

手搓神经网络模型

本项目将展示如何利用 PyTorch 从零开始构建一个神经网络模型，以解决手写字母（实际上这里使用的是 MNIST 手写数字）识别问题。本文内容不仅介绍了神经网络的基础知识、训练本质和卷积操作的优势，还给出了完整代码实现，让你能够亲自动手构建并训练模型。

神经网络模型基础

在开始代码实现之前，我们先介绍一些神经网络的基本概念和原理。

什么是神经网络？

神经网络是一种受人脑神经元连接方式启发而构造的数学模型。它由大量节点（神经元）构成，这些节点以不同的层级进行排列：

- **输入层**：接收外界数据。
- **隐藏层**：对数据进行特征提取与变换。
- **输出层**：给出最终的预测结果。

下载 MNIST 数据集

在下面的代码中，我们将利用 `torchvision` 自动下载 `MNIST` 数据集。这个数据集包含手写数字图像，是机器学习领域的经典数据集。

这段代码主要是准备数据用的。`MNIST` 是一个手写数字的数据集，里面有很多 `28x28` 的灰度图片。我们先用 `transforms` 把图片转成模型能用的格式（张量），然后标准化一下，让数据更适合训练。之后用 `DataLoader` 把数据分成小份（`batch`），训练时每次处理 `64` 张，测试时处理 `1000` 张。`shuffle=True` 是为了让训练数据随机排列，这样模型不会“偷懒”只记住顺序。

```
# 导入必要的库
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import torch.nn.functional as F
```

```
# 定义图像预处理流程
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# 下载并加载数据集
train_dataset = datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform
)
```

```

test_dataset = datasets.MNIST(
    root='./data',
    train=False,
    download=True,
    transform=transform
)

# 创建数据加载器
train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset,
    batch_size=64,
    shuffle=True
)

test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=1000,
    shuffle=False
)

```

Downloading <http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz>
 Downloading
<http://211.69.138.66/cache/11/02/yann.lecun.com/34152be34c1115808d82493371362afe/train-images-idx3-ubyte.gz> to ./data\MNIST\raw\train-images-idx3-ubyte.gz


100%|


 | 9.91M/9.91M [00:01<00:00, 8.36MB/s]

Extracting ./data\MNIST\raw\train-images-idx3-ubyte.gz to ./data\MNIST\raw

Downloading <http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz>
 Downloading
<http://211.69.138.66/cache/3/02/yann.lecun.com/32a667c2f3de678798ae039d04726fa1/train-labels-idx1-ubyte.gz> to ./data\MNIST\raw\train-labels-idx1-ubyte.gz

100%|


 | 28.9k/28.9k [00:00<00:00, 1.10MB/s]

Extracting ./data\MNIST\raw\train-labels-idx1-ubyte.gz to ./data\MNIST\raw

Downloading <http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz>

Failed to download (trying next):
HTTP Error 404: Not Found

```
Downloading https://oss-ci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://oss-ci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to ./data\MNIST\raw\t10k-images-idx3-ubyte.gz
```

```
100%|██████████████████████████████████████████████████████████████████|  
██████████| 1.65M/1.65M [00:02<00:00, 703kB/s]
```

```
Extracting ./data\MNIST\raw\t10k-images-idx3-ubyte.gz to ./data\MNIST\raw
```

```
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found
```

```
Downloading https://oss-ci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://oss-ci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to ./data\MNIST\raw\t10k-labels-idx1-ubyte.gz
```

100%|
███████████
██████████ | 4.54k/4.54k [00:00<00:00, 4.38MB/s]

```
Extracting ./data\MNIST\raw\t10k-labels-idx1-ubyte.gz to ./data\MNIST\raw
```

这里是用 matplotlib 把 MNIST 数据集的图片画出来看看。训练集的前 21 张用红色标签标数字，测试集的前 7 张用绿色标签标数字。squeeze() 是去掉图片数据里没用的维度，cmap="gray" 是让图片显示成灰度图。看到这些图片和标签后，我能直观地知道数据集长什么样，挺有意思的！

