深度学习作业——手写数字识别

Clark

1 概述

本次实验基于 PyTorch 框架构建卷积神经网络（CNNs），实现 MNIST 手写数字识别任务。通过构建多层卷积神经网络结构，结合池化层、激活函数和全连接层，实现了对手写数字图像的高精度分类。实验最终测试准确率达到 99% 以上，远超实验要求的 98% 标准。

2 实验原理

2.1 卷积神经网络基本原理

卷积神经网络由卷积层、池化层和激活函数层交替组成，构成深度网络结构。

2.1.1 卷积运算

卷积运算是 CNN 的核心操作，给定二维图像 I 作为输入，二维卷积核 K，卷积运算可表示为：

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n). \tag{1}$$

2.1.2 池化操作

池化操作使用相邻输出的统计特征作为输出，常用最大池化和均值池化，用于降低特征图尺寸并增强特征不变性。

2.1.3 激活函数

激活函数引入非线性变换，增强网络的表达能力。本实验采用 ReLU 激活函数：

$$\mathrm{ReLU}(x) = \max(0, x). \tag{2}$$

2.2 PyTorch 框架

PyTorch 是一个以 Python 优先的深度学习框架，支持动态图和强大的 GPU 加速，提供自动求导等功能。

3 实验环境与数据集

3.1 实验环境

- 开发工具: Anaconda，PyTorch
- 硬件环境: GPU 加速（如可用）
- 编程语言: Python

3.2 数据集

使用 MNIST 手写数字数据集，包含 60,000 个训练样本和 10,000 个测试样本，每个样本为 28×28 像素的灰度图像。

4 实验步骤与代码实现

4.1 数据预处理

我们使用 PyTorch 提供的 torchvision 库来加载 MNIST 数据集。首先，通过 transforms.ToTensor() 将图像数据转换为 PyTorch 张量，同时将像素值从 $[0, 255]$ 归一化到 $[0, 1]$ 区间。然后，我们分别下载训练集和测试集，并放置在./data 目录下。为了在训练过程中高效地加载数据，我们使用 DataLoader 进行批处理。对于训练集，我们设置批大小为 64，并打乱数据顺序（shuffle=True），这样可以使模型在每轮训练中看到不同顺序的数据，有利于提高泛化能力。对于测试集，我们设置批大小为 1000，并且不需要打乱顺序。

```
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import torch


# 转换为张量
transform = transforms.ToTensor()

# 下载训练集和测试集
train_data = torchvision.datasets.MNIST(root='./data', train=True, download=True,
    transform=transform)
test_data = torchvision.datasets.MNIST(root='./data', train=False, download=True,
    transform=transform)

# 构建数据加载器
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=1000, shuffle=False)
```

4.2 网络模型构建

我们构建了一个包含三个卷积层的卷积神经网络（CNNs）。每个卷积层后接一个 ReLU 激活函数和一个最大池化层。具体来说，第一卷积层使用 16 个 5×5 的卷积核，填充为 2 以保持空间分辨率，然后通过 2×2 最大池化层，将图像尺寸从 28×28 降低到 14×14。第二卷积层使用 32 个 5×5 的卷积核，同样填充 2，再经过 2×2 池化，得到 7×7 的特征图。第三卷积层使用 64 个 3×3 的卷积核，填充 1，再经过 2×2 池化，得到 3×3 的特征图。之后，我们使用 Dropout 层（丢弃率为 0.5）来防止过拟合，最后通过全连接层将特征映射到 10 个类别的输出。

在模型的前向传播过程中，我们依次通过三个卷积层，然后将特征图展平为一维向量，应用 Dropout 后，通过全连接层得到最终的输出。

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # 建立第一卷积层(Conv2d) -> 激励函数(ReLU) -> 池化(MaxPooling)
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )

        # 建立第二卷积层(Conv2d) -> 激励函数(ReLU) -> 池化(MaxPooling)
        self.conv2 = nn.Sequential(
            nn.Conv2d(
                in_channels=16,
                out_channels=32,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            nn.ReLU(),
            nn.MaxPool2d(2)
            )

        # 建立第三卷积层(Conv2d)
        self.conv3 = nn.Sequential(
            nn.Conv2d(
                in_channels=32,  # 输入通道数与第二层输出一致
                out_channels=64,  # 增加特征图数量
```

```
                kernel_size=3,    # 使用较小的卷积核
stride=1,                   padding=1
            ),
            nn.BatchNorm2d(64),  # 批归一化，加速收敛并提高稳定性
            nn.ReLU(),
            nn.MaxPool2d(2)  # 输出大小: (64, 3, 3)
        )

        # 调整全连接层输入尺寸
        self.dropout = nn.Dropout(0.5)  # 使用Dropout层防止过拟合
        self.out = nn.Linear(64 * 3 * 3, 10)  # 根据新的特征图尺寸调整

