结构，采用层次化的映射模式，实现 LeNet-5 的手写数字识别。

二、 实验原理

首先为了方便阅读我的文件，下面是文件说明：

文件名	文件说明
LeNet5.py	主文件(模型训练和保存)
test.py	从测试集中选择数字测试以及自己手写数字的测试
Train.txt	训练过程输出结果
models	模型存储位置（Mymodel 是 LeNet5 保存的模型 ReLU 是修改激活函数为 ReLU 后保存的模型）
MyNumber	自己手写的 0-9 数字
MyNumber_pred	自己手写数字的预测结果

有了上次实验的基础，对神经网络的原理有了初步的了解，而本次的实验是一个比较经典的卷积神经网络，大家都说 MNIST 数据集是深度学习入门的数据集，LeNet5 共分为 7 层，在老师提供的讲义中已经讲解的非常详细了：

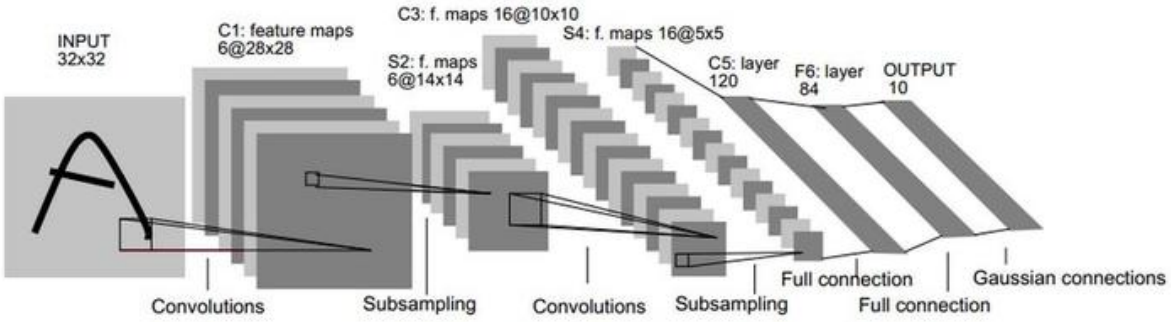


图 1 LeNet 结构

搭建神经网络的过程，其实思路是很简单的，只要按照讲义上每一层的结构，依次往下进行即可，重要的是其中的一些**参数**。

首先在图中显示的 Input 是 32×32 的，但是查阅资料[1]可知，实际上它的大小是 28×28 的，那么对于卷积层 C1，我们就需要增加一个 $padding$ 参数了，根据 $28-5+2*padding+1=28$，得到 $padding=2$。

对于其他的参数，我们就按照图 1 中所示的结构进行搭建即可。

反向传播过程依托 pytorch 可以进行自动梯度计算，从而可以比较简单的完成。

三、 实验步骤

首先数据集的导入和下载，采用 pytorch 内置的 MNIST 数据集，我们只需要调用即可，我选择的 batch_size 大小是 64，即每次处理 64 张图片

同时，我对数据集进行了标准化，更好的提高了训练的精度：

```
# 制作数据集
def data():
    transform = transforms.Compose([
        transforms.ToTensor(),
```

```
        transforms.Normalize(0.1307, 0.3081)
    ])

    # 下载训练集与测试集
    train_Data = datasets.MNIST(
        root='./dataset',
        train=True,
        download=True,
        transform=transform
    )
    test_Data = datasets.MNIST(
        root='./dataset',
        train=False,
        download=True,
        transform=transform
    )
    return train_Data, test_Data

train_Data, test_Data = data()
# 批次加载器
train_loader = DataLoader(train_Data, shuffle=True, batch_size=64)
test_loader = DataLoader(test_Data, shuffle=False, batch_size=64)
```