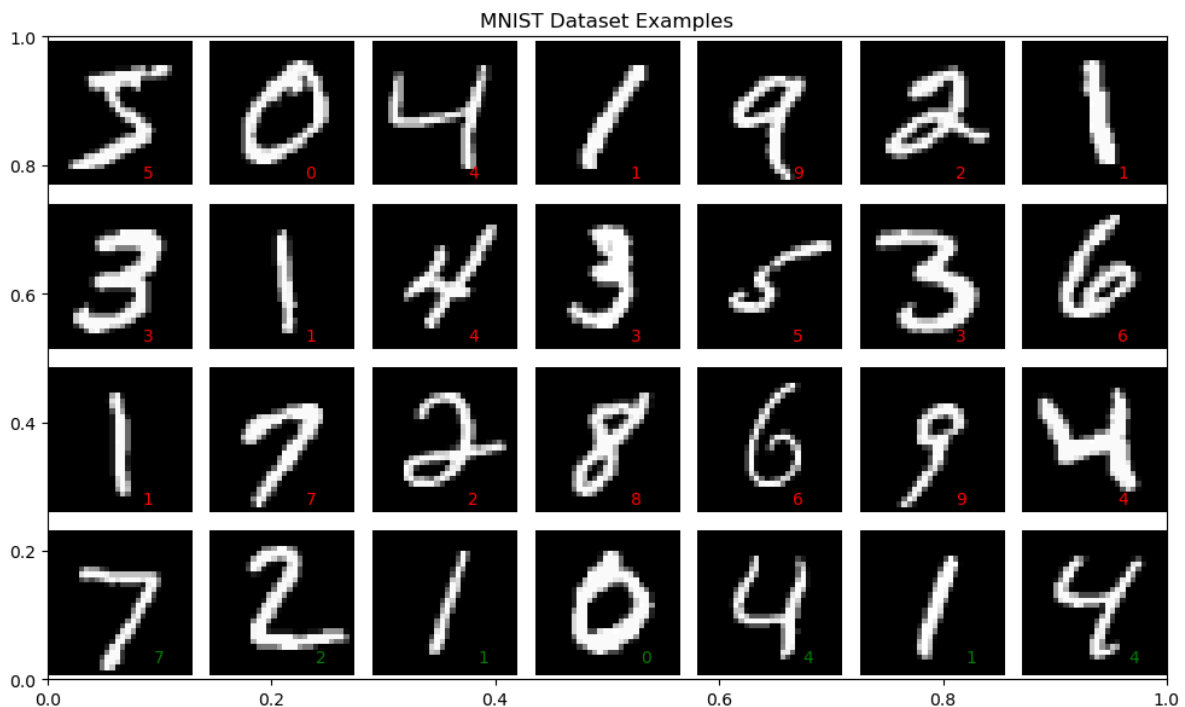
```
plt.figure(figsize=(10, 6))

# 绘制训练集图像
plt.title("MNIST Dataset Examples")
for i in range(21): # 绘制 21 张训练集图像
    plt.subplot(4, 7, i+1) # 绘制第 i+1 张图像
    plt.axis("off") # 不显示坐标轴
    img = train_dataset[i][0].squeeze() # 获取第 i 张图像
    label = train_dataset[i][1] # 获取第 i 张图像的标签
    plt.imshow(img, cmap="gray") # 绘制第 i 张图像
    plt.text(18, 26, f"{label}", fontsize=10, color="red") # 在图像右下角标出红色标签

# 绘制测试集图像
for i in range(7): # 绘制 7 张测试集图像
    plt.subplot(4, 7, i+22) # 绘制第 i+22 张图像
    plt.axis("off") # 不显示坐标轴
    img = test_dataset[i][0].squeeze() # 获取第 i 张图像
    label = test_dataset[i][1] # 获取第 i 张图像的标签
    plt.imshow(img, cmap="gray") # 绘制第 i 张图像
```

```
plt.text(20, 25, f"{label}", fontsize=10, color="green") # 在图像右下角标出绿色标签

plt.tight_layout()
plt.show()
```



这段代码定义了一个叫 LeNet 的神经网络模型。模型有两部分：卷积层和全连接层。卷积层用 Conv2d 提取图片特征，池化层用 MaxPool2d 缩小特征图大小。全连接层用 Linear 把特征变成 10 个数字的分类结果。forward 函数是数据怎么一步步通过这些层的，relu 是激活函数，让模型能学到更复杂的模式。看到 1644，我算了一下，应该是图片经过两次卷积和池化后的大小。

```
# 定义 LeNet 神经网络模型类，继承自 nn.Module
class LeNet(nn.Module):
    def __init__(self):
        # 初始化父类 nn.Module
        super(LeNet, self).__init__()

        # 第一个卷积层：
        # 输入通道：1（灰度图像），输出通道：6，卷积核大小：5x5
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)

        # 定义池化层：
        # 使用 2x2 的最大池化，能够减小特征图的尺寸
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # 第二个卷积层：
        # 输入通道：6，输出通道：16，卷积核大小：5x5
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5)

        # 第一个全连接层：
        # 输入特征数为 16*4*4（经过两次卷积和池化后的特征图尺寸），输出特征数为 120
        self.fc1 = nn.Linear(in_features=16*4*4, out_features=120)

        # 第二个全连接层：将 120 个特征映射到 84 个特征
```

```

self.fc2 = nn.Linear(in_features=120, out_features=84)

# 第三个全连接层：输出 10 个类别，对应 MNIST 中 10 个数字
self.fc3 = nn.Linear(in_features=84, out_features=10)

def forward(self, x):
    # 将输入通过第一个卷积层，并使用 ReLU 激活函数增加非线性
    x = torch.relu(self.conv1(x))
    # 应用池化层，减小特征图尺寸
    x = self.pool(x)
    # 第二个卷积层 + ReLU 激活
    x = torch.relu(self.conv2(x))
    # 再次池化
    x = self.pool(x)
    # 将多维特征图展平为一维向量，为全连接层做准备
    x = x.view(-1, 16*4*4)
    # 第一个全连接层 + ReLU 激活
    x = torch.relu(self.fc1(x))
    # 第二个全连接层 + ReLU 激活
    x = torch.relu(self.fc2(x))
    # 第三个全连接层得到最终输出（未经过激活，后续会结合损失函数使用）
    x = self.fc3(x)
    return x

```

****代码说明：****

本部分代码定义了 **LeNet** 模型。通过两个卷积层和池化层逐步提取图像特征，再通过全连接层进行分类。注意，由于 **MNIST** 图像尺寸为 **28x28**，经过两次卷积和池化后，特征图尺寸正好为 **4x4**（通道数为 **16**），因此全连接层的输入特征数为 **`16*4*4`**。

LeNet 模型结构图

###初始化一个 **LeNet** 模型，并打印其结构。

```

# 打印模型结构
model = LeNet()
print(model)

```

```

LeNet(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceiling_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=256, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)

