山东大学 软件 学院

人工智能综合实践 课程实验报告

学号：202400300121	姓名：李瑞菡	班级：24 级 AI 班
实验编号：实验一		
实验题目：深度学习环境搭建、模型实践与性能对比实验		
实验学时：8	实验日期：2025.9.4-2025.9.16	

实验目的：

1.掌握**深度学习基础环境的搭建方法**，包括操作系统适配、显卡驱动安装、CUDA 工具包配置、Anaconda 虚拟环境管理以及 PyTorch 框架部署，确保深度学习开发所需的软硬件环境正常运行。

2.能够**独立运行课程提供的基于 LeNet 的 CIFAR－10 图像分类示例程序和基于全连接模型的 MNIST 手写数字识别示例程序**，熟悉 “数据下载 － 模型训练 － 验证 － 保存 － 推理” 的深度学习完整流程。

3.学会**对示例程序进行适当修改**，如调整模型结构、数据预处理方式、训练参数等，使程序产生不同输出，深入理解代码逻辑与深度学习核心概念，提升代码实践与优化能力。

4.掌握**软件依赖包的安装方法**，能够处理依赖包之间的版本冲突等问题，保障程序在特定环境下的可复现性。

5.**对比深度学习模型在 CPU 和 GPU 环境下完成训练任务的性能差异**(如训练耗时、资源占用等)，理解 GPU 加速深度学习计算的原理与优势。

6.通过**记录实验过程中遇到的问题及解决办法**，培养发现问题、分析问题和解决问题的能力，为后续更复杂的深度学习实验与项目开展积累经验。

硬件环境：

搭载 Hygon C86 3185 8 核 CPU（3000MHz，16 逻辑处理器），配备 64.0GB 物理内存，系统制造商为 Suma，型号 W330 － H30，运行 Windows 10 专业版系统

软件环境：

VScode

实验步骤与内容：

实验任务分析说明:

① 独立完成全部实验<动手能力的培养，全流程过一遍>

② 完成操作系统、驱动程序、CUDA、Anaconda、PyTorch 的安装<环境的安装与配置>，

完成课程提供示例的运行，并尝试对示例程序进行适当修改<创新能力的培养>，使其具有不同的输出。掌握软件依赖包的安装及冲突处理能力

③ 比较 GPU 与 CPU 在完成相同计算任务时的时间消耗，以及对计算机运行带来的其他影响<资料的收集+动手实践的收获>

④ 将整个安装和运行过程和结果整理成实验报告并提交<过程的记录与收集>，报告中要记录至少两个过程中遇到的问题和解决办法<发现问题与解决问

题的能力>

⑤ 将修改后的程序代码打包提交

以下将记录实验过程以及遇到的问题与处理的方案

实验过程的记录

一、GPU 驱动程序的安装/更新<主要是想要体验一下全过程>

（对于过程的体验以及更新 GPU 驱动程序）

1. 查看显卡型号

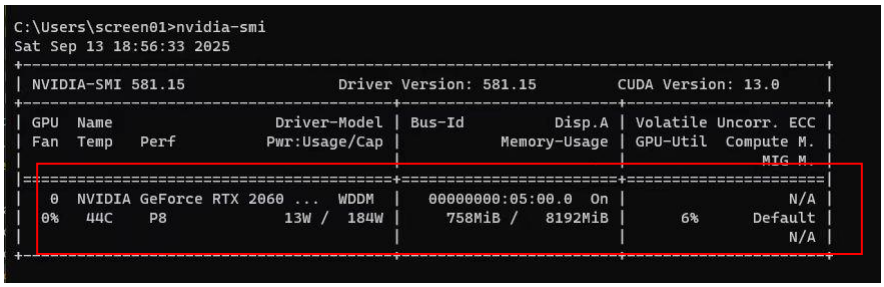
2. 安装显卡型号对应的显卡驱动

在 Nvidia 驱动程序下载官网：https://www.nvidia.cn/geforce/drivers/

选择与自己电脑上显卡对应的版本

3. 检验驱动是否安装成功

输入 **nvidia-smi** 查看驱动信息

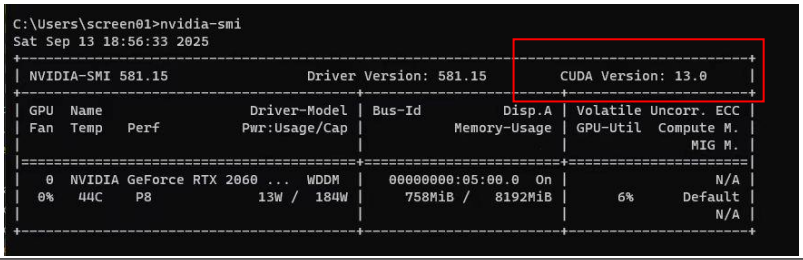


```
C:\Users\screen01>nvidia-smi
Sat Sep 13 18:56:33 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 581.15                 Driver Version: 581.15         CUDA Version: 13.0     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                  Driver-Model | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 2060 ...  WDDM  |   00000000:05:00.0  On |                  N/A |
|  0%   44C    P8             13W /  184W |     758MiB /   8192MiB |      6%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

二、CUDA 的安装

1. 确定 CUDA 版本

仍然输入 nvidia-smi 查看 CUDA 的驱动版本，可以看到 CUDA 的驱动版本为 13.0，因此本机可以支持<=13.0 版本的 cuda 安装

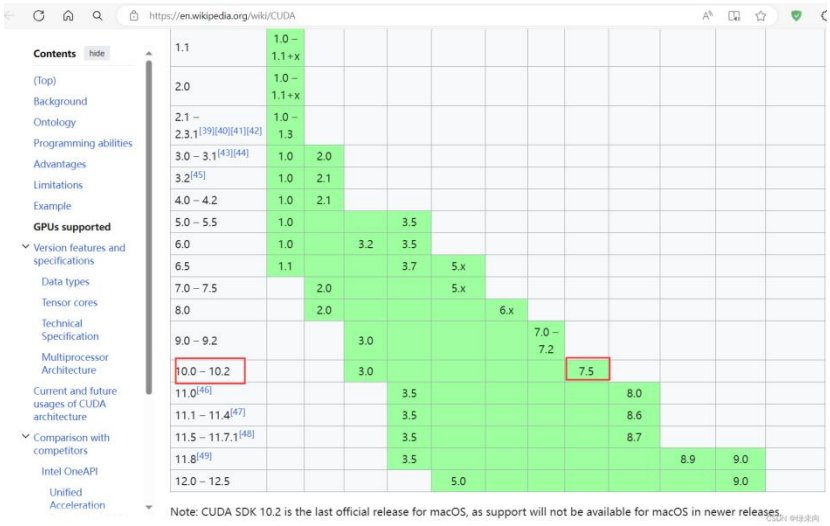


```
C:\Users\screen01>nvidia-smi
Sat Sep 13 18:56:33 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 581.15                 Driver Version: 581.15         CUDA Version: 13.0     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                  Driver-Model | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 2060 ...  WDDM  |   00000000:05:00.0  On |                  N/A |
|  0%   44C    P8             13W /  184W |     758MiB /   8192MiB |      6%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

