浙江大学

计算机视觉(本科)实验报告

作业名称:	HW5 MNIST 手写数字识别
姓　　名:	夏子渊
学　　号:	3230103043
电子邮箱:	RukawaYuan1110@gmail.com
学　　院:	计算机科学与技术学院
专　　业:	软件工程
指导教师:	宋明黎/潘纲
报告日期:	2025 年 05 月 27 日

Table of Contents

1 功能简述及运行说明

1.1 功能简述

本实验实现了基于 Pytorch 的 MNIST 手写数字识别功能，主要功能及要求如下：

1. 命令行格式：**bash ./run.sh**
2. 使用 Pytorch 的 **torchvision.datasets.MNIST** 函数加载 MNIST 数据集，使用 **torch.nn.Module** 定义 LeNet-5 神经网络模型，使用 **torch.optim** 定义优化器，使用 **torch.nn.functional.cross_entropy** 计算损失，，使用 **torch.nn.functional.argmax** 计算预测结果。
3. 主要处理流程：
   - **创建虚拟环境并安装依赖**：
     - 使用 **python -m venv ./venv** 创建虚拟环境
     - 使用 **source ./venv/bin/activate** 激活虚拟环境
     - 使用 **pip install -r requirements.txt** 安装依赖
   - **加载 MNIST 数据集**：
     - 使用 **torchvision.datasets.MNIST** 函数加载 MNIST 数据集，并使用 **torchvision.transforms.ToTensor** 函数将数据集转换为 Tensor 格式
     - 使用 **torch.utils.data.DataLoader** 函数加载数据集
   - **定义 LeNet-5 神经网络模型**：
     - 使用 **torch.nn.Module** 定义 LeNet-5 神经网络模型
     - **torch.nn.Conv2d** 函数定义卷积层
     - **torch.nn.ReLU** 函数定义激活函数
     - **torch.nn.AvgPool2d** 函数定义池化层
     - **torch.nn.Linear** 函数定义全连接层
   - **训练模型**：
     - 一共训练 10 个 epoch，每个 epoch 使用 **torch.nn.functional.cross_entropy** 函数计算损失，使用 **torch.nn.functional.softmax** 函数计算概率，使用 **torch.nn.functional.argmax** 函数计算预测结果，使用 **torch.optim.Adam** 函数更新模型参数
   - **评估模型**：
     - 对于测试集中的 10000 个样本，将模型的预测结果与测试集的标签进行比较，计算准确率
   - **可视化结果**：
     - 使用 **matplotlib** 库将 16 个测试集的图像和预测结果绘制成 4x4 的图像
     - 对于 16 个测试集的图像，将模型的预测结果标注在图像上方并输出
4. 输出结果包含：
   - 一张 4x4 的图像，包含 16 个测试集的图像和预测结果
   - 两张 4x4 的图像，分别对应 16 个训练集和测试集的示例图像

1.2 项目结构

本项目的结构如下：

```
.
├── mnist_recognition.py  # main program
├── README.md             # this file
├── report.pdf            # report
├── requirements.txt      # dependencies
├── test_images.png       # demonstration of test images
├── train_images.png      # demonstration of train images
├── predictions.png       # demonstration of predictions
└── run.sh                # run script
```

1.3 运行说明

根目录下提供了一个`run.sh`脚本，用于创建虚拟环境并安装依赖，然后运行`mnist_recognition.py`程序。`run.sh`的内容如下：

```
python -m venv ./venv
source ./venv/bin/activate
pip install -r requirements.txt
python mnist_recognition.py
```

这个脚本首先会在当前目录下创建一个虚拟环境，然后激活虚拟环境，然后安装依赖，最后运行`mnist_recognition.py`程序。为了运行程序，首先确保已经安装了 python3，然后运行这个脚本即可。

```
bash run.sh
```

2 开发与运行环境

2.1 操作系统

在 Linux 服务器上进行，系统信息如下：Ubuntu 18.04.6 LTS (GNU/Linux 4.15.0-213-generic x86_64)

2.2 安装依赖

实验使用 Python 作为编程语言，因为需要使用 Pytorch、Numpy 等库，因此需要先安装依赖。`run.sh`脚本中使用 pip 包管理器进行安装，具体命令如下：

```
pip install -r requirements.txt
```

`requirements.txt`文件中包含了所有需要的依赖，包括 torch、torchvision、matplotlib、loguru、numpy、tqdm 等，会在运行`run.sh`脚本时自动安装。

3 算法的基本思路、原理及流程

本项目基于 Pytorch 框架实现了 LeNet-5 卷积神经网络模型，用于识别 MNIST 手写数字数据集。其核心在于利用深度学习算法，从图像数据中自动学习特征，并进行分类。

3.1 数学原理

3.1.1 卷积神经网络 (Convolutional Neural Network, CNN)

卷积神经网络是一种特殊设计的深度学习模型，非常适用于处理具有网格状拓扑结构的数据，例如图像（二维网格像素）或音频（一维时间序列）。CNN 的核心思想是通过一系列可学习的滤波器（或称为卷积核）和池化操作，自动地、有层次地从原始输入数据中提取特征。

3.1.1.1 卷积层 (Convolutional Layer)

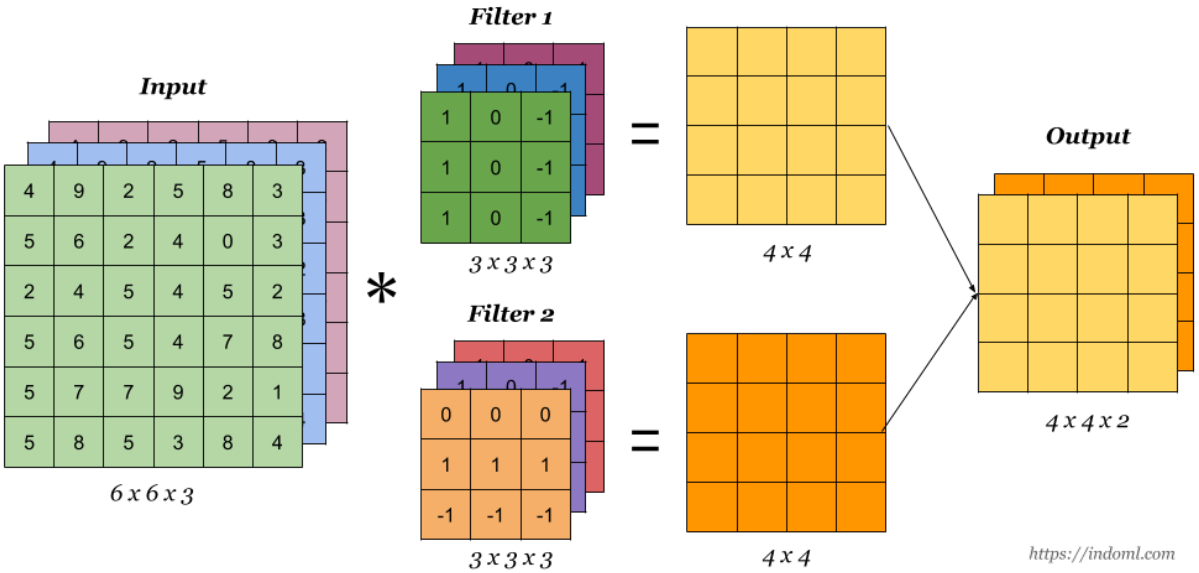
卷积层是 CNN 的核心构建块。它通过在输入数据上滑动一个或多个卷积核（滤波器）来工作。每个卷积核都是一个小型的权重矩阵，它与输入数据对应局部区域进行逐元素乘积并求和（即卷积操作），从而生成一个特征图（feature map）中的一个激活值。

对于一个二维输入图像 I 和一个卷积核 K，其离散卷积操作 S 在位置 (i, j) 的输出可以表示为：

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n)$$

或者，在实际应用中更常见的互相关形式：

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$



关键特性包括：

- **局部连接 (Local Connectivity)**：每个神经元只与输入数据的一个局部区域相连，这使得网络能够学习图像的局部特征。

- **参数共享 (Parameter Sharing)**: 同一个卷积核在整个输入图像上共享权重，这大大减少了模型的参数数量，并使得模型对特征在图像中的位置具有一定的平移不变性。