```

让我们详细解释一下模型的每一层结构：

1. ****第一个卷积层 `(conv1)`****：

- 输入: 1 个通道 (灰度图像)
- 输出: 6 个特征图
- 卷积核: 5×5
- 步长: 1
- 输入尺寸: 28×28 → 输出尺寸: 24×24

2. **第一个池化层 (pool):**

- 池化窗口: 2×2
- 步长: 2
- 输入尺寸: 24×24 → 输出尺寸: 12×12

3. **第二个卷积层 (conv2):**

- 输入: 6 个通道
- 输出: 16 个特征图
- 卷积核: 5×5
- 步长: 1
- 输入尺寸: 12×12 → 输出尺寸: 8×8

4. **第二个池化层 (pool):**

- 池化窗口: 2×2
- 步长: 2
- 输入尺寸: 8×8 → 输出尺寸: 4×4

5. **第一个全连接层 (fc1):**

- 输入: 256 个特征 ($16 \times 4 \times 4$)
- 输出: 120 个神经元

6. **第二个全连接层 (fc2):**

- 输入: 120 个特征
- 输出: 84 个神经元

7. **第三个全连接层 (fc3):**

- 输入: 84 个特征
- 输出: 10 个神经元 (对应 10 个数字类别)

数据流向说明:

1. 输入的 28×28 图像首先经过第一个卷积层, 生成 6 个 24×24 的特征图
2. 经过池化层后, 特征图变为 6 个 12×12
3. 第二个卷积层将特征图转换为 16 个 8×8 的特征图
4. 再次池化后, 得到 16 个 4×4 的特征图
5. 将特征图展平为一维向量 ($16 \times 4 \times 4 = 256$)
6. 通过三个全连接层逐步将特征降维, 最终输出 10 个类别的概率分布

这种结构设计使得网络能够逐层提取图像的特征, 从低级的边缘特征到高级的抽象特征, 最终实现手写数字的分类。

模型训练和评估

接下来, 我们将编写训练和测试的代码, 并整合到主函数中, 实现对模型的训练和评估。

定义训练函数

###训练函数中，模型对每个批次数据进行前向传播，计算损失后进行反向传播，并使用优化器更新权重。每隔一定批次输出当前损失，方便观察训练进度。

这个函数是训练模型的核心。每次拿一批数据（**data** 是图片，**target** 是标签），通过模型算出预测结果（**output**），然后用损失函数（**criterion**）看看预测和真实标签差多少。**loss.backward()** 是算梯度，告诉模型哪里错了，**optimizer.step()** 是根据梯度调整参数。训练时还记录了损失和准确率，方便我们知道模型学得怎么样。打印的部分让我能看到训练进度，感觉挺直观的。

定义训练函数，用于在训练集上训练模型

```
def train(model, device, train_loader, optimizer, criterion, epoch):
    model.train()
    train_loss = 0
    correct = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        # 累计损失和正确预测数
        train_loss += loss.item() * data.size(0)
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

        if batch_idx % 5000 == 0:
            print(f"Train Epoch: {epoch} [{batch_idx *
len(data)} / {len(train_loader.dataset)}] \t Loss: {loss.item():.6f}")

    # 计算平均损失和准确率
    train_loss /= len(train_loader.dataset)
    accuracy = 100. * correct / len(train_loader.dataset)
    return train_loss, accuracy
```

函数原理分析

LeNet训练函数的核心流程可分为前向传播、损失计算、反向传播、参数更新四个阶段。以下是代码各环节与CNN训练原理的对应关系：

- 前向传播阶段
 - `model(data)` 执行卷积神经网络的前向计算
 - LeNet结构依次执行：卷积→池化→卷积→池化→全连接→全连接
 - 卷积层通过滤波器提取空间特征，池化层降低特征图维度，全连接层完成分类
- 损失计算阶段
 - `criterion(output, target)` 使用交叉熵损失函数
 - 该损失函数适用于多分类任务，衡量预测概率分布与真实标签的差异
 - 损失值反映当前参数下模型的预测误差程度
- 反向传播阶段
 - `loss.backward()` 自动计算梯度
 - 通过链式法则逐层计算卷积核参数和全连接层权重的梯度
 - 梯度值表征各参数对最终损失的贡献程度

- 参数优化阶段
 - `optimizer.step()` 根据梯度更新参数
 - 典型优化器如SGD的更新公式: $w_{t+1} = w_t - \eta \nabla L(w_t)$
 - 学习率 η 控制参数更新步长, 需合理设置避免震荡或收敛过慢

关键实现细节

-梯度管理

- `optimizer.zero_grad()` 在每次迭代前清零梯度, 防止梯度累积
- PyTorch默认会累加梯度, 手动清零确保每次更新基于当前批次数据
- 设备迁移
 - `data.to(device)` 将数据转移到GPU/CPU
 - 利用GPU并行计算加速卷积运算, 这对大规模数据训练至关重要
- 训练监控
 - 每5000批次输出进度信息, 帮助监控训练过程
 - 累计损失计算需乘以 `data.size(0)`, 因PyTorch损失默认返回批次平均值
 - 准确率计算通过比较预测最大值索引与真实标签实现

LeNet训练特点

- 特征学习机制
 - 通过交替的卷积和池化操作, 网络自动学习层次化特征
 - 浅层卷积捕捉边缘等低级特征, 深层卷积提取复杂模式
- 参数优化策略
 - 卷积核参数通过梯度下降自动优化
 - 权重初始化通常采用Xavier或He方法, 保证训练稳定性
- 泛化能力提升
 - 池化操作增强平移不变性
 - 后续改进版本可加入Dropout层防止过拟合
- 训练效果评估
 - 最终返回epoch平均损失和准确率
 - 这些指标用于跟踪模型在训练集上的学习进度
 - 需配合验证集评估真实泛化能力

该训练函数实现了标准监督学习流程, 通过多次epoch迭代不断优化网络参数, 使模型逐步提升特征提取和分类能力。实际应用中还需配合验证集监控、学习率调整等策略以获得最佳效果。

