# 实验三：深度学习之图像识别 CNN（PyTorch CPU，2 学时）

## 学习目标

* 使用最简单的卷积神经网络（两层卷积）在 CPU 上完成 MNIST 手写数字识别的训练与推理。
* 会按步骤运行带有详细中文注释的脚本，理解数据加载、模型、训练、评估与保存的基本流程。
* 能保存训练日志、模型与预测可视化图，便于在实验报告中引用。 （本章课时：**2 学时**）

## 先修知识与软件环境

* 系统：Windows 10 / cmd
* Python：3.12（已配置清华源）
* 依赖：PyTorch CPU 版、torchvision、torchaudio（通过官方 CPU 源安装）

## 实验准备

1. 激活虚拟环境：

\.venv\Scripts\activate.bat

1. 安装 PyTorch（CPU 版，使用官方 CPU 源）：

pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu  
python -c "import torch;print(torch.\_\_version\_\_,'cuda?',torch.cuda.is\_available())"

1. 创建输出与数据目录：

mkdir outputs  
mkdir data

## 核心知识要点

* 卷积层：提取局部特征；池化层：降低尺寸、保留关键信息；全连接层：完成分类。
* 损失函数：交叉熵（CrossEntropyLoss）适用于多分类；优化器：Adam/SGD 都可，Adam 更省心。
* 批量训练：DataLoader 按批次读取数据，统计训练/验证准确率。

## 实验任务清单

1. 下载并加载 MNIST（自动下载到 ./data）。
2. 定义轻量 CNN（两层卷积 + 最大池化 + 两层全连接）。
3. 在 CPU 上训练 1–2 个 epoch，记录训练/验证损失与准确率。
4. 保存最佳模型 mnist\_cnn.pth，并生成预测可视化 pred\_grid.png。

## 操作步骤

执行策略：只需运行给定命令/代码即可完成。代码已包含详细中文注释。

1. 新建脚本 exp3\_cnn\_mnist.py，将以下代码完整复制粘贴保存：

"""  
脚本名称：exp3\_cnn\_mnist.py  
用途：在 CPU 上用最简单的 CNN 训练 MNIST，并保存模型与预测可视化图。  
  
你将学到：  
 - 数据如何被读取、标准化，再送入模型；张量的形状含义（batch, channel, height, width）。  
 - CNN 的基本结构（卷积→激活→池化→展平→全连接）以及每层的输出形状。  
 - 训练/验证的区别，损失与准确率的直观意义，以及如何保存/加载模型。  
  
如何运行（Windows 10 + cmd）：  
 1) 激活虚拟环境： .\.venv\Scripts\activate.bat  
 2) 安装依赖（一次）：pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu  
 3) 执行命令： python exp3\_cnn\_mnist.py --epochs 2 --batch-size 128 --lr 0.001 --optimizer adam --data-dir .\data --out-dir .\outputs  
  
输出文件：  
 - 模型：outputs\mnist\_cnn.pth  
 - 预测网格：outputs\pred\_grid.png（4×4 共 16 张图）  
 - 日志：控制台打印每个 epoch 的训练/验证损失与准确率（建议截图放入报告）  
  
小贴士（速度与稳定）：  
 - 如果训练较慢，可将 --epochs 改为 1，或将 --batch-size 改为 64。  
 - 本实验固定使用 CPU（无需/不使用 GPU）。随机性已通过随机种子尽量控制，但仍会有轻微波动。  
"""  
  
import argparse  
import os  
from typing import Tuple  
import random  
import numpy as np  
  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from torch.utils.data import DataLoader  
  
from torchvision import datasets, transforms  
import matplotlib.pyplot as plt  
  
  
def set\_seed(seed: int) -> None:  
 """设置随机种子，尽量保证结果可复现。  
 说明：深度学习中完全复现很困难，但这能显著降低随机波动。  
 """  
 random.seed(seed)  
 np.random.seed(seed)  
 torch.manual\_seed(seed)  
  
