基于 MINIST 数字手写体识别

Rongqi Zhu

2025 年 5 月 27 日

1 引言

本文基于 PyTorch 框架构建了一个手写数字识别系统，通过卷积神经网络实现了 MNIST 数据集的自动化分类任务。系统采用模块化设计，完整覆盖数据预处理、模型训练、性能验证及实际应用场景部署的全流程。

在数据处理层面，系统通过 `transforms.Compose` 构建标准化预处理管道，对输入图像执行归一化操作，有效提升模型收敛速度。结合 `DataLoader` 实现批量化数据加载，利用随机打乱机制增强训练过程的鲁棒性。网络架构设计方面，采用轻量级卷积神经网络结构，首层卷积核，后接最大池化层提取初级特征，第二层卷积，进一步抽象高阶特征，最终通过全连接层完成分类决策，其间使用 ReLU 激活函数与 CrossEntropyLoss 函数构建非线性表达能力。

模型训练过程中采用 Adam 优化器与交叉熵损失函数，训练阶段每 3000 个批次输出损失值以监控学习过程。验证集评估表明，该系统在 MNIST 测试集上实现了超过 99% 的分类准确率。为增强实用价值，系统提供模型参数持久化功能（保存为`.pth` 格式），支持外部自定义图像的端到端推理：输入图像经自适应尺寸调整、灰度转换及标准化后，通过反归一化处理，实现可视化，同时输出预测类别及其置信度概率。该实现方案不仅验证了卷积神经网络在手写数字识别任务中的有效性，更为相关图像分类问题提供了可扩展的技术框架。

2 项目总体框架

本项目实现了手写数字识别的完整流程，主要包含数据处理、模型构建、训练优化和应用部署四个核心部分。

数据处理

从 MNIST 数据集获取手写数字图像，将原始像素值 (0-255) 转换为标准化浮点数值。使用 PyTorch 的 DataLoader 进行批量读取，训练时随机打乱数据顺序以提升模型泛化能力。

模型结构

网络包含两个卷积层和两个全连接层：第一卷积层提取边缘等基础特征，第二卷积层组合复杂特征，全连接层最终完成数字分类。所有隐藏层使用 ReLU 激活函数，输出层通过 Softmax 计算各数字概率。

训练过程

采用 Adam 优化器自动调整学习率，以交叉熵作为损失函数。每次输入 128 张图像进行前向计算，根据预测误差反向更新网络参数。训练过程中定期输出损失值以监控收敛情况。

实际应用

训练完成的模型可保存为.pth 文件。部署时支持加载外部手写图片：自动调整图像尺寸后，通过相同预处理流程输入模型，最终返回识别结果及其置信度，并通过图像可视化展示预测效果。系统代码采用模块化设计，主要功能封装为独立函数，便于后续扩展和维护。

3 功能详细设计

3.1 导入需要的库

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
from PIL import Image
```

torch：PyTorch 的核心库

- 张量操作（如矩阵乘法等）
- 自动微分（autograd，用于神经网络训练）
- 设备管理（CPU/GPU 切换，如.to('cuda')）

torchvision：PyTorch 的计算机视觉工具库

- **子模块**：
  - `datasets`：预加载常用数据集（如 MNIST、CIFAR-10）。
  - `transforms`：图像预处理（如裁剪、归一化、数据增强）。
  - `models`：预训练模型（如 ResNet、VGG）。

3.2 MINST 数据集下载

```
from torch.utils.data import DataLoader
train_set = datasets.MNIST("data", train=True, download=True,transform=pipeline
    ↪ )
test_set = datasets.MNIST("data", train=False, download=True, transform=
    ↪ pipeline)
```

数据预处理管道

```
pipeline = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
```

创建数据加载器

```
train_loader = DataLoader(train_set, batch_size=Batch_size, shuffle=True)
test_loader = DataLoader(test_set, batch_size=Batch_size, shuffle=True)
```

3.3 CNN 网络结构

```
class Digit(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 10, 5)
        self.conv2 = nn.Conv2d(10, 20, 3)
        self.fc1 = nn.Linear(20*10*10, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2)
        x = F.relu(self.conv2(x))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

3.3.1 卷积层组

- **第一卷积层**:
  - 卷积核尺寸：5×5
  - 输入通道：1（灰度图像）
  - 输出通道：10

- **第二卷积层**:
  - 卷积核尺寸：3×3
  - 输入通道：10
  - 输出通道：20

3.3.2 全连接层组

- **第一个全连接层**:
  - 输入维度：$20 \times 10 \times 10 = 2000$
  - 输出维度：500
- **输出层**:
  - 输入维度：500
  - 输出维度：10（对应数字 0-9 的分类）