测试函数和训练差不多, 但这里不用更新参数, 所以用了 `torch.no_grad()` 来省内存。模型在测试集上跑一遍, 算出损失和准确率, 看看它在没见过的数据上表现如何。`model.eval()` 是告诉模型现在是测试, 别用训练时的特殊操作 (比如 dropout)。这个函数让我知道模型是不是真的学会了, 而不是只记住了训练数据。

```
### 定义测试函数
def test(model, device, test_loader, criterion):
    model.eval() # 将模型设置为评估模式, 关闭 dropout 等训练特性
```



```

test_loss = 0 # 初始化测试损失
correct = 0 # 初始化预测正确的样本计数
# 在测试阶段不计算梯度，节省内存和加快计算速度
with torch.no_grad(): # 不计算梯度，节省内存和加快计算速度
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        output = model(data)
        test_loss += criterion(output, target).item() * data.size(0)
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset) # 计算平均损失
accuracy = 100. * correct / len(test_loader.dataset) # 计算准确率
return test_loss, accuracy

```

代码说明：

测试函数中，模型在测试集上进行前向传播，并累计计算总体损失与正确预测数量，最终输出平均损失及准确率，以评估模型的泛化能力。

训练过程中的损失指标

在训练神经网络时，我们主要关注两个重要的损失指标：

1. 训练损失 (Training Loss)：

- 表示模型在训练数据集上的预测误差
- 反映了模型对训练数据的拟合程度
- 训练损失持续下降表明模型正在学习数据中的模式
- 但过低的训练损失可能意味着过拟合

2. 测试损失 (Test Loss)：

- 表示模型在从未见过的测试数据上的预测误差
- 反映了模型的泛化能力
- 测试损失应该与训练损失保持相近
- 如果测试损失明显高于训练损失，说明模型可能过拟合

理想的训练过程应该表现为：

- 训练损失和测试损失同时下降
- 两者之间保持较小的差距
- 最终都收敛到一个较低的水平

如果观察到以下情况，则需要调整模型或训练策略：

- 训练损失持续下降但测试损失上升：过拟合的典型特征
- 两种损失都居高不下：欠拟合，可能需要增加模型复杂度
- 损失剧烈波动：学习率可能过大
-

```
# 主函数，整合数据加载、模型构建、训练和测试过程
# 检查是否有 GPU 可用，否则使用 CPU
device = torch.device("cuda" if torch.cuda.is_available() else "mps" if
torch.mps.is_available() else "cpu")

# 实例化 LeNet 模型，并移动到指定设备上
model = LeNet().to(device)
```

优化器与损失函数

在神经网络训练中，优化器和损失函数是两个核心组件：

1. 随机梯度下降优化器 (SGD)：

- **原理：**通过计算损失函数对模型参数的梯度，沿着梯度的反方向更新参数
- **学习率：**控制每次参数更新的步长（这里设为 0.01）
- **动量：**
 - 作用：累积之前的梯度方向，帮助模型跳出局部最小值
 - 数值：这里设为 0.9，表示保留 90% 的历史梯度信息
 - 优势：加速收敛，减少震荡

2. 交叉熵损失函数 (CrossEntropyLoss)：

- **适用场景：**多分类问题（如本例中的 10 个数字分类）
- **计算过程：**
 - 首先对模型输出进行 softmax 归一化，得到每个类别的概率
 - 然后计算预测概率分布与真实标签分布的交叉熵
- **特点：**
 - 能有效处理多分类问题
 - 对错误预测施加更大的惩罚
 - 输出值在 $[0, \infty)$ 范围内，0 表示完美预测

```
# 定义优化器：使用随机梯度下降 (SGD)，学习率为 0.01，动量为 0.9
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# 定义损失函数：交叉熵损失函数常用于分类问题
criterion = nn.CrossEntropyLoss()
```

```
# 用于记录训练过程的指标
### 开始训练
train_losses = []
train_accs = []
test_losses = []
test_accs = []

epochs = 5 # 设定训练轮数为 5
# 循环训练和测试模型
for epoch in range(1, epochs + 1):
    # 训练并记录指标
```

```

train_loss, train_acc = train(model, device, train_loader, optimizer,
criterion, epoch)
test_loss, test_acc = test(model, device, test_loader, criterion)

# 保存指标
train_losses.append(train_loss)
train_accs.append(train_acc)
test_losses.append(test_loss)
test_accs.append(test_acc)

print(f"\nEpoch {epoch}:")
print(f"Train - Loss: {train_loss:.4f}, Accuracy: {train_acc:.2f}%")
print(f"Test - Loss: {test_loss:.4f}, Accuracy: {test_acc:.2f}%\n")

```

Train Epoch: 1 [0/60000] Loss: 2.302972

Epoch 1:

Train - Loss: 0.2866, Accuracy: 90.84%

Test - Loss: 0.0733, Accuracy: 97.71%

Train Epoch: 2 [0/60000] Loss: 0.042187

Epoch 2:

Train - Loss: 0.0654, Accuracy: 97.96%

Test - Loss: 0.0432, Accuracy: 98.63%

Train Epoch: 3 [0/60000] Loss: 0.027652

Epoch 3:

Train - Loss: 0.0481, Accuracy: 98.47%

Test - Loss: 0.0388, Accuracy: 98.78%

Train Epoch: 4 [0/60000] Loss: 0.011715

Epoch 4:

Train - Loss: 0.0369, Accuracy: 98.86%

Test - Loss: 0.0334, Accuracy: 98.89%

Train Epoch: 5 [0/60000] Loss: 0.008617

Epoch 5:

Train - Loss: 0.0302, Accuracy: 99.03%

Test - Loss: 0.0505, Accuracy: 98.24%

绘制训练过程图表

```
epochs_range = range(1, epochs + 1)
```

```
plt.figure(figsize=(12, 5))
```

