深度学习项目实战

本章旨在讲述如何使用现有的网络框架搭建神经网络
使用的框架：PyTorch
实践项目：手写数字识别（MNIST）

相关背景

什么是pytorch

Pytorch由Facebook AI Research和其他几个实验室开发，是一种用于构建深度学习模型的框架。它功能完备，使用简单易上手。并且pytorch完全支持GPU加速，是快速实验和原型设计的常用选择。

为什么要使用GPU加速

因为深度学习模型通常涉及大量的矩阵运算，而GPU（图形处理器）拥有数千个核心，能够并行处理这些计算，从而显著加快模型训练和推理的速度。在GPU的加持下，我们可以更快地得到模型训练的结果。

前期准备工作

包管理器Anaconda的安装与配置

Anaconda 用于管理不同版本python及其使用的库。

1. 登陆Anaconda官网：www.anaconda.com
2. 选择对应的python版本安装（使用最新稳定版即可）
3. 根据提示一键安装，记住安装的位置
4. 验证安装是否成功，可以打开终端（或Anaconda Prompt）输入 `conda --version` 查看版本信息。如果命令行显示“base”，则安装成功。
5. 配置一个conda环境：输入命令：

```
create -n myenv python=3.9
```

6. 输入命令：

```
conda activate myenv
```

即可激活相应的python环境

PyTorch环境的配置及安装

在激活的 conda 环境中，使用以下命令安装 PyTorch（请根据你的 CUDA 版本或 CPU 情况选择合适的安装命令，可在 PyTorch 官网获取最新命令）：

```
pip install torch torchvision torchaudio=、、、
```

安装完成后，可在 Python 中运行 `import torch` 验证是否安装成功。
下面是系统验证CUDA是否可用：

```
import torch
# 1. 检查CUDA是否可用
print("CUDA可用:", torch.cuda.is_available()) # 应返回True

# 2. 查看显卡数量和名称

print("显卡数量:", torch.cuda.device_count()) # 至少为1

print("当前显卡:", torch.cuda.current_device()) # 通常为0

print("显卡名称:", torch.cuda.get_device_name(0)) # 显示具体型号

# 3. 验证CUDA计算（创建张量并在GPU上运算）

if torch.cuda.is_available():
# 在GPU上创建一个随机张量
x = torch.rand(3, 3).cuda()
# 执行简单计算（矩阵乘法）
y = x @ x.T # 等价于x.matmul(x.T)
print("GPU计算结果:\n", y)
```

注意
安装前请确认你的 CUDA 版本（如果使用 GPU）或选择 CPU 版本的安装命令，具体命令请参考 PyTorch 官网。
可在命令行中输入：

```
nvidia-smi
```

选取小于设备“Driver Version”版本的Pytorch。

Python编辑器的选择与Jupyter notebook的使用

本教程使用vscode和jupyter notebook插件。其他编辑器亦可。

PyTorch 实战：LeNet 实现 MNIST 手写数字识别全流程详解

本文基于提供的代码和目录大纲，详细拆解用 PyTorch 训练 LeNet 网络完成 MNIST 手写数字识别的全流程。从环境准备到模型评估，逐行解析代码逻辑与原理，帮助新手掌握深度学习模型开发的核心步骤，理解每个环节的设计思路与作用。

MNIST数据集介绍

MNIST（Modified National Institute of Standards and Technology）是一个包含手写数字的数据集，里面有上万张数字图片，每张图片都标注了对应的数字标签（0到9）。这个数据集对我们来说，几乎可以算作一个“练手项目”。我们会用它来训练一个神经网络，让它学会识别这些手写数字。

- MNIST 包含 60,000 张手写数字训练图像和 10,000 张测试图像。
- 数据集由大小为 28x28 像素的灰度图像组成。
- 对图像进行归一化处理，使其适合 28x28 像素的边界框，并进行抗锯齿处理，引入灰度级。
- MNIST 广泛用于机器学习领域的训练和测试，尤其是图像分类任务。

MNIST 数据集分为两个子集：

1. **训练集**：该子集包含 60,000 张手写数字图像，用于训练机器学习模型。
2. **测试集**：该子集由 10,000 张图像组成，用于测试和基准测试训练有素的模型。