2. 查找显卡对应的算力（这一步万老师上课强调过，第一次的没有关注，后来去查询对应的算力，发现对应的算力是 7.5）

3. 确定自己可以选择的 CUDA Runtime Version

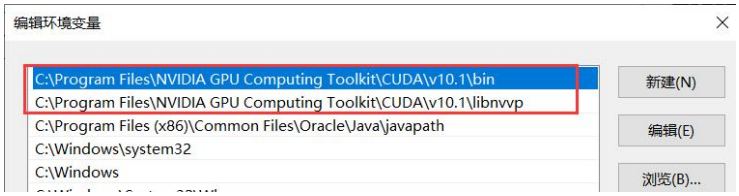
可以发现 **7.5 的算力对应 10-10.2 的 CUDA 版本**



4. 找到适合自己的 CUDA 版本进行安装

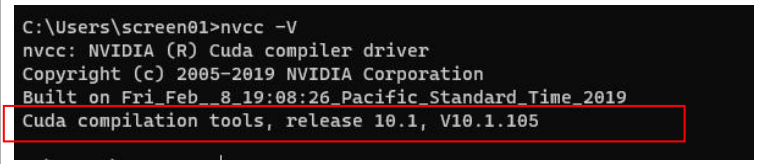
5. 添加环境变量

我下载完成后**只自动配置了 4 条系统变量，没有自动配置 path 中的环境变量**，因此手动配置了 path 中的环境变量



6. 验证 CUDA 是否安装成功（这一步第一次存在问题，一直显示不存在此命令，原因是环境变量未配置）

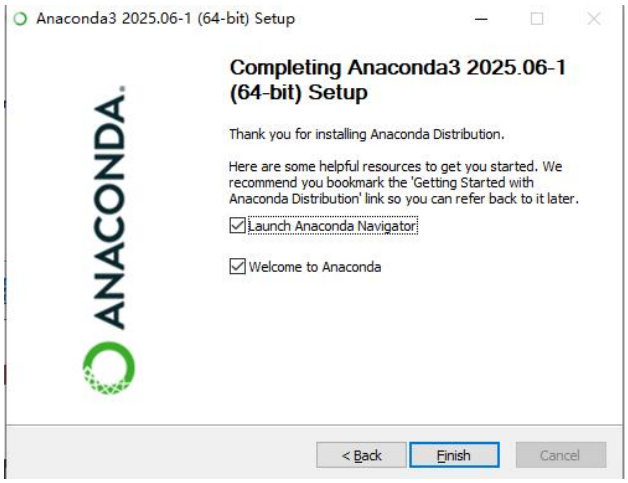
配置完成环境变量后，打开命令提示符，输入 **nvcc -V**



三、Anaconda 的安装

在下载安装之前，结合万老师上课所讲先对 anaconda 进行了初步的了解：Anaconda，是一个开源的 Python 发行版本，其包含了 conda、Python 等 180 多个科学包及其依赖项。其中，conda 是一个开源的包、环境管理器，可以用于在同一个机器上安装不同版本的软件包及其依赖，并能够在不同的环境之间切换。

1. 由于**先尝试在官网进行下载，但是下载速度太慢了**，通过查找相关的资料<*CSDN*>发现可以利用**清华大学镜像网站**下载最新版的 Anaconda



2. 第二步，配置清华镜像源作为下载源

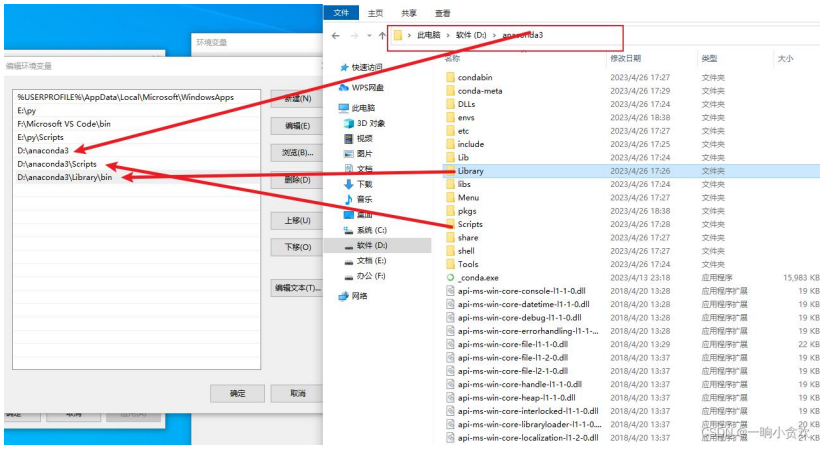
解释：Anaconda 的下载源默认在国外，如果不配置我们国内源的话，下载速度会非常慢，而且很多时候会导致网络错误而下载失败

在 Anaconda Prompt，进行以下命令的执行：

```
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main/
conda config --set show_channel_urls yes
```