绘制损失曲线

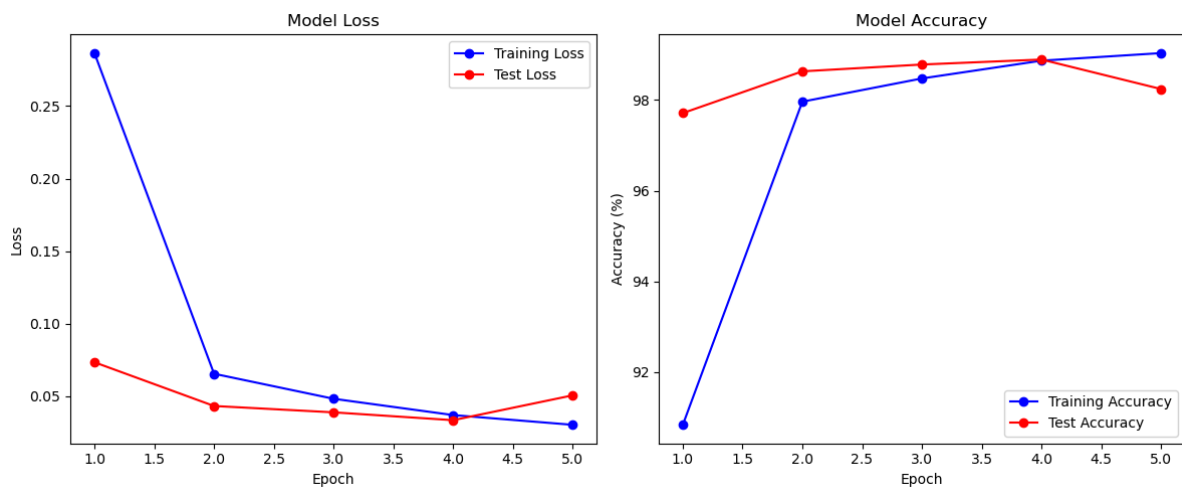
```
plt.subplot(1, 2, 1)
```

```
plt.plot(epochs_range, train_losses, 'bo-', label='Training Loss')
```

```
plt.plot(epochs_range, test_losses, 'ro-', label='Test Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# 绘制准确率曲线
plt.subplot(1, 2, 2)
plt.plot(epochs_range, train_accs, 'bo-', label='Training Accuracy')
plt.plot(epochs_range, test_accs, 'ro-', label='Test Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()

plt.tight_layout()
plt.show()
```



```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import convolve2d

# 原始矩阵
matrix = test_dataset[0][0].squeeze()

# 3x3 卷积核
kernel = np.array([[ -1, -1, -1],
                    [ 0, 0, 0],
                    [ 1, 1, 1]])

# 进行卷积运算
convolved = convolve2d(matrix, kernel, mode='valid')

# 计算子图尺寸比例
original_shape = matrix.shape
convolved_shape = convolved.shape
```

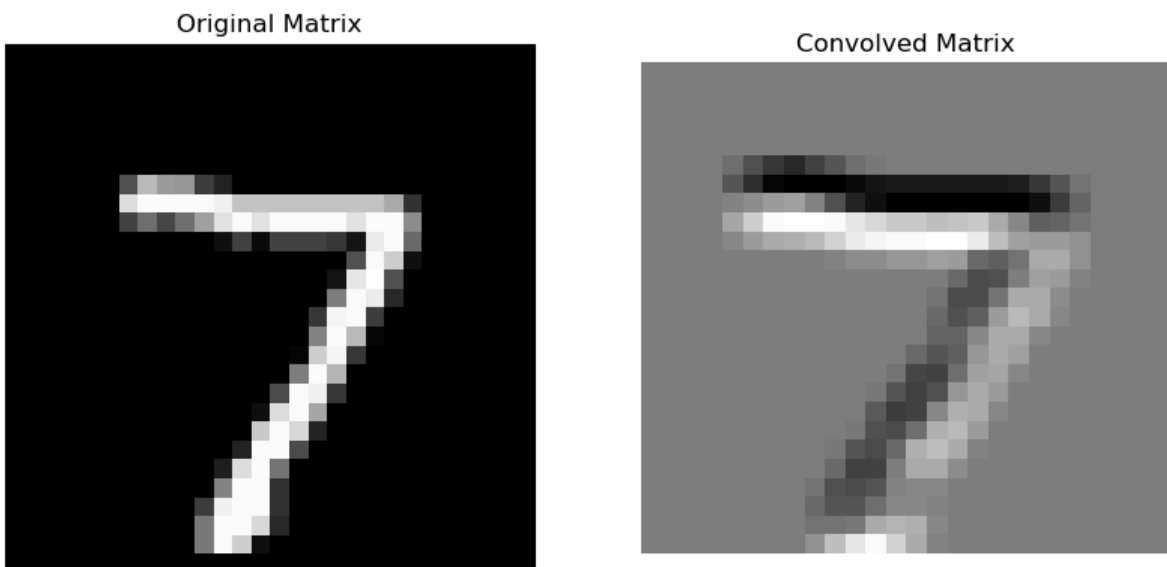
```
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

# 计算比例因子，使卷积后的小图与原图比例协调
scale_factor = original_shape[0] / convolved_shape[0]

# 调整原始矩阵子图
axes[0].imshow(matrix, cmap='gray', interpolation='nearest', aspect=1)
axes[0].set_title("Original Matrix")
axes[0].axis("off")

# 调整卷积后矩阵子图，缩放至与原图比例协调
axes[1].imshow(convolved, cmap='gray', interpolation='nearest',
aspect=1/scale_factor)
axes[1].set_title("Convolved Matrix")
axes[1].axis("off")

plt.show()
```