下面就是网络搭建的过程了，按照每一层的定义，以及维度的变换，我们只需要保证，各个函数输入输出的参数正确即可：

值得注意的是 c5 层之后我们要将数据进行展平操作后才可以进行全连接层的操作：

```
class LeNet5(nn.Module):
    def __init__(self):
        # 搭建神经网络
        super(LeNet5, self).__init__()
        self.c1 = nn.Conv2d(1,6,kernel_size=5,padding=2)
        self.Sigmoid = nn.Sigmoid()
        self.s2 = nn.AvgPool2d(kernel_size=2,stride=2)
        self.c3 = nn.Conv2d(6,16,kernel_size=5)
        self.s4 = nn.AvgPool2d(kernel_size=2,stride=2)
        self.c5 = nn.Conv2d(16,120,kernel_size=5)
        self.flatten = nn.Flatten()
        self.f6 = nn.Linear(120,84)
        self.out = nn.Linear(84,10)

    def forward(self,x):
        x = self.c1(x)
        x = self.Sigmoid(self.s2(x))
        x = self.c3(x)
        x = self.Sigmoid(self.s4(x))
        x = self.c5(x)
        x = self.flatten(x)
        x = self.Sigmoid(self.f6(x))
        x = self.out(x)
        return(x)
```

实例化我们网络，以及定义损失函数为交叉熵函数，优化器为 SGD，采用随机梯度下降的方式进行反向传播。

```
model = LeNet5().to(device)
# 损失函数
lossfunc = nn.CrossEntropyLoss()
# 优化器
optimizer = torch.optim.SGD(model.parameters(),lr=0.01,momentum=0.9)
```

下面就是训练过程了，有了 pytorch 的加持，需要写的代码其实并不是很多，在

pytorch 下反向传播算法也变成了自动进行，我们只需要调用即可，在这里我对于每一次提取到的数据，首先都转到 **GPU** 上，得到每个批次的 Loss 后，进行反向传播调整参数，同时我进行了测试集正确率的验证，以**测试集的精度**为标准，**保存最大测试集正确率**(思路：初始化一个正确率，每次计算得到的精度若比初始化大，保存模型并更新正确率)的模型！实现我的训练过程，并且在训练结束后可视化 Loss 收敛过程，输出最优的精度：

```
model = LeNet5().to(device)
# 损失函数
lossfunc = nn.CrossEntropyLoss()
# 优化器
optimizer = torch.optim.SGD(model.paradef train(model, lossfunc, optimizer, train_dataloa
der, test_dataloader):
    train_loss = 0.0
    train_acc = 0.0
    test_acc = 0.0
    count = 0
    start = time.time()

    for _, (X, y) in enumerate(train_dataloader):
        # 每次提取到 X, y 都先转到 GPU
        (X, y)  = X.to(device), y.to(device)

        # 计算当前批次的输出
        y_hat = model(X)

        # 计算当前批次的损失值
        loss_batch = lossfunc(y_hat, y)

        # 计算预测值
        _, y_pred = torch.max(y_hat, axis=1)

        # 计算当前批次的准确率
        acc_batch = torch.sum(y_pred == y) / y_hat.shape[0]

        optimizer.zero_grad()
        loss_batch.backward()
        optimizer.step()

        train_loss += loss_batch.item()
        train_acc += acc_batch.item()
        count += 1

    end = time.time()
    train_loss = train_loss / count
    train_acc = train_acc * 100 / count
    print("训练误差：", train_loss)
    print("训练精度：", train_acc)
    print(f'训练时间：{end - start}秒')

    # 计算在测试集上的精度
    model.eval()
    total_samples = 0
    correct_predictions = 0

    with torch.no_grad():
        for _, (X, y) in enumerate(test_dataloader):
```

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

            y_hat = model(X)
            _, y_pred = torch.max(y_hat, axis=1)

            correct_predictions += torch.sum(y_pred == y)
            total_samples += y.shape[0]

    test_accuracy = correct_predictions.item() / total_samples * 100
    print("测试精度：", test_accuracy)

    return train_loss , test_accuracymeters(),lr=0.01,momentum=0.9)
```

同时我在另外一个 py 文件中完成了一个**验证模型性能**的测试函数，导入我们刚刚训练好的模型，在**测试集**中选择数据，使用模型得出预测结果，和真实的结果进行对比，进而可以直观的看出模型预测的准确性，也看到了 MINIST 数据集中图片真正的样子，还有我自己手写的数字我也在代码中实现了测试！