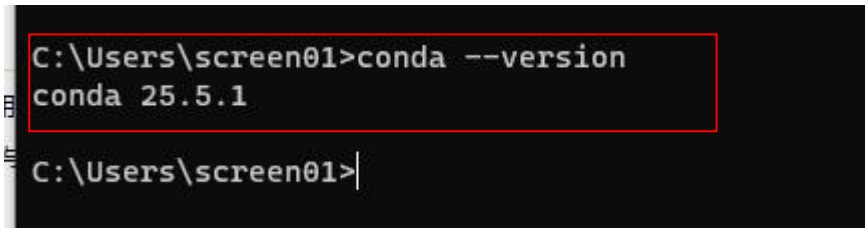
3. 配置环境变量

由于此步骤当时没有截图，故参考 CSDN 给出操作步骤



4. 验证是否安装成功

打开命令提示符，输入 **conda --version**

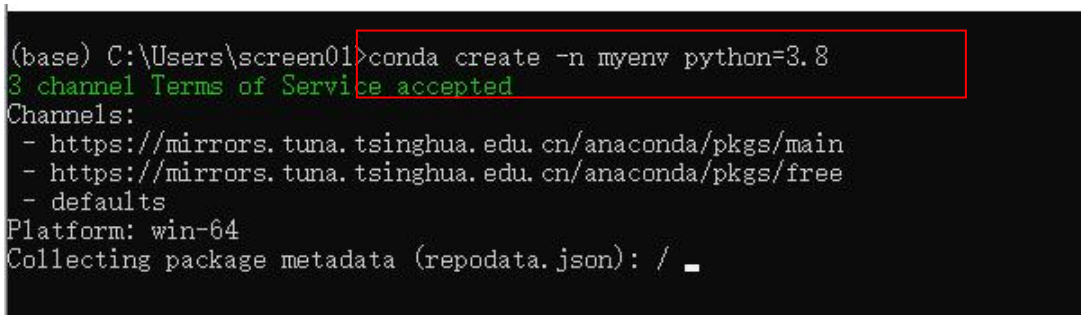


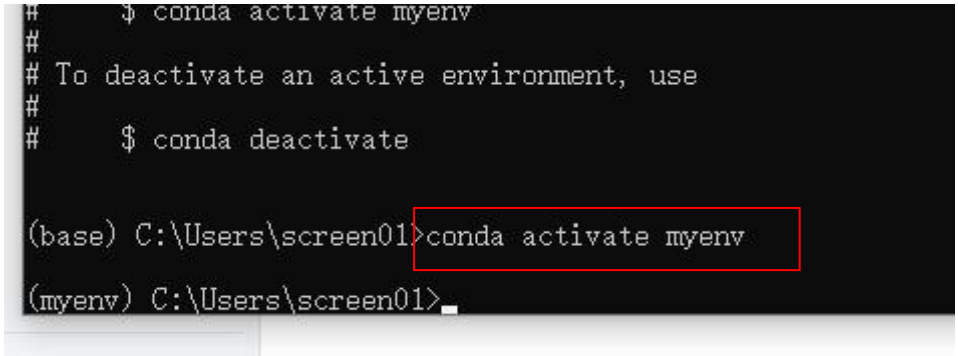
四、**PyTorch 的安装**（**注：此步骤均在 anaconda prompt 上进行的，后续运行示例在 vscode 上重新进行 Pytorch 的安装，此处更多的是对于流程的熟悉**）

1. 创建并激活虚拟环境

打开 Anaconda Prompt，输入命令 **conda create -n myenv python=3.8**，回车执行，创建名为 myenv、Python 版本为 3.8 的虚拟环境。

环境创建完成后，输入 **conda activate myenv**，回车激活该虚拟环境，此时命令行前缀会变为（myenv）





2. 进入 PyTorch 官网安装页面，根据需求选择 CPU 或 GPU 版本，进行

PyTorch 的安装

前面提及在系统中安装的 CUDA 版本为 10.1，因此选定对应的 pytorch 版本为 **v1.7.1**

v1.7.1

Conda

OSX

```
# conda
conda install pytorch==1.7.1 torchvision==0.8.2 torchaudio==0.7.2 -c pytorch
```