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)  # 通过第三层卷积
        x = x.view(x.size(0), -1)
        x = self.dropout(x)  # 在全连接前加入dropout
        output = self.out(x)
        return output
```

4.3 模型训练与优化

我们使用 GPU 进行模型训练（如果可用）。首先将模型移动到 GPU 设备上，然后定义损失函数和优化器。这里我们使用交叉熵损失函数，它适用于多分类问题。优化器选择 Adam，初始学习率为 0.001。同时，我们使用学习率调度器 StepLR，每 5 个 epoch 将学习率减半，以帮助模型在训练后期更精细地调整参数。

训练过程共进行 10 个 epoch。在每个 epoch 中，我们遍历训练数据加载器，将数据转移到 GPU，然后执行前向传播、计算损失、反向传播和参数更新。每个 epoch 结束后，我们更新学习率，并打印平均损失和当前学习率。

```
# 使用GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNN().to(device)

# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# 增加训练周期
num_epochs = 10  # 10轮

# 学习率调度器
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.5)
```

```
85
86  # 开始训练for epoch in range(num_epochs):
87      model.train()
88      running_loss = 0.0
89
90      for batch_idx, (images, labels) in enumerate(train_loader):
91          images, labels = images.to(device), labels.to(device)
92
93          optimizer.zero_grad()
94          outputs = model(images)
95          loss = criterion(outputs, labels)
96          loss.backward()
97          optimizer.step()
98
99          running_loss += loss.item()
100
101     # 每个epoch后更新学习率
102     scheduler.step()
103
104     # 打印每个epoch的平均损失
105     avg_loss = running_loss / len(train_loader)
106     print(f"Epoch [{epoch + 1}/{num_epochs}], Average Loss: {avg_loss:.4f}, LR: {
            scheduler.get_last_lr()[0]:.6f}")
```

4.4 模型评估

在模型训练完成后，我们使用测试集对模型性能进行评估。首先将模型设置为评估模式（model.eval()），此时禁用 Dropout 等训练时的特定操作。然后，我们遍历测试数据加载器，将数据转移到 GPU，计算模型输出，并统计预测正确的样本数。最后，计算并打印测试准确率。

```
107 model.eval()
108 correct = 0
109 total = 0
110
111 with torch.no_grad():
112     for images, labels in test_loader:
113         images, labels = images.to(device), labels.to(device)
114         outputs = model(images)
115         _, predicted = torch.max(outputs.data, 1)
116         total += labels.size(0)
117         correct += (predicted == labels).sum().item()
118
119 accuracy = 100 * correct / total
120 print(f"Test Accuracy: {accuracy:.2f}%")
```

5 实验结果与分析

5.1 实验结果

经过 10 个 epoch 的训练，模型在 MNIST 测试集上达到了 99.32% 的精确度，达到实验要求。

```
Epoch [1/10], Average Loss: 0.1876, LR: 0.001000
Epoch [2/10], Average Loss: 0.0623, LR: 0.001000
Epoch [3/10], Average Loss: 0.0469, LR: 0.001000
Epoch [4/10], Average Loss: 0.0409, LR: 0.001000
Epoch [5/10], Average Loss: 0.0334, LR: 0.000500
Epoch [6/10], Average Loss: 0.0240, LR: 0.000500
Epoch [7/10], Average Loss: 0.0204, LR: 0.000500
Epoch [8/10], Average Loss: 0.0179, LR: 0.000500
Epoch [9/10], Average Loss: 0.0165, LR: 0.000500
Epoch [10/10], Average Loss: 0.0155, LR: 0.000250
Test Accuracy: 99.32%

(.venv) D:\学科\数学物理\研究生课>
```

5.2 关键改进措施

- 网络深度优化: 增加第三卷积层，提升特征提取能力
- 批归一化: 加入 BatchNorm 层，加速训练收敛
- Dropout 正则化: 防止过拟合，提高泛化能力
- 学习率调度: 动态调整学习率，优化训练过程

6 总结

本次实验成功构建了一个三层卷积神经网络，实现了 MNIST 手写数字的高精度识别。通过合理设计网络结构、优化训练策略和使用正则化技术，模型在测试集上达到了 99.32% 的准确率，充分验证了卷积神经网络在图像分类任务中的有效性。

实验过程中深入理解了卷积神经网络的工作原理、PyTorch 框架的使用方法以及模型优化的关键技术，为后续更复杂的计算机视觉任务奠定了坚实基础。

实验验证: 本实验完全按照实验手册要求，使用 PyTorch 框架构建 CNN 网络，在 MNIST 数据集上训练评估，测试准确率超过 98% 的要求，达到 99.32%，实验目标圆满完成。