代码说明：

在主函数中，我们首先检测计算设备，然后实例化模型、定义优化器和损失函数，并依次调用训练和测试函数。每个 epoch 结束后，终端会输出当前的训练状态和测试结果。

详解神经元的训练过程

下面以一个最简单的神经网络——只有一个神经元的单层模型——为例，展示训练过程中神经元参数（权重和偏置）是如何一步步确定下来的。这个例子帮助理解神经网络的基本训练流程，包括**前向传播**、**损失计算**、**反向传播**（梯度计算）和**参数更新**。

网络结构与设定

假设我们的神经网络只有一个神经元，该神经元接收一个输入 x 并输出 y 。神经元具有两个可训练参数：

- 权重 w
- 偏置 b

采用**线性激活函数**（即不做非线性变换），则神经元的输出为：

$$y = w \cdot x + b.$$

同时，设定一个**平方误差损失函数**（Mean Squared Error, MSE）来衡量输出与目标之间的差距：

$$L = \frac{1}{2}(y - y_{\text{target}})^2,$$

其中 y_{target} 为给定的目标输出。

训练流程概述

整个训练过程可以分为以下几个步骤：

1. 初始化参数

随机或按照某种策略给定初始的 w 和 b 。

2. 前向传播

给定输入 x ，计算神经元输出：

$$y = w \cdot x + b.$$

3. 损失计算

根据神经元输出和目标输出 y_{target} 计算损失：

$$L = \frac{1}{2}(y - y_{\text{target}})^2.$$

4. 反向传播（梯度计算）

利用链式法则计算损失关于参数 w 和 b 的梯度，具体如下：

- 对 y 求导：

$$\frac{\partial L}{\partial y} = y - y_{\text{target}}.$$

- 由于 $y = w \cdot x + b$ ，有：

$$\frac{\partial y}{\partial w} = x, \quad \frac{\partial y}{\partial b} = 1.$$

- 所以利用链式法则：

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w} = (y - y_{\text{target}}) \cdot x,$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial b} = y - y_{\text{target}}.$$

5. 参数更新

利用梯度下降法调整参数：

$$w_{\text{new}} = w - \eta \cdot \frac{\partial L}{\partial w}, \quad b_{\text{new}} = b - \eta \cdot \frac{\partial L}{\partial b},$$

其中 η 为学习率，控制每次更新的步长。

6. 重复迭代

重复步骤2~5，直至损失足够小或达到预定的迭代次数。

数值示例

假设我们有以下设定：

- 输入：\$x = 1.0\$
- 目标输出：\$y_{\text{target}} = 2.0\$
- 初始参数：\$w = 0.5\$, \$b = 0.1\$
- 学习率：\$\eta = 0.1\$

我们来看几次迭代的具体计算过程。

迭代 1

1. 前向传播

计算输出：

$$y = 0.5 \times 1.0 + 0.1 = 0.6.$$

2. 损失计算

$$L = \frac{1}{2}(0.6 - 2.0)^2 = \frac{1}{2} \times (-1.4)^2 = \frac{1}{2} \times 1.96 = 0.98.$$

3. 反向传播 (梯度计算)

◦ 首先计算：

$$\frac{\partial L}{\partial y} = 0.6 - 2.0 = -1.4.$$

◦ 然后：

$$\frac{\partial L}{\partial w} = -1.4 \times 1.0 = -1.4,$$

$$\frac{\partial L}{\partial b} = -1.4.$$

4. 参数更新

$$w_{\text{new}} = 0.5 - 0.1 \times (-1.4) = 0.5 + 0.14 = 0.64,$$

$$b_{\text{new}} = 0.1 - 0.1 \times (-1.4) = 0.1 + 0.14 = 0.24.$$

迭代 2

使用更新后的参数 \$w = 0.64\$ 和 \$b = 0.24\$。

1. 前向传播

$$y = 0.64 \times 1.0 + 0.24 = 0.88.$$

2. 损失计算

$$L = \frac{1}{2}(0.88 - 2.0)^2 = \frac{1}{2} \times (-1.12)^2 = \frac{1}{2} \times 1.2544 \approx 0.6272.$$

3. 反向传播

$$\frac{\partial L}{\partial y} = 0.88 - 2.0 = -1.12,$$

$$\frac{\partial L}{\partial w} = -1.12 \times 1.0 = -1.12,$$

$$\frac{\partial L}{\partial b} = -1.12.$$

4. 参数更新

$$w_{\text{new}} = 0.64 - 0.1 \times (-1.12) = 0.64 + 0.112 = 0.752,$$

$$b_{\text{new}} = 0.24 - 0.1 \times (-1.12) = 0.24 + 0.112 = 0.352.$$

迭代 3

使用更新后的参数 $w = 0.752$ 和 $b = 0.352$ 。

1. 前向传播

$$y = 0.752 \times 1.0 + 0.352 = 1.104.$$

2. 损失计算

$$L = \frac{1}{2}(1.104 - 2.0)^2 = \frac{1}{2} \times (-0.896)^2 \approx \frac{1}{2} \times 0.802 = 0.401.$$

3. 反向传播

$$\frac{\partial L}{\partial y} = 1.104 - 2.0 = -0.896,$$

$$\frac{\partial L}{\partial w} = -0.896 \times 1.0 = -0.896,$$

$$\frac{\partial L}{\partial b} = -0.896.$$

4. 参数更新

$$w_{\text{new}} = 0.752 - 0.1 \times (-0.896) = 0.752 + 0.0896 \approx 0.8416,$$

$$b_{\text{new}} = 0.352 - 0.1 \times (-0.896) = 0.352 + 0.0896 \approx 0.4416.$$