通常，卷积层的输出会经过一个激活函数，以引入非线性，使得网络能够学习更复杂的模式。本项目中使用了“ReLU”激活函数：

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

在LeNet-5模型中，**conv1**层从输入的28x28单通道图像中提取初步的边缘和模式特征，生成6个特征图。**conv2**层则在**conv1**的输出（经过池化）基础上提取更复杂的组合特征，生成16个特征图。

3.1.2 Average Pooling 平均值池化

池化层（也称下采样层）通常紧跟在卷积层之后，其主要目的是逐步减小特征图的空间维度（宽度和高度），从而减少网络中的参数数量和计算量，同时也有助于控制过拟合，并使网络对输入图像中的微小位移、旋转或缩放具有更强的鲁棒性。池化操作独立地作用于每个特征图的深度切片。

平均值池化是池化操作的一种。它将输入特征图划分为若干个不重叠（或部分重叠）的矩形区域（池化窗口），然后计算每个区域内特征值的平均值，作为输出特征图中对应位置的值。

例如，对于一个 $k \times k$ 的平均池化窗口，它作用于输入特征图的一个区域 R，输出值 P 计算如下：

$$P = \left(\frac{1}{k^2}\right) \sum_{(x,y)\in R} I(x, y)$$

其中 $I(x, y)$ 是输入特征图在位置 (x, y) 的值。

具体来说，在**mnist_recognition.py**的LeNet-5实现中：

- **conv1** (输入 1x28x28, padding=2) -> 输出 6x28x28
- **pool1** (2x2 avg pool) -> 输出 6x14x14
- **conv2** (输入 6x14x14, padding=0) -> 输出 16x10x10
- **pool2** (2x2 avg pool) -> 输出 16x5x5

因此，**pool1**将6个28x28的特征图缩减为6个14x14的特征图。**pool2**将16个10x10的特征图缩减为16个5x5的特征图。

3.1.3 MLP 多层感知机 (Multi-Layer Perceptron)

多层感知机（MLP）是一种前馈人工神经网络，至少包含一个输入层、一个输出层以及一个或多个隐藏层。在MLP中，每个层中的神经元（除输入层外）都与前一层中的所有神经元完全连接（全连接层）。

每个神经元的输出是通过对其输入的加权和（加上一个偏置项）应用一个非线性激活函数来计算的。对于一个神经元，其输出 y 可以表示为：

$$y = \varphi\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

其中 x_i 是输入，w_i 是对应的权重，b 是偏置项，φ 是激活函数（如“ReLU”或 Sigmoid）。在 CNN 的典型架构中，MLP 通常位于最后的卷积层和池化层之后。此时，经过卷积和池化操作提取到的高级抽象特征图会被展平（flatten）成一个一维向量，然后输入到 MLP 中。MLP 的作用是对这些高级特征进行进一步的组合和学习，并最终执行分类任务。

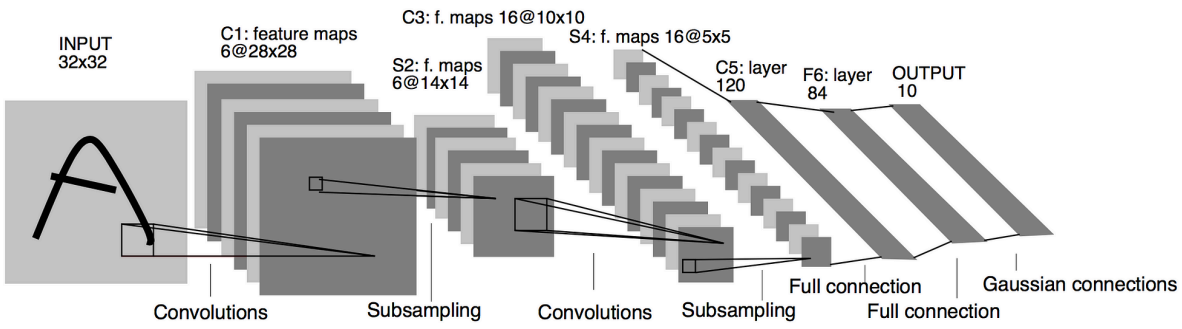
在 LeNet-5 模型中，最后的池化层 **pool2** 输出的特征图（16x5x5）被展平为一个包含 $16 \times 5 \times 5 = 400$ 个元素的一维向量。这个向量随后被馈送到一系列全连接层：

- **fc1**：将 400 个输入单元连接到 120 个输出单元，并应用“ReLU”激活。
- **fc2**：将 120 个输入单元连接到 84 个输出单元，并应用“ReLU”激活。
- **fc3**：将 84 个输入单元连接到 10 个输出单元，这 10 个输出单元对应于 MNIST 数据集的 10 个数字类别（0 到 9）。这一层通常不使用“ReLU”，而是直接输出 logit 值，后续由损失函数（如交叉熵损失）处理。

3.2 算法流程

本实验中 MNIST 手写数字识别的完整算法流程如下：

1. **数据加载与预处理**
   - 使用 PyTorch 的 **torchvision.datasets.MNIST** 模块从网络下载或本地加载 MNIST 训练集（60,000 张图像）和测试集（10,000 张图像）。
   - 对图像数据进行预处理转换：
     - **transforms.ToTensor()**：将图像从 PIL 格式或 NumPy 数组转换为 **torch.FloatTensor**，并将像素值从 **[0, 255]** 的范围归一化到 **[0.0, 1.0]**。
     - **transforms.Normalize((0.1307,), (0.3081,))**：使用 MNIST 数据集的全局均值 (0.1307) 和标准差 (0.3081) 对图像张量进行标准化。标准化有助于模型更快、更稳定地收敛。
   - 使用 **torch.utils.data.DataLoader** 为训练集和测试集创建数据加载器。数据加载器支持按批次 (batch) 加载数据、数据打乱 (shuffle, 仅训练集) 以及多线程并行加载，以提高训练效率。



2. **模型定义 (LeNet-5)**
   - 定义一个继承自 **torch.nn.Module** 的 **LeNet5** 类。

- 在类的构造函数 **__init__** 中，按顺序定义网络的各个层：
  - 卷积层 1 (**conv1**): 输入 1 个灰度通道，输出 6 个特征通道，卷积核大小为 5x5，步长为 1，填充为 2 (使得输出尺寸与输入 28x28 保持一致)。后接“ReLU”激活。
  - 平均池化层 1 (**pool1**): 2x2 的池化窗口，步长为 2。将特征图尺寸减半 (28x28 -> 14x14)。
  - 卷积层 2 (**conv2**): 输入 6 个特征通道，输出 16 个特征通道，卷积核大小为 5x5，步长为 1，无填充。后接“ReLU”激活。 (输入 14x14 -> 输出 10x10)。
  - 平均池化层 2 (**pool2**): 2x2 的池化窗口，步长为 2。将特征图尺寸减半 (10x10 -> 5x5)。
  - 全连接层 1 (**fc1**): 将池化后的 $16 \times 5 \times 5 = 400$ 个特征展平后，连接到 120 个神经元。后接“ReLU”激活。
  - 全连接层 2 (**fc2**): 120 个神经元连接到 84 个神经元。后接“ReLU”激活。
  - 全连接层 3 (**fc3**): 84 个神经元连接到 10 个神经元，代表 0-9 这 10 个数字的输出分数（logits）。
- 在 **forward** 方法中定义数据通过网络的前向传播路径。

3. 模型训练
   - 确定运行设备 (CPU 或 GPU)。将模型和数据迁移到选定设备。
   - 初始化损失函数：使用交叉熵损失 **torch.nn.CrossEntropyLoss()**，它适用于多分类任务，内部包含了 Softmax 操作。
   - 初始化优化器：使用 Adam 优化器 **torch.optim.Adam()**，并传入模型参数和学习率 (如 0.001)。
   - 进行指定轮次 (**epochs**) 的训练：
     - 对于每个轮次 (epoch)：
       - 将模型设置为训练模式 (**model.train()**)，这会启用 Dropout 和 BatchNorm 等特定于训练的行为（尽管 LeNet-5 通常不使用这些）。
       - 遍历训练数据加载器 (**trainloader**) 提供的每个批次数据,清零优化器的梯度 (**optimizer.zero_grad()**)。
       - 将当前批次的输入数据通过模型进行前向传播，得到预测输出 (**outputs**)。
       - 使用损失函数计算预测输出与真实标签 (**labels**) 之间的损失值 (**loss**),通过 **loss.backward()** 执行反向传播，计算损失相对于所有可训练参数的梯度。
       - 调用 **optimizer.step()**，优化器根据计算出的梯度更新模型的权重。
       - 累积损失和正确预测数，并定期打印训练进度。