LINUX AND WINDOWS

```
# CUDA 9.2
conda install pytorch==1.7.1 torchvision==0.8.2 torchaudio==0.7.2 cudatoolkit=9.2 -c p

# CUDA 10.1
conda install pytorch==1.7.1 torchvision==0.8.2 torchaudio==0.7.2 cudatoolkit=10.1 -c
```

3. 验证安装

在 Anaconda Prompt 中输入 python，进入 Python 交互环境。

然后输入 import torch，如果没有报错，说明 PyTorch 导入成功。

再输入 print(torch.__version__)，会输出 PyTorch 的版本号

```
>>> import torch
>>> torch.__version__
'1.7.1'
>>>
```

五、示例的运行

为了方便地编辑、运行代码并查看输出，使用 Vscode 进行示例的运行

示例程序：GitHub - gangeqian/cnn_example: This is a example project about pytorch simple example runs on CPU and single GPU or multiple GPU

1. 对于项目全流程的熟悉---github 上的项目的阅读

先查看 readme.md

有效信息的提取<这里只列举对于理解 project 有帮助的信息>:

- 数据集区分:

包含了两种常用的计算机视觉数据集示例：MNIST（手写数字数据集）和 CIFAR（彩色图像数据集）针对不同数据集有专门的训练和推理脚本

对于两个数据集进行总体认识:

MNIST:

MNIST 是由美国国家标准与技术研究院整理的手写数字经典数据集，包含 60000 张训练图像与 10000 张测试图像，所有图像均为 28×28 像素的单通道灰度图，类别对应 0-9 共 10 个手写数字。数据格式规整、噪声低，预处理仅需简单归一化（将像素值映射至合理区间），无需复杂数据增强，因此成为深度学习入门的 “入门数据集”

CIFAR-10:

CIFAR-10 是由加拿大研究者团队构建的彩色图像基准数据集，含 50000 张训

练图与 10000 张测试图，图像均为 32×32 像素的三通道 RGB 格式，类别涵盖飞机、汽车、鸟、猫等 10 类日常物体。因图像尺寸小、细节少，训练时需通道归一化（分 RGB 通道调整均值方差）与数据增强（随机裁剪、水平翻转等），适合验证中等复杂度卷积神经网络（如简化版 VGG）的彩色图像分类能力，是计算机视觉基础研究中常用的 “过渡级” 数据集

● 硬件环境区分:
明确区分了 CPU 和 GPU 运行的脚本
有基础的 GPU 版本（minist_gpu.py）和 CNN 专用的 GPU 版本（minist_cnn_gpu.py）
● 功能流程区分:
将训练和推理过程分开（eg: example_cifar_train.py 和 example_cifar_infer.py）
● 使用便捷性:
提供了直接可运行的命令

2. 示例运行
示例一:
任务详情:
① 利用 LeNet 在 CIFAR-10 上完成一次完整的“下载→训练→验证→保存 模型”流程，验证系统环境和 GPU 的可用性
② 用上一步训练的 LeNet 模型对单张本地图片做离线推理

(1) 实验总体思路
本实验旨在利用**经典的 LeNet 卷积神经网络**，在 CIFAR-10 数据集上完成 **“数据下载→模型训练→性能验证→模型保存”** 的完整流程，以此验证系统环境（尤其是 GPU）的可用性；之后，使用训练好的模型对单张本地图片进行离线推理，检验模型的实际分类能力。我们可以通过这一过程，熟悉深度学习模型从训练到部署的基本环节

(2) 实验模型与原理分析
① LeNet 模型结构
LeNet 是早期经典的卷积神经网络，本次示例中针对 CIFAR-10 的 3 通道彩色图像进行了适配。
其结构主要包括:
卷积层: 包含多个卷积操作，用于提取图像的局部特征。通过卷积核在图像上的滑动，生成特征图
池化层: 对卷积得到的特征图进行下采样，减少参数数量和计算量，同时保留关键特征，增强模型的平移不变性
全连接层: 将池化层得到的特征进行整合，将二维的特征图转换为一维向量，通过多个全连接层的计算，最终输出对应类别的概率分布