3.3.3 前向传播流程

特征提取阶段

1. 第一卷积 → ReLU 激活 → 2×2 最大池化
2. 第二卷积 → ReLU 激活

分类决策阶段

1. 特征图展平处理
2. 两层全连接变换
3. 最终对数 softmax 归一化

维度变换过程

输入数据经历以下维度变化：

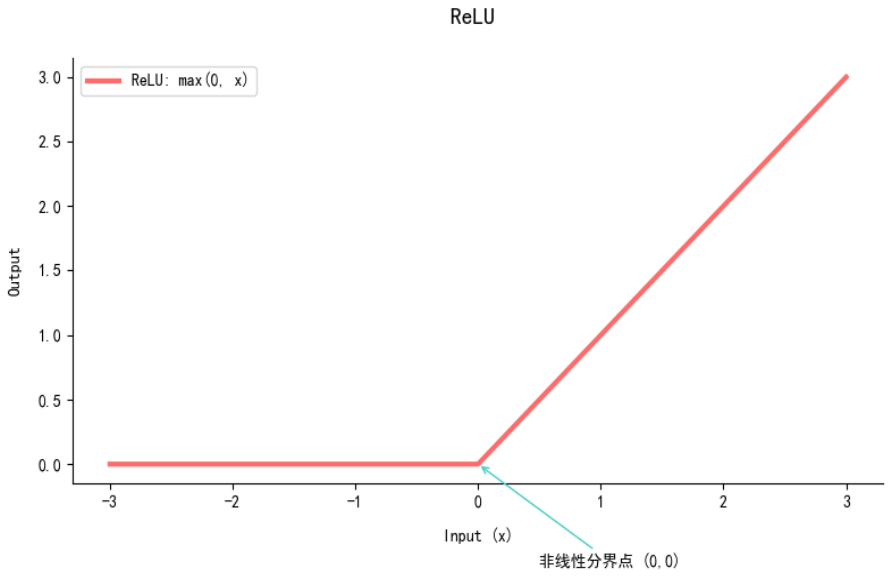
$$1 \times 28 \times 28 \xrightarrow{\text{第一卷积}} 10 \times 24 \times 24 \xrightarrow{\text{池化}} 10 \times 12 \times 12$$

$$\xrightarrow{\text{第二卷积}} 20 \times 10 \times 10 \xrightarrow{\text{展平}} 2000 \xrightarrow{\text{全连接 1}} 500 \xrightarrow{\text{全连接 2}} 10$$

3.3.4 激活函数

本项目使用的激活函数为 Relu 激活函数，在此简单介绍一下 Relu 激活函数：Relu 的全称为修正线性单元 (Rectified Linear Unit) ，其函数表达式和图像如下所示：ReLU 函数表达式：

$$\mathrm{ReLU}(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$



Relu 的优点如下：由 Relu 的原始图像和导数图像可知，Relu 可能使部分神经元的值变为 0, 降低神经网络复杂性, 从而有效缓解过拟合的问题。由于当 x>0 时，Relu 的梯度恒为 1，所以随着神经网络越来越复杂，不会导致梯度累乘后变得很大或很小，从而不会发生梯度爆炸或梯度消失问题。Relu 的计算非常简单，提高了神经网络的效率

训练过程

```
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
```

```
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 3000 == 0:
            print("第{}次训练:loss:␣{:.3f}".format(epoch, loss.item()))
```

步骤	说明
获取训练数据及标签	从数据加载器（DataLoader）中读取输入数据和对应标签
梯度清零	清除优化器中上一轮的梯度，避免梯度累积
模型预测	输入数据通过模型前向传播，得到预测值 pred
计算损失	比较预测值和真实标签，计算损失值 loss
反向传播	根据损失计算模型参数的梯度
更新参数	优化器根据梯度更新模型参数

3.4 测试过程

```
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.cross_entropy(output, target, reduction='sum').item
                ↪ ()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)
    print("测试: loss:{:.3f},正确率:{:.2f}\n".format(test_loss, 100.0 * correct
        ↪ / len(test_loader.dataset)))
```