```
# 加载已经训练好的模型
model = LeNet5()
model.load_state_dict(torch.load("models/Mymodel.pth"))
model.to(device)
model.eval()

def test(model, test_num, test_loader):
    count = 0  # 记录已处理的样本数量

    for i, (X, y) in enumerate(test_loader):
        if count >= test_num:
            break

        batch_size = X.size(0)  # 当前批次中的样本数量
        for j in range(batch_size):
            img_true, label = X[j][0].numpy(), y[j].item()
            X_batch = Variable(X.to(device))

            with torch.no_grad():
                pred = model(X_batch)
                y_pred = torch.argmax(pred[j]).item()
            print("-----------验证模型性能开始------------")
            print("预测结果：", y_pred)
            print("真实标签：", label)
            plt.imshow(img_true, cmap='gray')
            plt.show()

            count += 1
            if count >= test_num:
                break

# 对自己手写的数字处理的部分
output_folder = 'MyNumber_pred'
os.makedirs(output_folder, exist_ok=True)

for filename in os.listdir(path):
    # 构建文件的完整路径
    file_path = os.path.join(path, filename)

    # 打开图像并进行处理
    image = Image.open(file_path)
    image = image.resize((28, 28))
    image = image.convert('L')
    image = ImageOps.invert(image)  # 黑白反转
```

```
    processed_image = transform(image)
    input_tensor = torch.unsqueeze(processed_image, dim=0)

    # 利用模型进行预测
    with torch.no_grad():
        input_tensor = input_tensor.to(device)
        outputs = model(input_tensor)
        _, predicted = torch.max(outputs.data, 1)
        prediction = predicted.item()

    # 展示图像和预测结果
    plt.imshow(image, cmap='gray')
    plt.title(f"Prediction: {prediction}")
    plt.savefig(os.path.join(output_folder, f"{filename[0]}预测结果.png"))
    plt.close()
```

四、 实验结果

训练 120 轮 Loss:

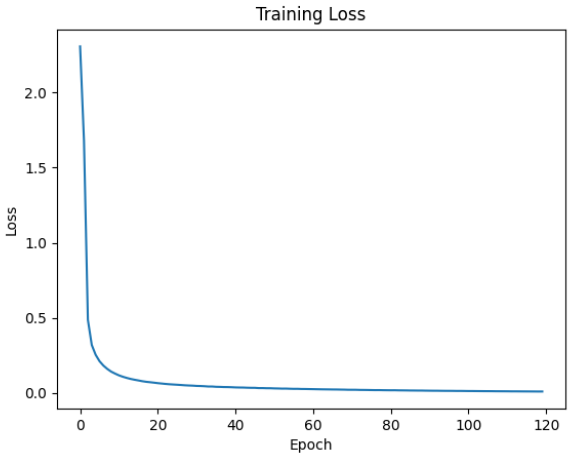
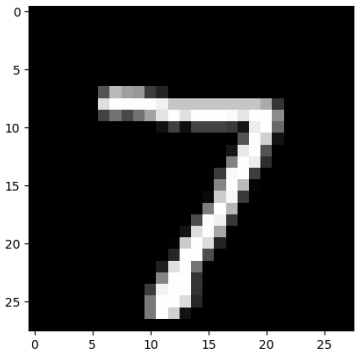


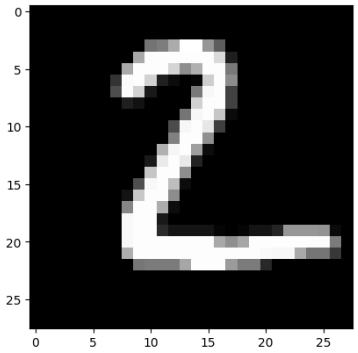
图 2 训练 120 轮 Loss 图

最佳模型的准确率高达 **98.83%**，这是在第 117 轮训练得到的；

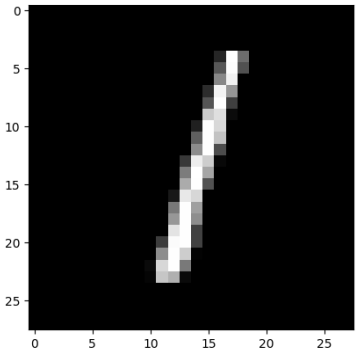
在我写的测试模型精度的函数上进行测试(即从**测试集中选图片真正进行预测**), 结果也是相当不错的：



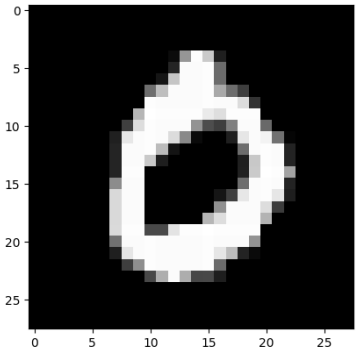