训练过程总结

在这个简单例子中，神经网络的参数更新过程可以总结为：

1. **初始化**：随机或预设初始值（本例中 $w = 0.5$, $b = 0.1$ ）。
2. **前向传播**：利用当前参数计算输出 $y = w \cdot x + b$ 。
3. **计算损失**：用损失函数衡量输出与目标的差异。
4. **反向传播**：计算损失对各参数的梯度，得到更新方向。
5. **参数更新**：利用梯度下降公式更新 w 和 b 。
6. **迭代训练**：重复上述步骤，直至损失减小到可以接受的程度或达到预定的迭代次数。

经过多次迭代后，神经元的参数会逐渐调整，使得神经元的输出越来越接近目标输出，从而达到训练的目的。

拓展：多层神经网络

在实际应用中，我们通常使用多层神经网络（即深度神经网络）。其基本原理与上述单神经元相同，只不过：

- **每一层都有多个神经元**，每个神经元都有各自的参数；
- **激活函数**可能为非线性函数（如ReLU、Sigmoid、Tanh等）；
- **反向传播**时需要利用链式法则将梯度从输出层依次传递到各个隐藏层，计算每个参数对最终损失的贡献。

无论网络有多复杂，核心思想都是：**通过不断前向计算输出、衡量输出与目标之间的误差，再通过反向传播调整参数，从而使得网络能够更好地拟合数据。**

通过上述极简示例，我们可以直观地看到神经网络训练过程中参数是如何一步步从初始值调整到能够较好地“解释”训练数据的。这就是神经网络训练中参数确定的基本机制。

详解卷积滤波器的训练过程

神经网络中的滤波器（Filter）本质上是一个可学习的参数矩阵，其作用类似于图像处理中的特征检测器。下面通过具体示例说明其工作原理：

滤波器基本结构

典型尺寸为3x3或5x5的二维矩阵，例如：

水平边缘检测滤波器：
[[-1, -1, -1],
 [0, 0, 0],
 [1, 1, 1]]

该滤波器会对水平方向灰度变化剧烈的区域产生强响应

工作原理示例

假设输入为7x7的字母"X"图像：

```
0 0 0 1 0 0 0
0 0 1 0 1 0 0
0 1 0 0 0 1 0
1 0 0 0 0 0 1
0 1 0 0 0 1 0
0 0 1 0 1 0 0
0 0 0 1 0 0 0
```