每张图片的尺寸是 **28x28 像素**，也就是 784 个数据点（28 * 28）。这些图片是灰度图，不包含颜色信息，每个像素的亮度值从 0（黑）到 255（白）。我们的任务就是训练一个模型，让它学会从这些 28x28 的图片中识别出数字。

一、工具与环境准备：搭建深度学习基础框架

核心代码引用：

```
import torchvision
from torch.utils.tensorboard import SummaryWriter
import torch
torch.cuda.is_available()
import torch
# 1. 检查CUDA是否可用
print("CUDA可用:", torch.cuda.is_available())  # 应返回True
# 2. 查看显卡数量和名称
print("显卡数量:", torch.cuda.device_count())  # 至少为1
print("当前显卡:", torch.cuda.current_device())  # 通常为0
print("显卡名称:", torch.cuda.get_device_name(0))  # 显示具体型号
```

```
# 3. 验证CUDA计算（创建张量并在GPU上运算）
if torch.cuda.is_available():
    # 在GPU上创建一个随机张量
    x = torch.rand(3, 3).cuda()
    # 执行简单计算（矩阵乘法）
    y = x @ x.T  # 等价于x.matmul(x.T)
    print("GPU计算结果:\n", y)
```

1. **工具库导入逻辑**
   - `torchvision`：PyTorch 官方计算机视觉工具库，提供现成数据集（如 MNIST）、数据变换工具和经典网络模型，避免重复开发。
   - `SummaryWriter`：Tensorboard 的 PyTorch 接口，用于记录训练过程中的损失、准确率等指标，实现可视化监控。
   - `torch`：PyTorch 核心库，提供张量计算、GPU 加速、自动求导等核心功能，是深度学习的基础。
2. **CUDA 环境检查的必要性**
   - 深度学习训练涉及大量矩阵运算，GPU（显卡）的并行计算能力远超 CPU，可大幅缩短训练时间。
   - `torch.cuda.is_available()`：判断系统是否支持 GPU 加速，返回 True 则说明可使用 CUDA。
   - 显卡信息查询代码（设备数量、名称）：帮助确认硬件资源，避免因显卡不兼容或未启用导致的错误。
   - GPU 计算验证：通过创建随机张量并执行矩阵乘法，验证 GPU 是否能正常进行计算，确保后续训练可顺利使用 GPU。

二、数据集准备与加载：构建模型的 “原料库”

```
#准备数据集
from torch import nn
from torch.utils.data import DataLoader
train_data = torchvision.datasets.MNIST(root="../data",train = True,
transform=torchvision.transforms.ToTensor(),download=True)
test_data = torchvision.datasets.MNIST(root="../data",train = False,
transform=torchvision.transforms.ToTensor(),download=True)
#数据集长度
train_data_size = len(train_data)
test_data_size = len(test_data)
print("训练数据集长度：{}".format(train_data_size))
print("测试数据集长度：{}".format(test_data_size))
#加载数据集
train_dataloader = DataLoader(train_data, batch_size=8192)
test_dataloader = DataLoader(test_data, batch_size=8192)
```

1. **MNIST 数据集介绍**
   - MNIST 是手写数字识别经典数据集，包含 60000 张训练图像和 10000 张测试图像，每张图像为 28×28 像素的灰度图（1 通道），标签为 0-9 的数字。
2. **数据集加载参数详解**
   - `root="../data"`：指定数据集存储路径，若路径不存在会自动创建。
   - `train=True/False`：区分训练集（True）和测试集（False），训练集用于模型学习，测试集用于评估模型泛化能力。
   - `transform=torchvision.transforms.ToTensor()`：将图像数据转换为 PyTorch 张量（Tensor），同时将像素值从 0-255 归一化到 0-1，符合神经网络输入要求。
   - `download=True`：若指定路径下没有数据集，自动从官方源下载，无需手动获取。
3. **数据集长度与批量加载**
   - 打印数据集长度：让开发者了解数据规模，60000 条训练数据和 10000 条测试数据是 MNIST 的标准规模。
   - `DataLoader`：PyTorch 的数据加载工具，核心作用是批量处理数据。
     - `batch_size=8192`：每次加载 8192 条数据进行训练 / 测试，批量加载可利用 GPU 并行计算能力，提升训练效率；批量大小需根据 GPU 显存调整，显存越大可设置越大。
     - 自动打乱数据（默认开启）：避免训练过程中模型学习到数据顺序规律，提升泛化能力。