步骤	说明
加载模型和优化器	从文件加载训练好的模型权重（和优化器状态）
设置模型为评估模式	关闭 Dropout、BatchNorm 等训练专用层的行为
禁用梯度计算	减少内存占用，加速推理
获取测试数据	从测试集读取输入数据（无需标签，除非需计算指标）
模型前向传播	输入数据通过模型，得到预测输出
计算评估指标	可选：若有标签，计算准确率、F1 等指标
后处理与输出结果	对输出进行解析（如分类取 argmax），打印或保存结果

3.5 CrossEntropyLoss

CrossEntropyLoss 是手写数字识别中的核心损失函数。它实际上是 Softmax + Log + NLLLoss 三个函数的集成，下面分别说明这三个函数：

Softmax

Softmax 回归是一个线性多分类模型，在 MINIST 手写数字识别问题中，Softmax 最终会给出预测值对于 10 个类别（0~9）出现的概率，最终模型的预测结果就是概率最大的类别。Softmax 计算公式如下：

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{1}$$

其中分子的 z_i 是多分类中的第 i 类的输出值，分母将所有类别的输出值求和，使用指数函数来将其转换为概率，最终将神经网络上一层的原始数据归一化到 $[0, 1]$，使用指数函数的原因是因为上一层的数据有正有负，所以使用指数函数将其变为大于 0 的值。具体转换过程如下图所示，可以通过判断哪类的输出概率最大，来判断最后的分类结果。

Log

经过 Softmax 后，还要将其结果取 log(对数), 目的是将乘法转化为加法，从而减少计算量，同时保证函数的单调性，因为 $\ln(x)$ 单调递增且：

$$\ln(x) + \ln(y) = \ln(xy) \tag{2}$$

NLLLoss

最终使用 NLLLoss 计算损失，损失函数定义为：

$$\text{Loss}(\hat{Y}, Y) = -Y \cdot \log(\hat{Y}) \tag{3}$$

其中的参数含义：

- $\hat{Y}$ 表示 Softmax 经过 log 后的值
- Y 为训练数据对应 target 的 One-hot 编码，表示此训练数据对应的 target。

3.6 保存训练好的模型

```
model_save_path = "./mnist_cnn.pth"
torch.save({
    'epoch': epochs,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),},model_save_path)
print(f"模型已成功保存到␣{model_save_path}")
```

保存训练好的模型，方便以后的测试，增强了实用性。

3.7 测试自己的数据

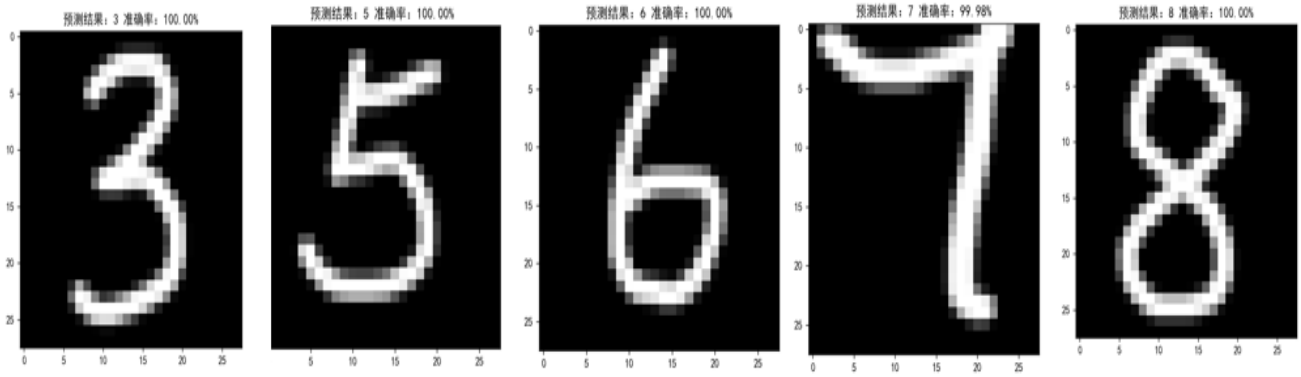
```
def test_mydata():
    checkpoint = torch.load("./mnist_cnn.pth")
    model.load_state_dict(checkpoint['model_state_dict'])
    model.eval()

    image = Image.open('mytest/img_7.png').convert('L')
    image = image.resize((28, 28))

    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))])
    image_tensor = transform(image).unsqueeze(0).to(device)

    with torch.no_grad():
```

```
        output = model(image_tensor)
    prob, pred = torch.exp(output).max(dim=1)
    print(f'预测结果：{pred.item()}␣准确率：{prob.item():.2%}')

    plt.rcParams['font.sans-serif'] = ['SimHei']
    plt.rcParams['axes.unicode_minus'] = False
    img_show = image_tensor.cpu().squeeze()
    img_show = img_show * 0.3081 + 0.1307
    plt.imshow(img_show.numpy(), cmap='gray')
    plt.title(f'预测结果：{pred.item()}␣准确率：{prob.item():.2%}')
    plt.show()
```