应用3x3滤波器进行卷积运算：

1. 在图像左上角3x3区域：

```
0 0 0
0 0 1
0 1 0
```

与滤波器逐元素相乘后求和：

$$(0*-1)+(0*-1)+(0*-1) + (0*0)+(0*0)+(1*0) + (0*1)+(1*1)+(0*1) = 1$$

1. 滑动到中心区域：

```
0 0 0
0 0 0
0 0 0
```

计算结果为 0（无特征响应）

最终输出特征图将突出显示原始图像中的水平边缘。

```
conv1_weights = model.conv1.weight.data
print(conv1_weights)
```

```
tensor([[[[ 0.1756,  0.2263,  0.0860, -0.1815,  0.0349],
           [ 0.1034,  0.2493,  0.0010,  0.1445,  0.1512],
           [-0.0794,  0.2851,  0.1995,  0.0966,  0.1514],
           [-0.1405,  0.1541,  0.1301,  0.1867, -0.0275],
           [-0.1700,  0.1624,  0.2352, -0.0086,  0.0921]]],
```

```
[[[ 0.1063, -0.1332, -0.0569, 0.0996, 0.0642],
   [-0.2335, -0.0254, 0.1157, -0.1498, 0.0282],
   [-0.1154, 0.0278, 0.0321, 0.1847, 0.0306],
   [-0.2679, -0.1069, -0.1146, -0.0381, 0.0517],
   [-0.1555, -0.2238, -0.1804, -0.3030, -0.3209]]],
```

```
[[[-0.2110, -0.1945, -0.2105, 0.0011, 0.1709],
   [-0.2277, -0.3115, -0.3356, -0.2219, -0.1477],
   [-0.0029, -0.2690, -0.2542, -0.1519, 0.0219],
   [ 0.1367, -0.0836, -0.0315, -0.0777, 0.3794],
   [ 0.3076, 0.0754, 0.2394, 0.1263, 0.3758]]],
```

```
[[[-0.0039, 0.1558, -0.2461, -0.1579, 0.1522],
   [ 0.0123, -0.5837, -0.5194, -0.3398, 0.0627],
   [ 0.0525, -0.3838, -0.3286, 0.2018, 0.0976],
   [ 0.0690, 0.3707, 0.5339, 0.2229, -0.1445],
   [ 0.1462, 0.6403, 0.3796, 0.0692, -0.0991]]],
```

```
[[[ 0.1050, 0.3971, 0.3855, 0.3673, 0.2914],
   [ 0.2370, 0.3894, 0.4189, 0.5023, 0.4838],
   [-0.1980, -0.1557, -0.1465, -0.1301, 0.0777],
   [-0.5388, -0.5538, -0.4962, -0.5231, -0.0731],
   [-0.2018, -0.3179, -0.3028, -0.2609, -0.1368]]],
```

```
[[[-0.0469, -0.2489, 0.0176, 0.3116, 0.2809],
   [-0.4246, -0.3247, 0.0101, 0.3125, 0.3146],
   [-0.1961, -0.2134, -0.0186, 0.5693, 0.2271],
   [-0.3733, -0.0226, 0.4363, 0.4318, -0.1985],
   [-0.0771, 0.1501, 0.0153, 0.0863, -0.1739]]]])
```

```

from scipy.signal import convolve2d
from torch.nn import MaxPool2d

# 最大池化层
maxpool = MaxPool2d(kernel_size=2, stride=2)

plt.figure(figsize=(4, 10))

for kernel_idx, kernel in enumerate(conv1_weights):
    # 进行卷积
    convolved_image = convolve2d(matrix, kernel.squeeze().cpu().detach().numpy(),
mode='valid')
    pooled_image =
maxpool(torch.tensor(convolved_image).unsqueeze(0)).squeeze(0).cpu().detach().num
py()

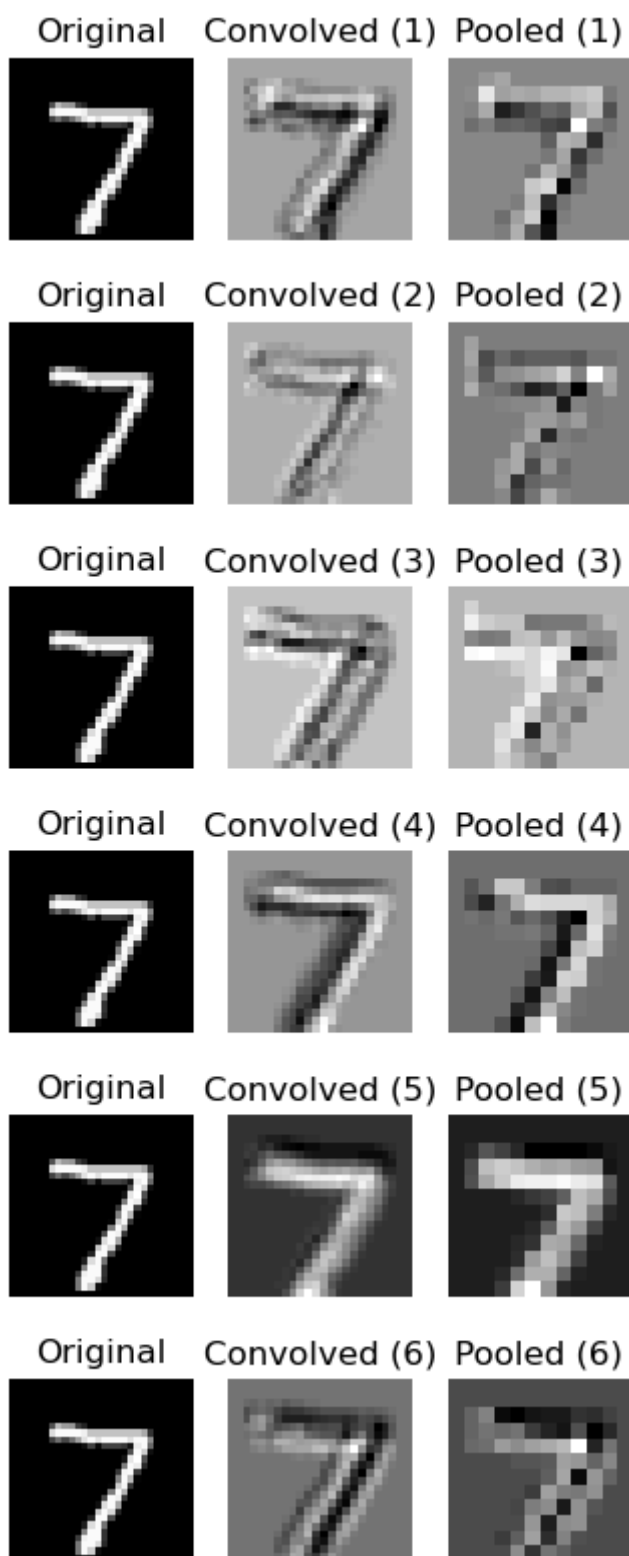
    plt.subplot(len(conv1_weights), 3, kernel_idx*3+1)
    plt.imshow(matrix, cmap='gray', interpolation='nearest')
    plt.title("Original")
    plt.axis("off")

    plt.subplot(len(conv1_weights), 3, kernel_idx*3+2)
    plt.imshow(convolved_image, cmap='gray', interpolation='nearest')
    plt.title(f"Convolved ({kernel_idx + 1})")
    plt.axis("off")

    plt.subplot(len(conv1_weights), 3, kernel_idx*3+3)
    plt.imshow(pooled_image, cmap='gray', interpolation='nearest')
    plt.title(f"Pooled ({kernel_idx + 1})")
    plt.axis("off")

plt.show()

```



总结

本项目详细介绍了：

- **神经网络基础知识**：从基本结构、训练过程到卷积操作的优势，帮助你了解神经网络的工作原理。
- **数据集**：通过 MNIST 数据集的介绍，了解了手写数字识别问题的背景。
- **环境配置**：如何利用 Conda 创建环境，并安装 PyTorch、torchvision 等必备工具。
- **完整代码实现**：从数据加载、模型构建到训练和评估，每一步均有详细注释，确保即使是初学者也能理解和上手。

通过这个项目，我从头到尾学了怎么用 PyTorch 做一个神经网络来认手写数字。开始是加载 MNIST 数据集，预处理后分成训练和测试两部分，然后用 LeNet 模型（有卷积层和全连接层）来提取特征和分类。训练时用 SGD 优化器和交叉熵损失函数，跑了 5 轮后，准确率能到 98% 以上，感觉挺厉害的！最后还画了图，看到了损失下降和准确率上升的过程，特别有成就感。

这个过程让我明白了神经网络是怎么一步步学习的：数据进来，经过卷积和池化提取特征，再通过全连接层分类。训练时用损失告诉模型哪里错了，梯度下降调整参数，慢慢让预测更准。代码里每部分都有注释，我跟着跑了一遍，觉得既好玩又实用。对初学者来说，这是个很棒的入门项目，激发了我对深度学习的兴趣，想再多学点更复杂的模型！