三、LeNet 网络构建：搭建识别 “核心模型”

```
#创建网络
class LeNet(nn.Module):
    def __init__(self) :
        super(LeNet,self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(1,6,5,1,2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,stride=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(6,16,5),
            nn.ReLU(),
            nn.MaxPool2d(2,2)
        )
        self.fc1 = nn.Sequential(
            nn.Linear(16 * 5 * 5,120),
            nn.ReLU()
        )
        self.fc2 = nn.Sequential(
            nn.Linear(120,84),
```

```
            nn.ReLU()
        )
        self.fc3 = nn.Linear(84,10)
    #前向传播函数
    def forward(self,x):
            x = self.conv1(x)
            x = self.conv2(x)
            x = x.view(x.size(0), -1)           # flatten the output of
conv2 to (batch_size, 16 * 5 * 5)
            x = self.fc1(x)
            x = self.fc2(x)
            output = self.fc3(x)
            return output
myLeNet = LeNet()
myLeNet = myLeNet.cuda()
if torch.cuda.is_available():   # cuda可用时可置于GPU中训练网络，以加快计算速度
    myLeNet.cuda()
    print("cuda is available, and the calculation will be moved to GPU\n")
else:
    print("cuda is unavailable!")
```

1. **LeNet 网络简介**
   - LeNet 是经典的卷积神经网络（CNN），由 Yann LeCun 提出，专门用于手写数字识别，是深度学习在计算机视觉领域的奠基性模型，结构分为 “卷积层 + 池化层” 特征提取和 “全连接层” 分类决策两部分。
2. **网络层参数详解**
   - 继承`nn.Module`：PyTorch 中所有神经网络模型都需继承该类，其提供了参数管理、前向传播框架等核心功能。
   - `__init__`方法：定义网络层结构，使用`nn.Sequential`将多个层组合成模块，简化代码。
     - 卷积层（`nn.Conv2d`）：提取图像局部特征，参数含义依次为（输入通道数、输出通道数、卷积核大小、步长、填充）。
       - `conv1`：输入 1 通道（灰度图），输出 6 通道（6 个卷积核提取 6 种特征），卷积核 5×5，步长 1，填充 2（保持输入输出尺寸一致，28×28→28×28）。
       - `conv2`：输入 6 通道，输出 16 通道，卷积核 5×5，步长 1，无填充（28×28→24×24，池化后 12×12→6×6？实际计算：conv1 池化后 14×14，conv2 无填充卷积后 10×10，池化后 5×5）。
     - 激活函数（`nn.ReLU`）：引入非线性关系，解决线性模型无法拟合复杂特征的问题，ReLU 函数公式为 max (0,x)，计算简单且避免梯度消失。
     - 池化层（`nn.MaxPool2d`）：降低特征图尺寸，减少参数数量和计算量，同时保留关键特征，参数为（池化核大小、步长），此处 2×2 池化核 + 步长 2，特征图尺寸减半。

- 全连接层（`nn.Linear`）：将卷积提取的特征映射到分类空间，参数为（输入特征数、输出特征数）。
  - `fc1`：输入 16×5×5（conv2 输出的 16 通道 ×5×5 特征图），输出 120 维特征。
  - `fc3`：输出 10 维（对应 0-9 共 10 个数字类别）。

3. **前向传播函数（`forward`）**
   - 定义数据在网络中的流动路径，是模型的核心逻辑。
   - `x.view(x.size(0), -1)`：将卷积层输出的 4 维张量（batch_size, 通道数，高度，宽度）扁平化为 2 维张量（batch_size, 通道数 × 高度 × 宽度），适配全连接层输入要求，`-1` 表示自动计算剩余维度。
4. **模型移至 GPU**
   - 若 CUDA 可用，通过 `myLeNet.cuda()` 将模型参数移至 GPU，后续数据也需移至 GPU，确保计算设备一致（否则会报错）。

四、训练核心配置：设置模型 “学习规则”

```
#训练和测试
optimizer = torch.optim.RMSprop(myLeNet.parameters(),lr = 1e-3)
loss_func = nn.CrossEntropyLoss()
loss_func = loss_func.cuda()
#设置训练的一些参数
total_train_step = 0
total_test_step = 0
epoch = 50
#添加Tensorboard
writer = SummaryWriter("../logs_train")
```