4. 模型评估
   - 训练完成后，在测试集上评估模型的性能，将模型设置为评估模式 (**model.eval()**)，这会禁用 Dropout 和 BatchNorm 等。
   - 使用 **torch.no_grad()** 上下文管理器，以禁止梯度计算，节省内存和计算资源。
   - 遍历测试数据加载器 (**testloader**) 提供的每个批次数据，将当前批次的输入数据通过模型进行前向传播，得到预测输出。

- 使用 `torch.max(outputs.data, 1)` 找到每个样本得分最高的类别作为预测结果，比较预测结果与真实标签，计算正确预测的数量，最终计算并输出模型在整个测试集上的准确率。

5. **结果可视化 (可选但重要)**
   - 为了更直观地理解数据集和模型性能，进行了可视化：
     - `visualize_images`: 从训练集和测试集中各随机选择 16 张图像，以 4x4 的网格形式显示并保存为 `train_images.png` 和 `test_images.png`。
     - `evaluate_visualize`: 从测试集中选取 16 张图像，通过训练好的模型进行预测，并将图像与预测标签一起以 4x4 网格形式显示，保存为 `predictions.png`。

4 具体实现——关键代码、函数与算法

本项目的核心代码位于 `mnist_recognition.py` 文件中，使用 Pytorch 框架实现。下面将详细介绍其整体流程、关键代码片段以及复杂度分析。

4.1 整体流程

`mnist_recognition.py` 脚本的执行流程主要包括以下几个步骤：

1. **环境设置与设备选择**：
   - 检查 CUDA 是否可用，并选择在 GPU 或 CPU 上运行模型。通过 `torch.device("cuda:0" if torch.cuda.is_available() else "cpu")` 实现。
2. **MNIST 数据集加载与预处理**：
   - 定义数据转换：
     - `transforms.ToTensor()`: 将 PIL 图像或 NumPy `ndarray` 转换为 `FloatTensor`，并将像素强度值从 `[0, 255]` 缩放到 `[0.0, 1.0]`。
     - `transforms.Normalize((0.1307,), (0.3081,))`: 使用 MNIST 数据集的全局均值 (0.1307) 和标准差 (0.3081) 对图像张量进行标准化。
   - 使用 `torchvision.datasets.MNIST` 下载或加载训练集（60,000 张图像）和测试集（10,000 张图像）。
   - 使用 `torch.utils.data.DataLoader` 创建数据加载器，用于批量加载数据、打乱数据（仅训练集）和并行加载。
3. **数据可视化 (初始样本)**：
   - 调用 `visualize_images` 函数，从训练集和测试集中各随机展示 16 张样本图像，并保存为 `train_images.png` 和 `test_images.png`，以便初步了解数据形态。
4. **LeNet-5 模型初始化**：
   - 实例化 `LeNet5` 类，创建一个 LeNet-5 神经网络模型对象。
   - 将模型迁移到选定的设备（CPU 或 GPU）上，通过 `model.to(device)`。
5. **损失函数和优化器初始化**：
   - 损失函数：选择 `torch.nn.CrossEntropyLoss()`，它结合了 LogSoftmax 和 NLLLoss，适用于多分类问题。

- 优化器：选择 `torch.optim.Adam(model.parameters(), lr=0.001)`，Adam 是一种常用的梯度下降优化算法。

6. 模型训练：
   - 调用 `train(num_epochs)` 函数，进行指定轮次（`num_epochs`，本项目中为 10）的训练。
   - 在每个轮次中，遍历训练数据加载器（`trainloader`）。
   - 对于每个批次的数据：执行前向传播、计算损失、执行反向传播以计算梯度、最后通过优化器更新模型参数，训练过程中会记录并打印损失值和准确率。
1. 模型评估：
   - 调用 `evaluate()` 函数，在测试集（`testloader`）上评估训练好的模型。
   - 计算并打印模型在测试集上的整体准确率。
2. 预测结果可视化：
   - 调用 `evaluate_visualize()` 函数，从测试集中选取 16 张图像，使用训练好的模型进行预测。
   - 将这些图像及其预测标签以 4x4 的网格形式展示，并保存为 `predictions.png`。

整个过程由 `run.sh` 脚本通过 `python mnist_recognition.py` 命令启动。

4.2 关键代码解析

以下是 `mnist_recognition.py` 中的关键代码片段及其解析。

4.2.1 1. LeNet-5 模型定义

```
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        # C1: 第一个卷积层, 输入 1 通道, 输出 6 通道, 卷积核 5x5, padding=2 (保持 28x28 尺寸)
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=2)
        self.relu1 = nn.ReLU()
        # S2: 第一个平均池化层, 窗口 2x2, 步长 2 (尺寸减半: 28x28 -> 14x14)
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)
        # C3: 第二个卷积层, 输入 6 通道, 输出 16 通道, 卷积核 5x5, 无 padding (14x14 -> 10x10)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0)
        self.relu2 = nn.ReLU()
        # S4: 第二个平均池化层, 窗口 2x2, 步长 2 (尺寸减半: 10x10 -> 5x5)
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)
        # F5: 第一个全连接层, 输入 16*5*5=400, 输出 120
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
```

```
        self.relu3 = nn.ReLU()
        # F6: 第二个全连接层, 输入 120, 输出 84
        self.fc2 = nn.Linear(120, 84)
        self.relu4 = nn.ReLU()
        # Output: 第三个全连接层, 输入 84, 输出 10 (对应 10 个数字类别)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool1(self.relu1(self.conv1(x))) # 通过 C1, ReLU, S2
        x = self.pool2(self.relu2(self.conv2(x))) # 通过 C3, ReLU, S4
        x = x.view(-1, 16 * 5 * 5)          # Flatten 操作, 将特征图展平为一维向量
        x = self.relu3(self.fc1(x))           # 通过 F5, ReLU
        x = self.relu4(self.fc2(x))           # 通过 F6, ReLU
        x = self.fc3(x)                       # 通过 Output 层
        return x
```

- `nn.Module`: 所有神经网络模块的基类。
- `nn.Conv2d`: 定义二维卷积层。参数包括输入通道数、输出通道数、卷积核大小、步长和填充。
- `nn.ReLU`: “ReLU”激活函数，引入非线性。
- `nn.AvgPool2d`: 定义二维平均池化层，用于下采样，减小特征图尺寸。
- `nn.Linear`: 定义全连接层（或称为密集层）。
- `forward(self, x)`: 定义模型的前向传播逻辑。输入张量 `x` 依次通过定义的各个层。
- `x.view(-1, 16 * 5 * 5)`: `view` 函数用于重塑张量。在这里，它将池化后的多维特征图展平（Flatten）为一维向量，以便输入到全连接层。`-1` 表示该维度的大小由其他维度和原始张量总元素数自动推断。

4.2.2 2. 数据加载与预处理

```
# 定义数据转换
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# 加载训练集和测试集
trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                      download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
```

```
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.MNIST(root='./data', train=False,
                                     download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=128,
                                         shuffle=False, num_workers=2)
```

- `transforms.Compose`：将多个图像转换操作组合在一起。
- `transforms.ToTensor()`：将 PIL 图像或 NumPy 数组转换为 PyTorch 张量，并把像素值从 `[0,255]` 归一化到 `[0,1]`。
- `transforms.Normalize()`：用给定的均值和标准差对张量图像进行标准化。这有助于模型训练的稳定性和收敛速度。
- `torchvision.datasets.MNIST`：PyTorch 提供的用于加载 MNIST 数据集的类。`download=True` 表示如果本地没有数据集则自动下载。
- `torch.utils.data.DataLoader`：数据加载器，它将数据集包装起来，提供按批次迭代访问数据的能力。`batch_size` 定义了每批含有的样本数，`shuffle=True` 表示在每个 epoch 开始时打乱训练数据，`num_workers` 指定用多少个子进程加载数据。