  
class SimpleCNN(nn.Module):  
 """最小可用 CNN：两层卷积 + 最大池化 + 两层全连接。  
  
 输入张量形状： (batch\_size, 1, 28, 28)  
 - 1 表示灰度通道；28×28 是 MNIST 图像分辨率。  
  
 结构与形状变化：  
 - Conv1: in=1, out=16, kernel=3, padding=1 → 形状保持 28×28（通道变为 16）  
 - Conv2: in=16, out=32, kernel=3, padding=1 → 形状保持 28×28（通道变为 32）  
 - MaxPool: kernel=2, stride=2 → 尺寸减半为 14×14  
 - Flatten: 32×14×14 = 6272 → 展平成一维向量  
 - FC1: 6272 → 128 → 特征压缩  
 - FC2: 128 → 10 → 输出 10 类（数字 0~9）  
 """  
  
 def \_\_init\_\_(self) -> None:  
 super().\_\_init\_\_()  
 # 卷积核 3×3，padding=1 表示在边缘补 1 圈，使宽高不变  
 self.conv1 = nn.Conv2d(1, 16, kernel\_size=3, stride=1, padding=1)  
 self.conv2 = nn.Conv2d(16, 32, kernel\_size=3, stride=1, padding=1)  
 self.pool = nn.MaxPool2d(kernel\_size=2, stride=2) # 将 28×28 压到 14×14  
 # 32 个通道 × 14 × 14 = 6272（展平后的长度）  
 self.fc1 = nn.Linear(32 \* 14 \* 14, 128)  
 self.fc2 = nn.Linear(128, 10)  
  
 def forward(self, x: torch.Tensor) -> torch.Tensor:  
 # x 初始形状：(B, 1, 28, 28)  
 x = F.relu(self.conv1(x)) # → (B, 16, 28, 28)  
 x = F.relu(self.conv2(x)) # → (B, 32, 28, 28)  
 x = self.pool(x) # → (B, 32, 14, 14)  
 x = torch.flatten(x, 1) # → (B, 6272)  
 x = F.relu(self.fc1(x)) # → (B, 128)  
 x = self.fc2(x) # → (B, 10)  
 return x  
  
  
def count\_parameters(model: nn.Module) -> int:  
 """统计可训练参数量，帮助直观理解模型规模（值越小说明越轻量）。"""  
 return sum(p.numel() for p in model.parameters() if p.requires\_grad)  
  
  
def get\_loaders(data\_dir: str, batch\_size: int, seed: int) -> Tuple[DataLoader, DataLoader]:  
 """下载并构建训练/测试数据加载器（CPU 环境）。  
  
 关键点：  
 - transforms.ToTensor() 将像素从 [0,255] 映射到 [0,1]，并把形状从 (H,W) 变为 (1,H,W)。  
 - Normalize 使用 MNIST 官方均值/方差，让数据分布更稳定（有利于训练）。  
 - num\_workers 在 Windows 下建议为 0，避免多进程带来的不必要开销与兼容问题。  
 """  
 set\_seed(seed)  
 transform = transforms.Compose([  
 transforms.ToTensor(),  
 transforms.Normalize((0.1307,), (0.3081,))  
 ])  
  
 train\_ds = datasets.MNIST(root=data\_dir, train=True, download=True, transform=transform)  
 test\_ds = datasets.MNIST(root=data\_dir, train=False, download=True, transform=transform)  
  
 train\_loader = DataLoader(train\_ds, batch\_size=batch\_size, shuffle=True, num\_workers=0)  
 test\_loader = DataLoader(test\_ds, batch\_size=batch\_size, shuffle=False, num\_workers=0)  
  
 print(f" 训练集大小: {len(train\_ds)} 测试集大小: {len(test\_ds)} 批大小: {batch\_size}")  
 return train\_loader, test\_loader  
  
  
def accuracy(pred\_logits: torch.Tensor, targets: torch.Tensor) -> float:  
 """计算准确率（top-1）。  
 解释：选择每行（每个样本）10 类中概率最大的那个作为预测，与真实标签对比。  
 """  
 preds = pred\_logits.argmax(dim=1)  
 correct = (preds == targets).sum().item()  
 return correct / targets.size(0)  
  
  
def train\_one\_epoch(model: nn.Module, loader: DataLoader, optimizer: torch.optim.Optimizer, criterion: nn.Module) -> Tuple[float, float]:  
 """单个训练轮：前向→计算损失→反向传播→参数更新。  
 返回该轮的平均损失与平均准确率。  
 """  
 model.train() # 训练模式：启用 Dropout/BN 的训练行为（本模型未用到，但这是规范写法）  
 total\_loss = 0.0  
 total\_acc = 0.0  
 total\_samples = 0  
  