1. **优化器（`optimizer`）**
   - 用于更新模型参数，最小化损失函数，此处使用 RMSprop 优化器（自适应学习率优化器）。
   - `myLeNet.parameters()`：传入模型所有可训练参数，优化器自动管理这些参数的更新。
   - `lr=1e-3`：学习率（0.001），控制参数更新幅度，学习率过大会导致训练震荡不收敛，过小则训练速度过慢。
2. **损失函数（`loss_func`）**
   - 衡量模型预测结果与真实标签的差距，是模型学习的 “指南针”。
   - `nn.CrossEntropyLoss`：适用于多分类任务，自动包含 Softmax 函数（将输出转换为概率分布）和负对数似然损失计算，无需手动添加 Softmax 层。
   - 损失函数移至 GPU：与模型、数据保持设备一致，确保计算正常。
3. **训练参数设置**

- `total_train_step` / `total_test_step`：记录训练 / 测试步数，用于 Tensorboard 日志记录和训练进度监控。
- `epoch=50`：训练轮次，指整个训练集被模型学习的次数，轮次过少模型欠拟合（未充分学习），轮次过多可能过拟合（过度学习训练集噪声）。

4. **Tensorboard 初始化**
   - `SummaryWriter("../logs_train")`：创建日志写入器，日志存储在 `../logs_train` 目录，后续通过 `writer.add_scalar` 记录指标，支持实时可视化。

五、模型训练过程：让模型 “学会” 识别数字

```
for i in range(epoch):
    print("------第{}轮训练开始------".format(i+1))
#训练开始
    myLeNet.train()
    for data in train_dataloader:
        imgs, target = data
        if torch.cuda.is_available():
            imgs = imgs.cuda()
            target = target.cuda()
        outputs = myLeNet(imgs)
        loss = loss_func(outputs, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_train_step = total_train_step + 1
        if total_train_step % 100 == 0:
            print("训练次数:{}, Loss:
{}".format(total_train_step,loss.item()))
            writer.add_scalar("train_loss",loss.item(),total_train_step)
```

1. **训练轮次循环（`epoch` 循环）**
   - 每轮训练包含 “遍历全部训练数据” 和 “一次测试评估”，打印轮次信息方便监控训练进度。
2. **训练模式切换（`myLeNet.train()`）**
   - 启用模型的训练模式，对 Dropout 层（本模型未使用）、BatchNorm 层等特殊层生效，确保训练过程中这些层正常工作。
3. **批量数据训练循环**
   - 遍历 `train_dataloader`，每次获取一个批量的图像（`imgs`）和对应的标签（`target`）。
   - 数据移至 GPU：与模型设备一致，避免计算错误。

- 前向传播：`myLeNet(imgs)` 调用 `forward` 函数，得到模型预测输出（`outputs`，shape 为 (batch_size, 10)）。
- 计算损失：`loss_func(outputs, target)` 对比预测输出和真实标签，得到当前批量的损失值，损失值越大说明预测越不准确。

4. **反向传播与参数更新（核心步骤）**
   - `optimizer.zero_grad()`：清零上一轮的梯度，避免梯度累积导致参数更新异常（梯度累积适用于特定场景，此处默认清零）。
   - `loss.backward()`：自动计算损失函数对所有可训练参数的梯度（基于 PyTorch 的自动求导机制）。
   - `optimizer.step()`：根据计算出的梯度，使用优化器的更新规则（RMSprop）调整模型参数，减小损失。
5. **训练日志记录**
   - 每训练 100 步打印损失值：`loss.item()` 获取损失张量的数值（剥离计算图），方便查看训练趋势。
   - `writer.add_scalar("train_loss", loss.item(), total_train_step)`：将训练损失记录到 Tensorboard，横轴为训练步数，纵轴为损失值，支持后续可视化分析。