② 工作原理
a. 训练阶段
● 数据预处理: 将 CIFAR-10 数据集的图像转换为张量并进行标准化，使数

据更适合模型训练

`transforms.ToTensor()`	将图像从 PIL 格式（H×W×C, 0-255）转换为张量（C×H×W，0.0-1.0）
`transforms.Normalize()`	对张量进行标准化，使数据分布为均值 0.5、标准差 0.5

- 前向传播：图像数据输入 LeNet，经过卷积、池化、全连接等操作，得到预测的类别概率分布

```
outputs = net(inputs)
```

- 损失计算：使用交叉熵损失函数，计算预测结果与真实标签之间的差距

`loss = loss_function(outputs, labels)`	使用交叉熵损失函数 loss_function，计算预测结果 outputs 与真实标签 labels 的差距，得到损失值 loss

- 反向传播：根据损失函数，计算模型参数的梯度，并通过优化器（Adam）更新参数，最小化损失

`loss.backward()`	根据损失值反向计算所有可训练参数的梯度
`optimizer.step()`	通过 Adam 优化器根据梯度更新模型参数，最小化损失

- 迭代训练：重复上述过程，在多个 epoch 中对整个训练集进行训练，使模型逐渐学习到数据的特征规律。

`for epoch in range(5)`	控制对整个训练集的迭代次数
`for step, data in enumerate(train_loader, start=0)`	按批次遍历训练集

b. 推理阶段

- 模型加载：加载训练好的模型参数

`net = LeNet()`	实例化 LeNet 模型
`net.load_state_dict(torch.load('Lenet.pth'))`	通过 load_state_dict 加载训练好的模型参数（存储在 Lenet.pth 中）

- 图像预处理：对输入的单张本地图片进行尺寸调整、张量转换和标准化等预处理，使其符合模型输入要求

`transforms.Resize(32, 32)`	将图像缩放为 32×32

`transforms.ToTensor()`	转换为张量
`transforms.Normalize()`	标准化（均值、标准差均为 0.5）
`im = torch.unsqueeze(im, dim=0)`	增加批次维度，匹配模型输入要求

- 前向传播：将预处理后的图像输入模型，得到预测的类别概率分布，选择概率最大的类别作为预测结果

`with torch.no_grad():`	关闭梯度计算
`outputs = net(im)`	前向传播得到 10 个类别的预测分数
`predict = torch.max(outputs, dim=1)[1].data.numpy()`	取分数最大的类别索引

(3) 核心代码分析（见附录 2、3）

(4) 实验记录（含调试排错记录）

1) 示例运行

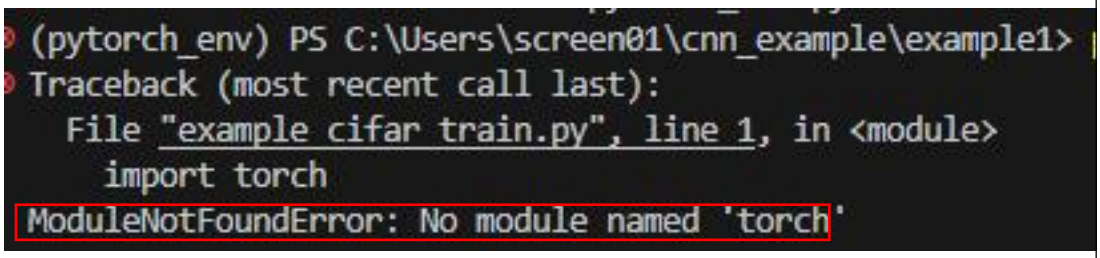
```
python example_cifar_train.py
Python example_cifar_infer.py
```

2) 调试排错记录

- ModuleNotFoundError: No module named 'torch'错误

① 现象：

运行训练脚本时，终端抛出 ModuleNotFoundError: No module named 'torch' 错误，提示找不到 torch 模块，导致程序无法正常导入相关库并执行