 for images, labels in loader:  
 # images: (B,1,28,28) labels: (B,) 数字 0~9  
 optimizer.zero\_grad() # 清空上一次的梯度  
 logits = model(images) # 前向传播，得到每个类别的打分  
 loss = criterion(logits, labels) # 交叉熵损失，衡量预测与真实的差距  
 loss.backward() # 反向传播，计算每个参数的梯度  
 optimizer.step() # 按梯度方向更新参数  
  
 batch\_size = labels.size(0)  
 total\_loss += loss.item() \* batch\_size  
 total\_acc += accuracy(logits, labels) \* batch\_size  
 total\_samples += batch\_size  
  
 return total\_loss / total\_samples, total\_acc / total\_samples  
  
  
def evaluate(model: nn.Module, loader: DataLoader, criterion: nn.Module) -> Tuple[float, float]:  
 """在验证/测试集评估模型：不更新参数，仅做前向与指标统计。"""  
 model.eval()  
 total\_loss = 0.0  
 total\_acc = 0.0  
 total\_samples = 0  
  
 with torch.no\_grad():  
 for images, labels in loader:  
 logits = model(images)  
 loss = criterion(logits, labels)  
 batch\_size = labels.size(0)  
 total\_loss += loss.item() \* batch\_size  
 total\_acc += accuracy(logits, labels) \* batch\_size  
 total\_samples += batch\_size  
  
 return total\_loss / total\_samples, total\_acc / total\_samples  
  
  
def save\_pred\_grid(model: nn.Module, loader: DataLoader, out\_png: str, num\_samples: int = 16) -> None:  
 """保存若干测试样本的预测网格图（默认 4×4 共 16 张）。  
 提示：若批大小小于 16，将自动取可用数量；生成的图适合放入报告中展示效果。  
 """  
 model.eval()  
 images, labels = next(iter(loader))  
 num = min(num\_samples, images.size(0))  
 images = images[:num]  
 labels = labels[:num]  
 with torch.no\_grad():  
 logits = model(images)  
 preds = logits.argmax(dim=1)  
  
 # 绘制 4×4 网格（如不足 16 张，则只画前 num 张）  
 rows, cols = 4, 4  
 plt.figure(figsize=(cols \* 2.0, rows \* 2.0))  
 for i in range(num):  
 plt.subplot(rows, cols, i + 1)  
 plt.imshow(images[i].squeeze(0), cmap="gray")  
 title = f"pred={preds[i].item()}\ntrue={labels[i].item()}"  
 plt.title(title, fontsize=9)  
 plt.axis("off")  
 plt.tight\_layout()  
 plt.savefig(out\_png, dpi=150)  
 plt.close()  
  
  
def main():  
 parser = argparse.ArgumentParser(description="Exp3: Simple CNN on MNIST (CPU)")  
 parser.add\_argument("--epochs", type=int, default=2, help="训练轮数，CPU 建议 1–2")  
 parser.add\_argument("--batch-size", type=int, default=128, help="批大小，CPU 建议 64–128")  
 parser.add\_argument("--lr", type=float, default=1e-3, help="学习率（步子大小，过大不稳定，过小收敛慢）")  
 parser.add\_argument("--optimizer", choices=["adam", "sgd"], default="adam", help="优化器选择（Adam 更省心）")  
 parser.add\_argument("--data-dir", default="./data", help="数据目录（MNIST 自动下载）")  
 parser.add\_argument("--out-dir", default="./outputs", help="输出目录")  
 parser.add\_argument("--seed", type=int, default=42, help="随机种子（保证多次运行结果接近）")  
 args = parser.parse\_args()  
  
 os.makedirs(args.out\_dir, exist\_ok=True)  
 set\_seed(args.seed)  
  
 # 1) 数据加载  
 print("[1/5] 加载数据（MNIST，将自动下载到 ./data）...")  
 train\_loader, test\_loader = get\_loaders(args.data\_dir, args.batch\_size, args.seed)  
 # 打印一个批次的形状，帮助理解张量维度  
 sample\_images, sample\_labels = next(iter(train\_loader))  
 print(  
 f" 样例批次形状：images={tuple(sample\_images.shape)} "  
 f"labels={tuple(sample\_labels.shape)} (B,1,28,28 / (B,))"  
 )  
  