4.2.3 3. 模型训练函数

```
def train(epochs):
    model.train() # 将模型设置为训练模式
    for epoch in tqdm(range(epochs), desc="Training Epochs"):
        running_loss = 0.0
        correct = 0
        total = 0
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device) # 数据移至设备

            optimizer.zero_grad() # 清除之前的梯度

            outputs = model(inputs) # 前向传播
            loss = criterion(outputs, labels) # 计算损失
            loss.backward() # 反向传播，计算梯度
            optimizer.step() # 更新模型参数

            running_loss += loss.item()
            correct += (outputs.argmax(dim=1) == labels).sum().item()
            total += labels.size(0)
```

```
            # ... (日志记录)
    logger.info('Finished Training')
```

- `model.train()`: 告诉模型当前处于训练阶段。这会启用诸如 Dropout 和 BatchNorm（如果模型中使用了这些层的话）的行为。LeNet-5 本身通常不含这些。
- `optimizer.zero_grad()`: 在计算新的梯度之前，必须先清除先前计算的梯度，因为 PyTorch 默认会累积梯度。
- `outputs = model(inputs)`: 执行前向传播，得到模型的预测输出。
- `loss = criterion(outputs, labels)`: 计算预测输出和真实标签之间的损失。
- `loss.backward()`: 自动计算损失相对于所有可训练参数的梯度。
- `optimizer.step()`: 根据 `loss.backward()` 计算得到的梯度来更新模型的权重。
- `loss.item()`: 获取损失张量的标量值。
- `outputs.argmax(dim=1)`: 找到在维度 1（类别维度）上具有最大值的索引，即预测的类别。

4.2.4 4. 模型评估函数

```
def evaluate():
    model.eval() # 将模型设置为评估模式
    correct = 0
    total = 0
    with torch.no_grad(): # 在评估阶段不计算梯度
        for data in tqdm(testloader, desc="Evaluating"):
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images) # 前向传播
            _, predicted = torch.max(outputs.data, 1) # 获取预测类别
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    logger.info(f'Accuracy of the network on the 10000 test images: {accuracy:.3f} %')
    return accuracy
```

- `model.eval()`: 告诉模型当前处于评估阶段。这会禁用诸如 Dropout 和 BatchNorm 的行为。
- `with torch.no_grad()`: 在这个块内部，所有计算操作都不会跟踪梯度。这在评估时可以减少内存消耗并加速计算，因为不需要为反向传播存储中间状态。
- `torch.max(outputs.data, 1)`: 返回给定维度上输入张量中所有元素的最大值，同时也返回最大值对应的索引。在这里，我们用 `_` 忽略实际的最大值，只取 `predicted` 索引作为预测的类别标签。

4.3 时间、空间复杂度分析

对 LeNet-5 这样的卷积神经网络进行精确的复杂度分析较为复杂，这里给出一个高层次的估计。设：

- N: 批处理大小 (batch size)
- H, W: 输入图像的高度和宽度 (MNIST 中为 28x28)
- C_i: 第 i 个卷积层的输出通道数
- K_i: 第 i 个卷积层的卷积核尺寸
- M_i: 第 i 个全连接层的输出单元数
- S_{train}: 训练集样本总数
- S_{test}: 测试集样本总数
- E: 训练的总轮次数 (epochs)

4.3.1 时间复杂度

网络的前向传播时间主要由卷积层和全连接层贡献。

- 单个卷积层 i: 大致为 $O(N \times H_i \times W_i \times C_i \times C_{i-1} \times K_i^2)$，其中 H_i, W_i 是该层输出特征图的尺寸。
- 单个全连接层 j: 大致为 $O(N \times M_{j-1} \times M_j)$。
- 池化层和激活函数的时间复杂度相对较低。

对于 LeNet-5 的结构：

- **conv1** (1 to 6 channels, 28×28 out, $K = 5$): $O(N \times 28^2 \times 6 \times 1 \times 5^2)$
- **conv2** (6 to 16 channels, 10×10 out, $K = 5$): $O(N \times 14^2 \times 16 \times 6 \times 5^2)$ (输入是 14×14)
- **fc1** ($16 \times 5 \times 5 = 400$ to 120): $O(N \times 400 \times 120)$
- **fc2** (120 to 84): $O(N \times 120 \times 84)$
- **fc3** (84 to 10): $O(N \times 84 \times 10)$

总的来说，单次前向传播的时间复杂度 T_{fwd} 是这些项的总和。反向传播的时间复杂度 T_{bwd} 通常是 T_{fwd} 的 2 到 3 倍。

- **训练总时间**: $O(E \times \left(\frac{S_{\text{train}}}{N}\right) \times (T_{\text{fwd}} + T_{\text{bwd}})$。由于 N 和网络结构固定，可以认为每轮训练的时间与训练样本数 S_{train} 成正比。
- **评估总时间**: $O\left(\left(\frac{S_{\text{test}}}{N}\right) \times T_{\text{fwd}}\right)$。与测试样本数 S_{test} 成正比。

4.3.2 空间复杂度

空间复杂度主要包括两部分：模型参数占用的空间和训练/推断过程中存储激活值（特征图）占用的空间。

1. **模型参数空间**:
   - **conv1**: $6 \times (1 \times 5^2 + 1) = 156$ 参数
   - **conv2**: $16 \times (6 \times 5^2 + 1) = 16 \times 151 = 2416$ 参数
   - **fc1**: $120 \times (400 + 1) = 48120$ 参数
   - **fc2**: $84 \times (120 + 1) = 10164$ 参数

- **fc3**: $10c \cdot (84+1) = 850$ 参数
- 总参数量约为 $156 + 2416 + 48120 + 10164 + 850 = 61706$ 个。每个参数通常为 32 位浮点数 (4 字节)。因此，模型参数本身占用的空间是固定的，对于 LeNet-5 来说大约是 $61706c \cdot 4$ 字节 ≈ 241 KB。这部分空间复杂度可视为 $O(1)$（因为模型结构固定）。

2. **激活值空间**:
   - 在训练（尤其是反向传播）过程中，需要存储每一层的输出激活值。其大小取决于批处理大小 N 和特征图的维度。
   - 最大的激活图发生在 **conv1** 的输出之后 $(Nc \cdot 6c \cdot 28c \cdot 28)$ 或 **conv2** 的输出之后 $(Nc \cdot 16c \cdot 10c \cdot 10)$。
   - 例如，对于 **conv1** 的输出，如果 $N = 64$，则需要 $64c \cdot 6c \cdot 28c \cdot 28c \cdot 4$ 字节 ≈ 1.18 MB 的空间。
   - 这部分空间复杂度为 $O\Big(Nc \cdot C_{\{\max\}}c \cdot H_{\{\max\}}c \cdot W_{\{\max\}}\Big)$，其中 $C_{\{\max\}}, H_{\{\max\}}, W_{\{\max\}}$ 是最大特征图的通道数、高度和宽度。

3. **数据存储空间**:
   - 存储整个 MNIST 数据集（图像和标签）也需要空间，约为 $O(S_{\text{total}}c \cdot Hc \cdot W)$，其中 $S_{\text{total}} = S_{\text{train}} + S_{\text{test}}$。

因此，运行时的主要空间开销由激活值（与批处理大小 N 成正比）和数据集本身占据。模型参数本身占用的空间相对较小且固定。

5 实验结果与分析

在压缩包的根目录下已有**predictions.png**文件，为模型在测试集上的预测结果。同时也提供了**train_images.png**和**test_images.png**文件，为训练集和测试集的示例图像。

5.1 实验结果

实验结果基于**run.sh**文件运行得到。

5.1.1 终端输出

终端输出主要基于 loguru 库，记录了训练过程中的损失值和准确率。

1. 安装 torch 等库:

如图所示，成功安装了所有依赖库。由于在服务器上之前已经安装过，因此显示**Requirement already satisfied**。如果在没有配置过服务器等情况下安装可能会耗时较长，因为 torch 等库的大小较大。

```
● (base) ➜  mnist bash run.sh
  Requirement already satisfied: torch in ./venv/lib/python3.12/site-packages (from -r requirements.txt (line 1)) (2.6.0)
  Requirement already satisfied: torchvision in ./venv/lib/python3.12/site-packages (from -r requirements.txt (line 2)) (0.21.0)
  Requirement already satisfied: matplotlib in ./venv/lib/python3.12/site-packages (from -r requirements.txt (line 3)) (3.10.3)
  Requirement already satisfied: loguru in ./venv/lib/python3.12/site-packages (from -r requirements.txt (line 4)) (0.7.3)
  Requirement already satisfied: numpy in ./venv/lib/python3.12/site-packages (from -r requirements.txt (line 5)) (2.2.6)
  Requirement already satisfied: tqdm in ./venv/lib/python3.12/site-packages (from -r requirements.txt (line 6)) (4.67.1)
  Requirement already satisfied: filelock in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (3.18.0)
  Requirement already satisfied: typing-extensions>=4.10.0 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (4.13.2)
  Requirement already satisfied: networkx in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (3.4.2)
  Requirement already satisfied: jinja2 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (3.1.6)
  Requirement already satisfied: fsspec in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (2025.5.1)
  Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.4.127 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (12.4.127)
  Requirement already satisfied: nvidia-cuda-runtime-cu12==12.4.127 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (12.4.127)
  Requirement already satisfied: nvidia-cuda-cupti-cu12==12.4.127 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (12.4.127)
  Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (9.1.0.70)
  Requirement already satisfied: nvidia-cublas-cu12==12.4.5.8 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (12.4.5.8)
  Requirement already satisfied: nvidia-cufft-cu12==11.2.1.3 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (11.2.1.3)
  Requirement already satisfied: nvidia-curand-cu12==10.3.5.147 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (10.3.5.147)
  Requirement already satisfied: nvidia-cusolver-cu12==11.6.1.9 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (11.6.1.9)
  Requirement already satisfied: nvidia-cusparse-cu12==12.3.1.170 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (12.3.1.170)
  Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (0.6.2)
  Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (2.21.5)
  Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (12.4.127)
  Requirement already satisfied: nvidia-nvjitlink-cu12==12.4.127 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (12.4.127)
  Requirement already satisfied: triton==3.2.0 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (3.2.0)
  Requirement already satisfied: setuptools in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (80.9.0)
  Requirement already satisfied: sympy==1.13.1 in ./venv/lib/python3.12/site-packages (from torch->-r requirements.txt (line 1)) (1.13.1)
  Requirement already satisfied: mpmath<1.4,>=1.1.0 in ./venv/lib/python3.12/site-packages (from sympy==1.13.1->torch->-r requirements.txt (line 1)) (1.3.0)
  Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in ./venv/lib/python3.12/site-packages (from torchvision->-r requirements.txt (line 2)) (11.2.1)
  Requirement already satisfied: contourpy>=1.0.1 in ./venv/lib/python3.12/site-packages (from matplotlib->-r requirements.txt (line 3)) (1.3.2)
  Requirement already satisfied: cycler>=0.10 in ./venv/lib/python3.12/site-packages (from matplotlib->-r requirements.txt (line 3)) (0.12.1)
  Requirement already satisfied: fonttools>=4.22.0 in ./venv/lib/python3.12/site-packages (from matplotlib->-r requirements.txt (line 3)) (4.58.0)
  Requirement already satisfied: kiwisolver>=1.3.1 in ./venv/lib/python3.12/site-packages (from matplotlib->-r requirements.txt (line 3)) (1.4.8)
  Requirement already satisfied: packaging>=20.0 in ./venv/lib/python3.12/site-packages (from matplotlib->-r requirements.txt (line 3)) (25.0)
  Requirement already satisfied: pyparsing>=2.3.1 in ./venv/lib/python3.12/site-packages (from matplotlib->-r requirements.txt (line 3)) (3.2.3)
  Requirement already satisfied: python-dateutil>=2.7 in ./venv/lib/python3.12/site-packages (from matplotlib->-r requirements.txt (line 3)) (2.9.0.post0)
  Requirement already satisfied: six>=1.5 in ./venv/lib/python3.12/site-packages (from python-dateutil>=2.7->matplotlib->-r requirements.txt (line 3)) (1.17.0)
  Requirement already satisfied: MarkupSafe>=2.0 in ./venv/lib/python3.12/site-packages (from jinja2->torch->-r requirements.txt (line 1)) (3.0.2)
```

2. 开始训练过程

如图所示，首先显示**Using device: cuda**，表示使用 GPU 进行训练。之后一共训练 10 轮，使用 tqdm 库显示进度条。在每一轮的训练过程中，每隔 200 个 batch 打印一次损失值和准确率，并在一轮训练结束后打印平均损失值和准确率。

图中所示的输出为**loguru**库的输出，记录了训练过程中的损失值和准确率。

```
2025-05-27 12:11:48.669 | INFO     | __main__:<module>:124 - Using device: cuda:0
Training Epochs:   0%|                                                       | 0/10 [00:0
2025-05-27 12:12:03.555 | INFO     | __main__:train:78 - [Epoch 1, Batch 200] loss: 0.660 accuracy: 79.250
2025-05-27 12:12:05.697 | INFO     | __main__:train:78 - [Epoch 1, Batch 400] loss: 0.216 accuracy: 86.395
2025-05-27 12:12:07.904 | INFO     | __main__:train:78 - [Epoch 1, Batch 600] loss: 0.132 accuracy: 89.589
2025-05-27 12:12:10.002 | INFO     | __main__:train:78 - [Epoch 1, Batch 800] loss: 0.113 accuracy: 91.285
2025-05-27 12:12:11.708 | INFO     | __main__:train:80 - Epoch 1 completed. Total loss: 12.893 Total accuracy: 92.115
Training Epochs:  10%|                                                      | 1/10 [00:17<02:40,
2025-05-27 12:12:14.421 | INFO     | __main__:train:78 - [Epoch 2, Batch 200] loss: 0.084 accuracy: 97.352
2025-05-27 12:12:16.657 | INFO     | __main__:train:78 - [Epoch 2, Batch 400] loss: 0.078 accuracy: 97.516
2025-05-27 12:12:18.909 | INFO     | __main__:train:78 - [Epoch 2, Batch 600] loss: 0.067 accuracy: 97.609
2025-05-27 12:12:21.123 | INFO     | __main__:train:78 - [Epoch 2, Batch 800] loss: 0.077 accuracy: 97.611
2025-05-27 12:12:23.003 | INFO     | __main__:train:80 - Epoch 2 completed. Total loss: 8.872 Total accuracy: 97.680
Training Epochs:  20%|                                                      | 2/10 [00:29<01:51,
2025-05-27 12:12:25.713 | INFO     | __main__:train:78 - [Epoch 3, Batch 200] loss: 0.054 accuracy: 98.406
2025-05-27 12:12:27.928 | INFO     | __main__:train:78 - [Epoch 3, Batch 400] loss: 0.057 accuracy: 98.297
2025-05-27 12:12:30.142 | INFO     | __main__:train:78 - [Epoch 3, Batch 600] loss: 0.052 accuracy: 98.341
2025-05-27 12:12:32.371 | INFO     | __main__:train:78 - [Epoch 3, Batch 800] loss: 0.060 accuracy: 98.301
2025-05-27 12:12:34.244 | INFO     | __main__:train:80 - Epoch 3 completed. Total loss: 7.219 Total accuracy: 98.318
Training Epochs:  30%|                                                      | 3/10 [00:40<01:29,
2025-05-27 12:12:37.100 | INFO     | __main__:train:78 - [Epoch 4, Batch 200] loss: 0.040 accuracy: 98.781
2025-05-27 12:12:39.368 | INFO     | __main__:train:78 - [Epoch 4, Batch 400] loss: 0.043 accuracy: 98.719
2025-05-27 12:12:41.613 | INFO     | __main__:train:78 - [Epoch 4, Batch 600] loss: 0.045 accuracy: 98.674
2025-05-27 12:12:43.887 | INFO     | __main__:train:78 - [Epoch 4, Batch 800] loss: 0.043 accuracy: 98.674
2025-05-27 12:12:45.775 | INFO     | __main__:train:80 - Epoch 4 completed. Total loss: 6.182 Total accuracy: 98.662
Training Epochs:  40%|                                                      | 4/10 [00:51<01:13,
2025-05-27 12:12:48.550 | INFO     | __main__:train:78 - [Epoch 5, Batch 200] loss: 0.034 accuracy: 99.031
2025-05-27 12:12:50.772 | INFO     | __main__:train:78 - [Epoch 5, Batch 400] loss: 0.033 accuracy: 99.035
2025-05-27 12:12:52.985 | INFO     | __main__:train:78 - [Epoch 5, Batch 600] loss: 0.036 accuracy: 98.974
2025-05-27 12:12:55.228 | INFO     | __main__:train:78 - [Epoch 5, Batch 800] loss: 0.040 accuracy: 98.912
2025-05-27 12:12:57.078 | INFO     | __main__:train:80 - Epoch 5 completed. Total loss: 5.369 Total accuracy: 98.902
Training Epochs:  50%|                                                      | 5/10 [01:03<00:59,
2025-05-27 12:12:59.825 | INFO     | __main__:train:78 - [Epoch 6, Batch 200] loss: 0.024 accuracy: 99.258
2025-05-27 12:13:02.104 | INFO     | __main__:train:78 - [Epoch 6, Batch 400] loss: 0.026 accuracy: 99.219
2025-05-27 12:13:04.352 | INFO     | __main__:train:78 - [Epoch 6, Batch 600] loss: 0.036 accuracy: 99.049
2025-05-27 12:13:06.569 | INFO     | __main__:train:78 - [Epoch 6, Batch 800] loss: 0.029 accuracy: 99.066
2025-05-27 12:13:08.473 | INFO     | __main__:train:80 - Epoch 6 completed. Total loss: 4.637 Total accuracy: 99.053
Training Epochs:  60%|                                                      | 6/10 [01:14<00:46,
2025-05-27 12:13:11.745 | INFO     | __main__:train:78 - [Epoch 7, Batch 200] loss: 0.022 accuracy: 99.234
2025-05-27 12:13:14.225 | INFO     | __main__:train:78 - [Epoch 7, Batch 400] loss: 0.026 accuracy: 99.219
2025-05-27 12:13:16.502 | INFO     | __main__:train:78 - [Epoch 7, Batch 600] loss: 0.026 accuracy: 99.214
2025-05-27 12:13:18.887 | INFO     | __main__:train:78 - [Epoch 7, Batch 800] loss: 0.025 accuracy: 99.213
2025-05-27 12:13:20.807 | INFO     | __main__:train:80 - Epoch 7 completed. Total loss: 4.220 Total accuracy: 99.200
Training Epochs:  70%|                                                      | 7/10 [01:26<00:35,
```