```
----------验证模型性能开始----------
预测结果： 7
真实标签： 7
```



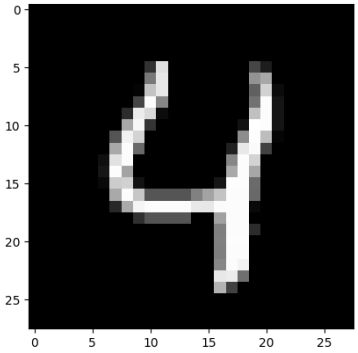
```
----------验证模型性能开始----------
预测结果： 2
真实标签： 2
```



```
----------验证模型性能开始----------
预测结果： 1
真实标签： 1
```



```
----------验证模型性能开始----------
预测结果： 0
真实标签： 0
```



```
----------验证模型性能开始----------
预测结果： 4
真实标签： 4
```

五、 实验探究

5.1 动态调整学习率策略

这里我尝试了，学习率衰减策略：

一种是即随着训练的进行，逐渐降低学习率，每隔多少个 epoch 对学习率进行衰减。

在这里我直接采取调用 torch 库中的 lr_scheduler 函数完成：

```
# 模型优化--动态调整学习率
from torch.optim import lr_scheduler

# 每隔 20 步学习率衰减为原来的 0.5
lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.5)
```

在同样采取 120 轮训练的情况下，其结果如下:

得到的最高准确率为 **98.68%**，但是收敛速度稍稍有所提高，在第 52 轮训练的时候，测试集的准确率就已经达到了 **98.66%**，相比未调整学习率的网络，其在 52 轮的时候准确率为 **98.52%**

	最高准确率	52 轮准确率
原始 LeNet5	98.83%	98.52%
调整学习率的 LeNet5	98.68%	**98.66%**

但是这种调整方式到后边并没有让我的网络收敛到最优值，我觉得可能是因为学习率到后面降的太低了。

5.2 调整网络结构策略

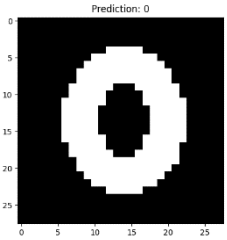
修改激活函数为 Relu，同样训练 120 轮的基础上，得到的结果如下：

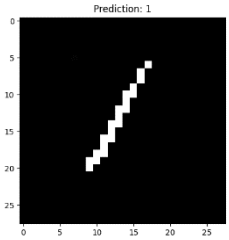
	最高准确率
原始 LeNet5	98.83%
修改 Relu 的 LeNet5	**99.11%**

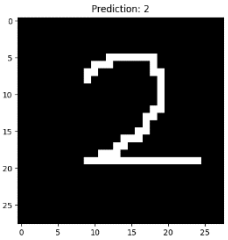