3.8 训练与测试结果

```
第1次训练:loss: 2.325
测试: loss:0.058,正确率:98.21

第2次训练:loss: 0.119
测试: loss:0.045,正确率:98.66

第3次训练:loss: 0.016
测试: loss:0.038,正确率:98.75

第4次训练:loss: 0.027
测试: loss:0.039,正确率:98.75

第5次训练:loss: 0.004
测试: loss:0.039,正确率:98.72
```

自定义数据测试



4 总结

通过本次手写体数字识别课题的开发，我不仅完成了手写数字识别的机器学习任务，更重要的是掌握了科研与工程实践中必备的工具，并对机器学习领域建立了初步认知，同时也让我意识到工具技能在项目开发中的重要性。

对于机器学习的初学者来说，MNIST 数据集的“友好”数据特性，大幅降低了学习门槛。该数据集样本格式统一，均为 28×28 像素的灰度图像，且附带精确标注的 0 - 9 数字类别标签，数据规模适中，包含 6 万条训练数据与 1 万条测试数据。无需花费大量精力处理复杂的数据清洗、格式转换问题，可将重心直接放在机器学习核心流程上，如数据加载、模型训练与评估。这种低门槛的学习环境，能有效减少学习初期的挫败感，增强学习信心。MNIST 数据集还能帮助学习者构建完整的机器学习项目思维。从数据预处理阶段的灰度归一化、图像填充，到模型训练过程中的损失函数选择、优化器配置，再到模型评估时使用准确率、混淆矩阵等指标分析性能，初学者可以在 MNIST 项目中完整经历机器学习项目的各个环节。使得能够深刻体会到不同算法在特征处理能力上的差异，以及算法选择对模型效果的关键作用。这种从基础算法到复杂模型的实践进阶，有助于循序渐进地掌握机器学习算法的精髓。

此外在开发过程中，我不仅对机器学习有了初步了解，还系统学习了 GitHub、Markdown、LaTeX 等实用工具，同时巩固了 Python 编程能力。这些技能相互交织，共同构成了完整的项目开发，让我对技术实践有了更立体的认知。

GitHub 的使用为项目开发带来了高效的协作与版本管理能力。从创建仓库开始，我逐步掌握了 commit 记录代码迭代、push 同步本地修改至云端的操作。每一次提交记录都成为可

追溯的开发足迹，无论是修复代码漏洞，还是尝试新的算法改进，都能通过版本管理轻松回溯与对比。这不仅保障了代码安全，更让我意识到规范的版本控制是大型项目开发的核心。

Markdown 的学习改变了我的文档撰写习惯。在项目中，无论是记录实验步骤、分析模型训练结果，还是整理算法原理，Markdown 都展现出强大的灵活性。通过简单的符号，就能快速搭建层次分明的文档结构，插入代码块、表格与数学公式，让技术内容的呈现既清晰又专业。这种轻量级的标记语言，大幅提升了文档撰写效率，同时增强了文章的可读性。

而 LaTeX 的接触，则打开了学术写作的新大门。虽然其复杂的语法在初期带来不少困扰，但当看到精心排版的公式、规范的论文样式最终呈现时，我深刻理解了学术界对它的青睐。在 MNIST 项目报告撰写中，LaTeX 的排版优势得以充分发挥，尤其是复杂的数学公式与专业图表的呈现，让报告兼具学术性与美观性，也为未来的学术研究与论文撰写积累了宝贵经验。

在编程语言方面，Python 作为项目开发的核心工具，其应用贯穿始终。通过 MNIST 项目，我进一步巩固了 Python 的基础语法，每一次代码调试、每一个功能实现，都加深了我对 Python 语言特性的理解，尤其是在处理图像数据、构建神经网络模型时，Python 的简洁高效与丰富的第三方库支持，让复杂的算法实现变得可行。

此次 MNIST 项目开发，GitHub、Markdown、LaTeX 与 Python 共同构建起技术实践的多维框架。它们不仅是完成项目的工具，更是提升工程素养与技术能力的重要阶梯。未来，我将继续深化这些工具的应用，将其融入更多的技术实践中，不断提升自己的综合开发能力。