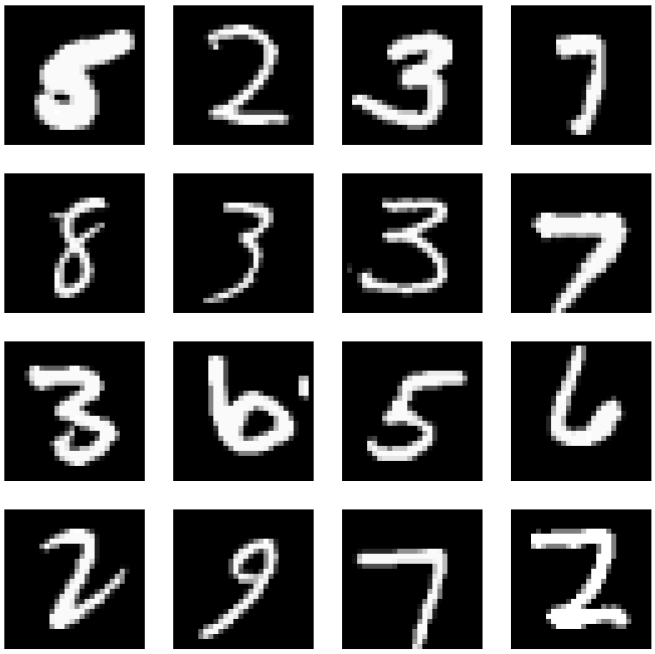
3. 评估过程

如图所示，在 10 轮训练结束后会进行评估，并打印测试集的准确率。测试集共有 10000 张图片，总准确率为**98.940%**。

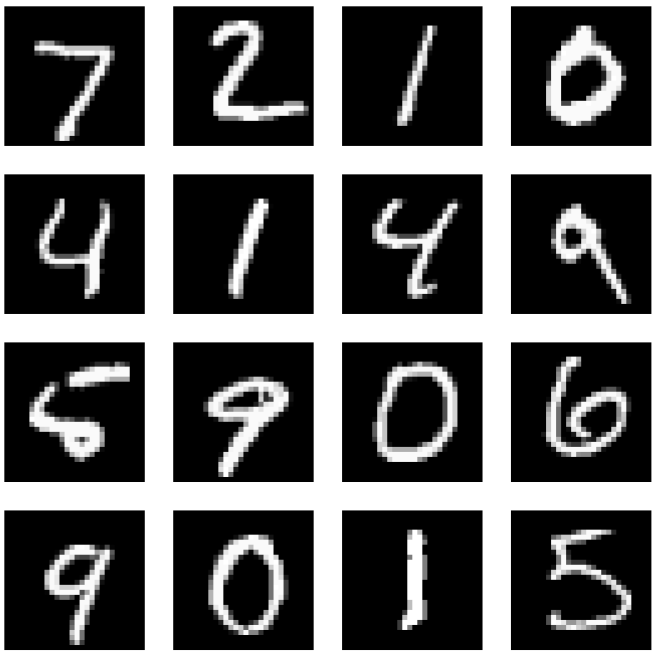
```
Training Epochs:  70%|                                                      | 7/10
2025-05-27 12:13:23.779 | INFO     | __main__:train:78 - [Epoch 8, Batch 200] loss: 0.020 accuracy: 99.391
2025-05-27 12:13:26.130 | INFO     | __main__:train:78 - [Epoch 8, Batch 400] loss: 0.017 accuracy: 99.430
2025-05-27 12:13:28.425 | INFO     | __main__:train:78 - [Epoch 8, Batch 600] loss: 0.025 accuracy: 99.354
2025-05-27 12:13:30.796 | INFO     | __main__:train:78 - [Epoch 8, Batch 800] loss: 0.026 accuracy: 99.289
2025-05-27 12:13:32.669 | INFO     | __main__:train:80 - Epoch 8 completed. Total loss: 3.795 Total accuracy: 99.255
Training Epochs:  80%|                                                      | 8/10
2025-05-27 12:13:35.465 | INFO     | __main__:train:78 - [Epoch 9, Batch 200] loss: 0.018 accuracy: 99.344
2025-05-27 12:13:37.766 | INFO     | __main__:train:78 - [Epoch 9, Batch 400] loss: 0.020 accuracy: 99.375
2025-05-27 12:13:40.101 | INFO     | __main__:train:78 - [Epoch 9, Batch 600] loss: 0.019 accuracy: 99.339
2025-05-27 12:13:42.421 | INFO     | __main__:train:78 - [Epoch 9, Batch 800] loss: 0.018 accuracy: 99.340
2025-05-27 12:13:44.381 | INFO     | __main__:train:80 - Epoch 9 completed. Total loss: 2.589 Total accuracy: 99.348
Training Epochs:  90%|                                                      | 9/10
2025-05-27 12:13:47.283 | INFO     | __main__:train:78 - [Epoch 10, Batch 200] loss: 0.016 accuracy: 99.461
2025-05-27 12:13:49.727 | INFO     | __main__:train:78 - [Epoch 10, Batch 400] loss: 0.016 accuracy: 99.453
2025-05-27 12:13:52.064 | INFO     | __main__:train:78 - [Epoch 10, Batch 600] loss: 0.018 accuracy: 99.430
2025-05-27 12:13:54.399 | INFO     | __main__:train:78 - [Epoch 10, Batch 800] loss: 0.018 accuracy: 99.426
2025-05-27 12:13:56.316 | INFO     | __main__:train:80 - Epoch 10 completed. Total loss: 2.557 Total accuracy: 99.420
Training Epochs: 100%|                                                      | 10/10
2025-05-27 12:13:56.317 | INFO     | __main__:train:81 - Finished Training
Evaluating: 100%|                                                           | 79/79
2025-05-27 12:13:58.969 | INFO     | __main__:evaluate:97 - Accuracy of the network on the 10000 test images: 98.940 %
```