 # 2) 定义模型/损失/优化器（CPU 环境）  
 print("[2/5] 创建模型与优化器（CPU）...")  
 model = SimpleCNN()  
 print(f" 可训练参数量：{count\_parameters(model):,}")  
 criterion = nn.CrossEntropyLoss()  
 if args.optimizer == "adam":  
 optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)  
 print(  
 " 使用 Adam 优化器（对学习率较不敏感，入门更稳）"  
 )  
 else:  
 optimizer = torch.optim.SGD(model.parameters(), lr=args.lr, momentum=0.9)  
 print(  
 " 使用 SGD 优化器（可配合较大学习率与动量）"  
 )  
  
 # 3) 训练与验证  
 print("[3/5] 开始训练（CPU 环境，小批量、轮数少即可看到效果）...")  
 best\_acc = 0.0  
 best\_path = os.path.join(args.out\_dir, "mnist\_cnn.pth")  
 for epoch in range(1, args.epochs + 1):  
 train\_loss, train\_acc = train\_one\_epoch(model, train\_loader, optimizer, criterion)  
 val\_loss, val\_acc = evaluate(model, test\_loader, criterion)  
 print(  
 f"Epoch {epoch:02d}/{args.epochs}: "  
 f"train\_loss={train\_loss:.4f}, train\_acc={train\_acc:.3f}, "  
 f"val\_loss={val\_loss:.4f}, val\_acc={val\_acc:.3f}"  
 )  
  
 # 保存更优模型（以验证集准确率为准）  
 if val\_acc > best\_acc:  
 best\_acc = val\_acc  
 torch.save({  
 "model\_state": model.state\_dict(),  
 "best\_val\_acc": best\_acc,  
 "config": vars(args)  
 }, best\_path)  
 print(f" 已保存更优模型: {best\_path} (val\_acc={best\_acc:.3f})")  
  
 # 4) 载入最佳模型并生成预测网格  
 print("[4/5] 载入最佳模型并生成预测网格图...")  
 ckpt = torch.load(best\_path, map\_location="cpu")  
 model.load\_state\_dict(ckpt["model\_state"])  
 grid\_png = os.path.join(args.out\_dir, "pred\_grid.png")  
 save\_pred\_grid(model, test\_loader, grid\_png, num\_samples=16)  
  
 # 5) 总结与下一步  
 print("[5/5] 完成。文件已生成到输出目录：")  
 print(" - 模型:", best\_path)  
 print(" - 预测网格:", grid\_png)  
 print("提示：如需更快实验，可减小 --epochs 或 --batch-size；如需更好精度，可适度增加 --epochs。")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

1. 运行（无需改代码）：

python exp3\_cnn\_mnist.py --epochs 2 --batch-size 128 --lr 0.001 --optimizer adam --data-dir .\data --out-dir .\outputs --seed 42

可选：尝试更小的 --epochs 1 或更小的 --batch-size 64 以加快速度。

1. 查看输出：

* 模型：outputs/mnist\_cnn.pth
* 预测网格图：outputs/pred\_grid.png
* 训练日志：查看控制台输出（建议截图保存到报告）

## 预期输出

* 在 CPU 上 1–2 个 epoch 可达较好的验证准确率（>95% 仅作参考，取决于批大小与训练轮数）。
* pred\_grid.png 展示 16 张测试图片与预测/真实标签，直观观察模型效果与偶发错误。

## 思考与讨论

1. 当我们把 --epochs 从 1 改为 2 时，准确率变化明显吗？为什么多训练会更好？
2. 如果把 --batch-size 改小，训练会更慢还是更快？准确率会受影响吗？
3. 观察 pred\_grid.png 中预测错误的样本，它们有哪些共同特征（例如写得潦草、模糊）？

## 常见错误与排错

* 下载数据失败：检查网络；或从其他同学处拷贝 ./data/MNIST 目录。
* 训练很慢：将 --epochs 改为 1，或将 --batch-size 改为 64；确保未使用 GPU 相关设置。
* ModuleNotFoundError: No module named 'torch'：确认按指南使用官方 CPU 源安装 PyTorch。

## 提交要求与评分标准

* 提交内容：PDF 实验报告 + outputs 目录（含模型与 pred\_grid.png）。
* 命名规范：学号\_姓名\_实验三
* 评分 Rubric：准确性 40% / 完整性 30% / 表达 20% / 规范 10%