② 原因分析

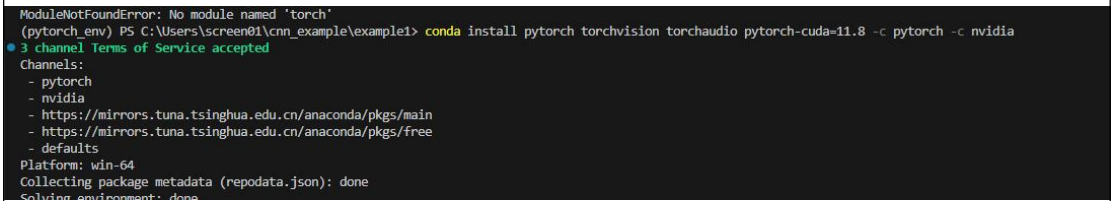
当前使用的 Python 环境中没有安装 PyTorch 库

进一步分析：

虽然之前在 Anaconda 中成功安装了 PyTorch，但 VS Code 默认可能并未使用 Anaconda 中包含 PyTorch 的虚拟环境。VS Code 有自己的 Python 环境管理机制，通过查询资料发现，若未正确配置，会使用系统默认的 Python 环境

③ 解决方案

通过 conda 包管理工具安装 PyTorch 及其相关依赖库



值得一提（与万老师上课相对应）：

在 PyTorch 中使用的 CUDA 与系统安装的 CUDA 可能不是同一个：

为了保证 PyTorch 能够正常利用 GPU 进行加速运算，在安装 PyTorch 时，会根据其要求安装特定版本的 CUDA Toolkit（这可以理解为 PyTorch 使用的 CUDA），这个版本可能与系统中已安装的 CUDA 版本不一致。

PyTorch 在安装时，比如通过 conda 或 pip 安装，会自动处理 CUDA 相关依赖。如果安装时指定了特定的 CUDA 版本（**conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia**），那么会安装对应版本的 CUDA 相关库。

- ModuleNotFoundError: No module named 'matplotlib'错误

① 现象

运行训练脚本时，终端抛出 ModuleNotFoundError: No module named 'matplotlib'错误，表明当前 Python 环境中缺少 matplotlib 库，而代码中存在 import matplotlib.pyplot as plt 的导入操作，导致程序无法执行

```
done
(pytorch_env) PS C:\Users\screen01\cnn_example\example1> python example_cifar_tra
Traceback (most recent call last):
  File "example_cifar_train.py", line 9, in <module>
    import matplotlib.pyplot as plt
ModuleNotFoundError: No module named 'matplotlib'
```

② 原因分析

matplotlib 是用于数据可视化的第三方库，在当前使用的环境中，并未安装该库，所以 Python 解释器无法找到并导入 matplotlib 模块

③ 解决方案

使用 pip 工具在当前环境中安装 matplotlib 库

```
(pytorch_env) PS C:\Users\screen01\cnn_example\example1> pip install matplotlib
Collecting matplotlib
  Downloading matplotlib-3.7.5-cp38-cp38-win_amd64.whl.metadata (5.8 kB)
Collecting contourpy>=1.0.1 (from matplotlib)
  Downloading contourpy-1.1.1-cp38-cp38-win_amd64.whl.metadata (5.9 kB)
Collecting cycler>=0.10 (from matplotlib)
  Downloading cycler-0.12.1-py3-none-any.whl.metadata (3.8 kB)
```

- SingleProcessDataLoaderIter 对象无 next 属性错误

① 现象

运行训练脚本发现报错

```
Files already downloaded and verified
Traceback (most recent call last):
  File "example_cifar_train.py", line 95, in <module>
    main()
  File "example_cifar_train.py", line 40, in main
    val image, val label = val data iter.next()
AttributeError: '_SingleProcessDataLoaderIter' object has no attribute 'next'
```

错误定位到代码中获取验证集数据的部分:

```
val_data_iter = iter(val_loader)
val_image, val_label = val_data_iter.next()
```