5.1.2 图片输出

1. 训练集图像

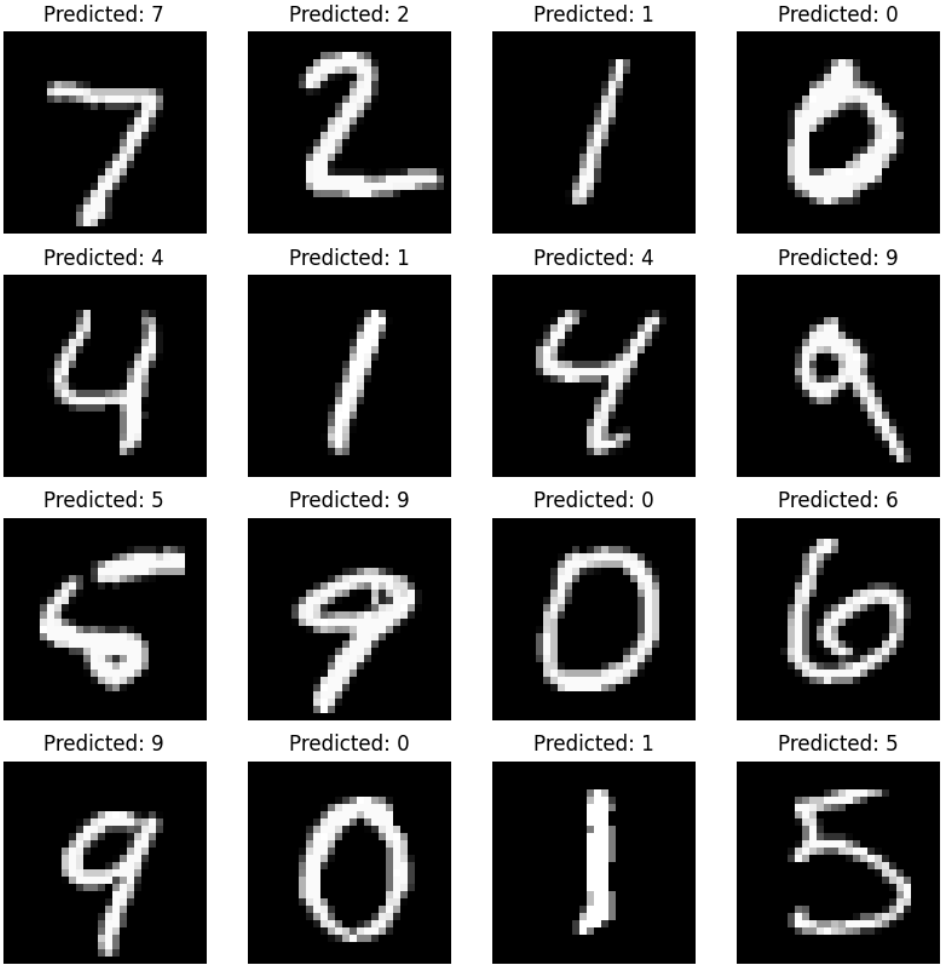


2. 测试集图像



3. 预测结果

如图所示，每张图片的上方都显示了模型输出的预测结果，可以看到预测结果都正确。



5.2 结果分析

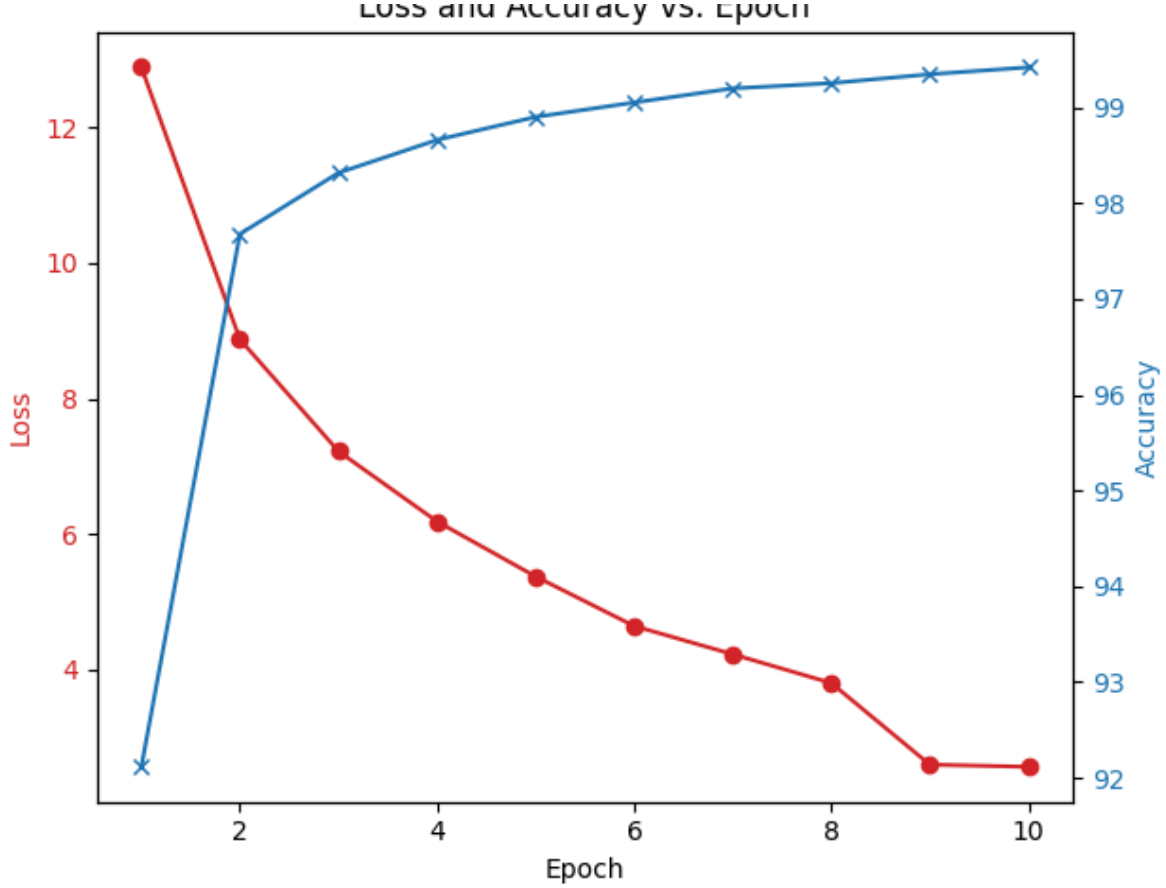
5.2.1 训练结果分析

整个训练过程中共训练了 10 轮，每轮训练的准确率如下：

- Epoch 1: Loss: 12.893, Accuracy: 92.115
- Epoch 2: Loss: 8.872, Accuracy: 97.680
- Epoch 3: Loss: 7.219, Accuracy: 98.318
- Epoch 4: Loss: 6.182, Accuracy: 98.662
- Epoch 5: Loss: 5.369, Accuracy: 98.902
- Epoch 6: Loss: 4.637, Accuracy: 99.053
- Epoch 7: Loss: 4.220, Accuracy: 99.200

- Epoch 8: Loss: 3.795, Accuracy: 99.255
- Epoch 9: Loss: 2.589, Accuracy: 99.348
- Epoch 10: Loss: 2.557, Accuracy: 99.420