测试集精确率大大提高，达到了 99%，同时收敛速度加快！

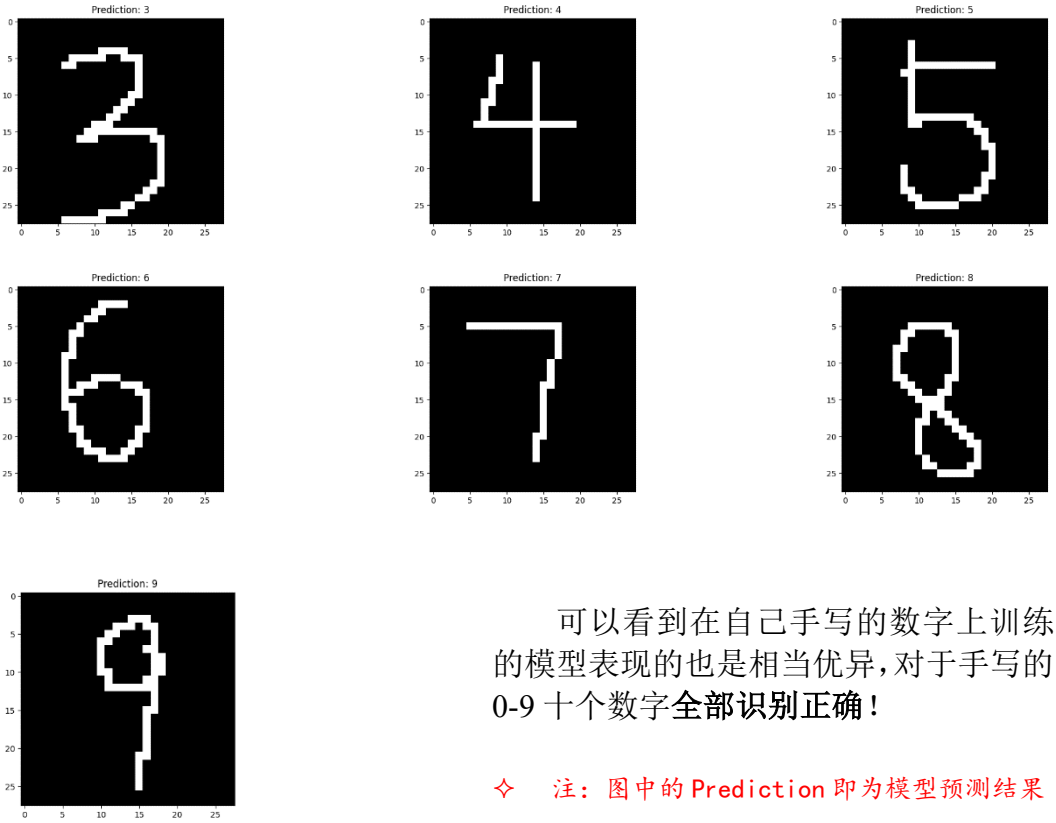
5.3 自己手写数字的识别

我自己利用电脑的画图软件**(这个可以指定 28*28 的像素，避免写的过大再进行形状调整的时候过糊)**手写了 9 个数字→见文件夹 **MyNumber**，想真实的体验一下我的网络到底有没有效果，所以我进行了该操作，将训练好的模型进行导入和识别，识别结果我导出在 MyNumber_pred 文件夹了，其结果如下：









可以看到在自己手写的数字上训练的模型表现的也是相当优异，对于手写的 0-9 十个数字**全部识别正确**！

✧ 注：图中的 Prediction 即为模型预测结果

六、 实验感想

在这次实验中基于 pytorch 框架，我实现了真正意义上的一个神经网络，在第一次实验中是基于 BP 神经网络的原理**手动搭建**的神经网络，而本次实验是基于 pytorch 搭建神经网络，大部分都是模块化的东西，整个流程实际上是比较固定的，网上开源的代码和讲解的视频也比较多，在这里我也是参考了不少资料，理解了 LeNet5 的原理以及应用，同时我看到网上有一些优化网络的方法，在本实验中我也进行了部分尝试，虽然有些优化效果不佳，但是我认为探索一下也是很有用的！

在这个实验过程有部分令人比较困惑的地方是，对于 MINIST 数据集，pytorch 已经将其内置，而且下载的数据集和我们想象的不一样，不是真正的将 6W 张图片放在文件夹中让我们直观的观看，所以我在 test 中选取了测试集中的图片，真正的看到了图片和预测结果以及真实结果对比，然后将我自己手写的数字作为测试进行输入，最终得到了一个比较直观的结果！

不过在这个过程中也引起了我的思考，在真正的神经网络训练的过程中，数据集是随着问题的改变而改变的，不可能都是 pytorch 内置好的数据集，所以该如何构建我们的数据集解决实际的问题呢？在 b 站上看到了一些视频，学到了如何构建自己的数据集进行分类，也切身感受到了神经网络是真正可以解决问题的，是可以很有效果的解决问题的一个好方法！

七、 参考资料

[1]. MNIST 数据集_保持理智 802 的博客-CSDN 博客
[2]. https://blog.csdn.net/m0_55196097/article/details/126921824
[3]. https://www.bilibili.com/video/BV1vU4y1A7QJ?t=104.2&p=4