六、模型测试过程：评估模型 “识别能力”

```
#测试步骤
    myLeNet.eval()
    total_test_loss = 0
    total_accuracy = 0
    with torch.no_grad():
        for data in test_dataloader:
            imgs,target = data
            if torch.cuda.is_available():
                imgs = imgs.cuda()
                target = target.cuda()
            outputs =  myLeNet(imgs)
            loss = loss_func(outputs, target)
            total_test_loss = total_test_loss + loss.item()
            accuracy = (outputs.argmax(1) == target).sum()
            total_accuracy = total_accuracy + accuracy
    print("整体测试Loss:{}".format(total_test_loss))
    print("整体测试准确率:{}".format(total_accuracy/test_data_size))
    writer.add_scalar("test_loss",total_test_loss,total_test_step)

writer.add_scalar("test_accuracy",total_accuracy/test_data_size,total_test_s
tep)
    total_test_step = total_test_step + 1
```

1. **测试模式切换（`myLeNet.eval()`）**
   - 禁用训练模式下的特殊层（如 Dropout、BatchNorm 的更新），确保测试过程中模型参数固定，评估结果准确。
2. **禁用梯度计算（`with torch.no_grad()`）**
   - 测试过程无需计算梯度（仅需前向传播），禁用梯度计算可节省 GPU 显存，大幅提升测试速度，同时避免不必要的计算开销。
3. **测试损失计算**
   - 遍历测试集所有批量数据，累计每个批量的损失值（`total_test_loss`），最终得到测试集整体损失，反映模型在未见过的数据上的预测误差。
4. **测试准确率计算**
   - `outputs.argmax(1)`：对模型输出的 10 维向量取最大值索引，即预测的数字类别（axis=1 表示按行取最大值索引，对应每个样本）。
   - `(outputs.argmax(1) == target)`：比较预测类别与真实标签，得到布尔张量（True 表示预测正确，False 表示错误）。
   - `.sum()`：统计预测正确的样本数量，累计得到`total_accuracy`。
   - 准确率 = 正确样本数 / 测试集总样本数，反映模型的实际识别能力（MNIST 数据集优秀模型准确率可达 99% 以上）。
5. **测试日志记录**
   - 将测试损失和准确率记录到 Tensorboard，方便与训练损失对比，判断模型是否过拟合（若训练损失下降但测试损失上升，说明过拟合）。

七、模型保存与效果评估：沉淀与优化模型

```
    torch.save(myLeNet,"myLeNet.pth".format(i))
    print("模型已保存")
writer.close()
```

1. **模型保存（`torch.save`）**
   - 保存训练好的模型，方便后续直接加载使用（无需重新训练）。
   - 此处保存整个模型（`myLeNet`），文件名为`myLeNet.pth`，格式为 PyTorch 专用的`.pth`文件，包含模型结构和所有参数。
   - 若需节省存储空间，也可仅保存模型参数（`torch.save(myLeNet.state_dict(), "myLeNet_params.pth")`），加载时需先创建模型结构再加载参数。
2. **Tensorboard 关闭（`writer.close()`）**
   - 训练结束后关闭日志写入器，确保所有日志数据正确写入文件，避免数据丢失。
3. **效果评估方法**
   - 运行 Tensorboard：在终端执行`tensorboard --logdir=../logs_train`，打开浏览器访问提示的地址（默认http://localhost:6006），可查看训练 / 测试损失曲线、准确率曲线。

- 曲线分析：
  - 训练损失应持续下降，最终趋于平稳（说明模型在学习）。
  - 测试损失应先下降后平稳，若后期上升则需调整（如减少 epoch、使用正则化）。
  - 测试准确率应持续上升，最终稳定在较高水平（本代码 epoch=50，预计准确率可达 98% 以上）。 至此，模型训练完成，可通过加载保存的模型文件进行后续预测或部署。

八、全流程总结与核心要点

本项目完整实现了 “数据准备→网络构建→训练配置→训练过程→测试评估→模型保存” 的深度学习模型开发流程，核心要点如下：

1. 设备一致性：模型、数据、损失函数需统一移至 GPU 或 CPU，避免计算错误。
2. 模式切换：训练时用 `train()`，测试时用 `eval()`，确保评估准确。
3. 可视化监控：Tensorboard 是关键工具，可实时跟踪训练进度，及时发现过拟合、训练震荡等问题。
4. 超参数调整：学习率（`lr`）、批量大小（`batch_size`）、训练轮次（`epoch`）需根据硬件资源和模型表现调整，是模型优化的核心方向。