可以看到，训练过程中准确率一直在上升，最终达到了 99.420%。



从训练结果来看，模型的表现呈现以下几个特点：

1. **损失值持续下降**：从第一个轮次的 12.893 到第十个轮次的 2.557，训练损失值在每个轮次都稳步下降。这表明模型在训练过程中有效地学习了数据特征，并且优化算法（Adam）在指导模型参数向着最小化损失函数的方向更新。尤其是在早期轮次，损失下降幅度较大，后期逐渐减缓，这是模型收敛的典型现象。
2. **准确率稳步提升**：与损失值下降相对应，训练准确率从第一个轮次的 92.115% 持续上升至第十个轮次的 99.420%。这直接反映了模型识别训练集样本能力的增强。准确率的提升同样在早期更为显著，后期增幅变小，说明模型逐渐学习到了数据中的细微差别。
3. **学习效率高**：模型在第一个轮次结束后，准确率就达到了 92.115%，说明 LeNet-5 架构对于 MNIST 这类图像分类任务具有良好的适应性，并且所选的超参数（如学习率）设置得当，使得模型能够快速入门。
4. **收敛趋势明显**：观察最后几个轮次，例如从第 8 轮到第 10 轮，损失值的下降幅度（3.795 -> 2.589 -> 2.557）和准确率的提升幅度（99.255% -> 99.348% -> 99.420%）都在

逐渐减小。这表明模型在训练集上的性能已接近饱和，进一步训练可能带来的提升有限，或者需要调整学习率等策略进行更精细的调优。

5. **训练效果优秀**：最终达到 99.420% 的训练准确率是一个非常高的水平，说明模型充分学习了训练数据的模式。结合测试集准确率（如后文分析，为 98.940%），两者差异不大，表明模型具有较好的泛化能力，没有出现严重的过拟合现象。

这些观察结果共同表明，本次 LeNet-5 模型的训练过程是健康且有效的。模型不仅快速学习了数据的基本特征，也在后续的训练中逐步优化，达到了较高的识别精度。

5.2.2 测试结果分析

在 MNIST 的测试集上运行测试，对于 10000 张测试图片，模型预测准确率为 98.940%。这一结果表明模型具有优秀的泛化能力，能够在未见过的数据上做出准确的预测。具体分析如下：

- **高准确率与泛化性**：98.940% 的准确率在 MNIST 这类基准数据集上是一个非常出色的表现。它证明了 LeNet-5 模型架构的有效性，以及通过训练学习到的特征能够很好地推广到测试集样本。
- **与训练准确率的比较**：训练集上的最终准确率为 99.420%。测试准确率（98.940%）与训练准确率之间存在约 0.48 个百分点的微小差距。这个差距很小，表明模型没有出现明显的过拟合现象。模型在学习训练数据特征的同时，也保持了对新数据的识别能力。
- **结果的可靠性**：MNIST 测试集包含 10000 张图片，覆盖了各种手写风格。在此规模的测试集上达到近 99% 的准确率，说明模型的性能是稳定和可靠的。从提供的 **predictions.png** 图像中也可以看到，模型对不同数字的预测均表现良好。
- **符合预期**：对于 LeNet-5 这样的经典卷积神经网络，在 MNIST 数据集上达到 98-99% 的准确率是符合文献和实践中的普遍预期的。这进一步验证了实验设置的合理性和模型训练的成功。

5.2.3 模型改进可能

尽管当前模型在 MNIST 数据集上已取得优异表现，但仍有一些潜在的改进方向可以探索，以期进一步提升模型性能或鲁棒性：

1. **数据增强 (Data Augmentation)**：
   - 虽然 MNIST 数据集相对简单，但引入轻微的数据增强技术，如随机旋转、平移、缩放或弹性形变，可以增加训练数据的多样性，有助于提高模型的泛化能力，使其对未见过的、略有变化的手写数字更具鲁棒性。
2. **优化器与学习率策略调整**：
   - 尝试不同的优化器：除了 Adam，可以试验 SGD 配合动量 (Momentum)、Adagrad、RMSprop 等其他优化算法。
   - 学习率调度：使用学习率衰减策略，如在训练过程中逐步降低学习率（例如，每隔几个 epoch 按一定比例衰减，或使用余弦退火等），有助于模型在训练后期更精细地调整权重，可能收敛到更好的局部最优解。

3. **网络结构调整**：
   - **批量归一化 (Batch Normalization)**：在卷积层或全连接层之后、激活函数之前加入批量归一化层。这有助于加速训练收敛，稳定训练过程，并可能允许使用更高的学习率，同时具有一定的正则化效果。
   - **Dropout 正则化**：在全连接层之间加入 Dropout 层，通过在训练时随机失活一部分神经元，减少神经元之间的协同作用，从而防止过拟合，提高模型的泛化能力。
   - **池化方式**：将平均池化 (AvgPool2d) 替换为最大池化 (MaxPool2d)。最大池化通常能更好地保留图像的显著特征，在许多图像分类任务中表现更优。
   - **激活函数**：虽然 ReLU 在本模型中表现良好，但可以尝试其他激活函数，如 LeakyReLU、ELU 等，观察是否对性能有改善。
   - **增加网络深度或宽度**：适度增加卷积层的数量或每层卷积核的数量，或者增加全连接层的神经元数量，可能会提升模型的表达能力，但需注意过拟合风险和计算成本的增加。
4. **超参数系统调优**：
   - 对学习率、批处理大小 (batch size)、训练轮次 (epochs)、Dropout 率、优化器特定参数（如 Adam 的 beta 值）等关键超参数进行更系统的搜索和优化，例如使用网格搜索 (Grid Search)、随机搜索 (Random Search) 或更高级的贝叶斯优化等方法。
5. **集成学习 (Ensemble Learning)**：
   - 训练多个独立的 LeNet-5 模型（例如，使用不同的随机初始化种子或略微不同的超参数配置），然后将它们的预测结果进行集成（如投票或平均），通常可以获得比单个模型更稳定和更优的性能。

实施这些改进可能需要更多的实验和计算资源，但它们为进一步提升模型在 MNIST 乃至更复杂图像识别任务上的表现提供了有价值的探索方向。

6 结论与心得体会

6.1 实验结论

通过本次基于 Pytorch 的 MNIST 手写数字识别实验，我成功构建并训练了 LeNet-5 卷积神经网络模型。实验结果表明，该模型能够有效地从 MNIST 数据集中学习手写数字的特征，并在测试集上达到了令人满意的识别准确率。这验证了卷积神经网络在图像识别领域的强大能力，以及 Pytorch 框架在构建和训练深度学习模型方面的高效性和便捷性。

实验不仅加深了我对 CNN 各组成部分（如卷积层、池化层、全连接层）功能的理解，也让我直观地看到了数据预处理、模型训练、参数优化和模型评估在整个机器学习项目中的重要性。

6.2 心得体会

在本次实验过程中，我深刻体会到理论知识与编程实践相结合的重要性。最初接触 LeNet-5 模型时，其层级结构和参数传递方式对我来说还比较抽象。然而，通过亲手编写每一层网络的代码，定义前向传播路径，并观察数据在网络中维度的变化，这些概念逐

渐变得清晰和具体。调试过程中遇到的各种问题，例如数据维度不匹配、损失函数不收敛等，都促使我更深入地研究 Pytorch 的 API 文档和相关深度学习原理，这个过程极大地锻炼了我的问题解决能力和独立思考能力。这种从无到有的创造过程带来了巨大的成就感，也激发了我对深度学习领域更浓厚的探索兴趣。

6.3 实验收获

本次实验使我在多个方面都获得了显著的技术提升和经验积累。

首先，我熟练掌握了使用 Pytorch 框架搭建、训练和评估卷积神经网络的全过程，包括数据集的加载与预处理（如`ToTensor`，`Normalize`）、模型结构的定义（继承`nn.Module`，使用`nn.Conv2d`，`nn.AvgPool2d`，`nn.Linear`，`nn.ReLU`等模块）、损失函数的选择（`CrossEntropyLoss`）以及优化器的使用（`Adam`）。

同时，我对深度学习中的核心概念，如卷积操作、池化操作、激活函数、反向传播和梯度下降等，有了更直观和深入的理解。此外，实验还提升了我的代码调试能力和实验结果分析能力，学会了如何通过观察训练过程中的损失和准确率曲线来判断模型状态，以及如何通过可视化手段（如展示样本图像和预测结果）来评估模型性能。这些经验对于未来进行更复杂的机器学习项目打下了坚实的基础。

7 参考文献

- PyTorch 官方网站：https://pytorch.org
- PyTorch 官方文档：https://pytorch.org/docs/stable/index.html
- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.
- MNIST handwritten digit database: http://yann.lecun.com/exdb/mnist/