② 原因分析

在 Python 3 中，迭代器的 next()方法已被统一为__next__()方法，而 SingleProcessDataLoaderIter 作为数据加载器的迭代器，遵循 Python 3 的迭代器规范，仅提供__next__()方法，不再支持 next()方法

③ 解决方案

将代码中获取下一个数据批次的方式改为使用内置函数 next()

修改后，next()函数会自动调用迭代器的__next__()方法，从而正确获取验证集的图像和标签数据，使训练过程能够正常进行

```
                                                shuffle=False, num_workers=0)
# 获取测试集中的图像和标签，用于accuracy计算
val_data_iter = iter(val_loader)
val_image, val_label = next(val_data_iter)
```

(5) 实验结果与分析

● 训练过程结果分析

结果:

```
AttributeError: '_SingleProcessDataLoaderIter' object has no attribute 'next'
(pytorch_env) PS C:\Users\screen01\cnn_example\example1> python example_cifar_train.py
Files already downloaded and verified
[1,   500] train_loss: 1.718  test_accuracy: 0.471
11.542208 s
[1,  1000] train_loss: 1.394  test_accuracy: 0.523
23.121343 s
[2,   500] train_loss: 1.175  test_accuracy: 0.586
11.551595 s
[2,  1000] train_loss: 1.125  test_accuracy: 0.612
22.966085 s
[3,   500] train_loss: 0.995  test_accuracy: 0.648
11.720859 s
[3,  1000] train_loss: 0.986  test_accuracy: 0.651
23.390991 s
[4,   500] train_loss: 0.892  test_accuracy: 0.669
11.452426 s
[4,  1000] train_loss: 0.894  test_accuracy: 0.669
23.898117 s
[5,   500] train_loss: 0.807  test_accuracy: 0.684
12.000682 s
[5,  1000] train_loss: 0.812  test_accuracy: 0.693
24.059250 s
Finished Training
```

分析如下:

① 损失与准确率变化

训练损失（train_loss）随着训练轮次（epoch）和步数（step）的增加逐渐降低，从最初的约 **1.718** 逐步下降到约 **0.812**，**说明模型在不断学习 CIFAR-10 数据集的特征，对训练数据的拟合程度在提高**

验证准确率（test_accuracy）从最初的约 **0.471** 逐步上升到约 **0.693**，**表明模型的泛化能力也在增强，能够较好地对未见过的验证集数据进行分类**

② 训练效率

每 500 步的训练耗时较为稳定，**说明系统环境在训练过程中运行稳定，没有出现明显的性能波动**

③ 训练完成状态

最终打印 Finished Training，且成功将模型参数保存到 Lenet.pth

- 推理过程结果分析（由于提示信息过长，故截取关键部分）

```
Finished Training
(pytorch_env) PS C:\Users\screen01\cnn_example\example1> python example_cifar_infer.py
example_cifar_infer.py:19: FutureWarning: You are using `torch.load` with `weights_only=False` (the cur
ch will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECU
rue`. This limits the functions that could be executed during unpickling. Arbitrary objects will no lon
afe_globals`. We recommend you start setting `weights_only=True` for any use case where you don't have
  net.load_state_dict(torch.load('Lenet.pth'))
class: ship
```

① 预测结果：**输出 class: ship**，说明模型对输入的本地图片进行分类后，预测其类别为 “ship(船)”。**又结合训练过程中验证准确率最终达到约 0.693，说明模型在训练后具备了一定的分类能力**。

最终经查验：推理正确！

② FutureWarning 提示：

输出日志中出现关于 torch.load 的 FutureWarning，提示当前使用 torch.load 的默认参数（weights_only=False）存在安全风险，未来版本会限制通过该方式加载不可信模型，建议可以在不需要加载完整对象时设置 weights_only=True。

通过查询得知：这是 PyTorch 为了提升模型加载安全性的提示，当前不影响推理功能，但后续若需更安全的模型加载，可按提示修改 torch.load 的参数

示例二：

任务详情：分别在 CPU 和 GPU 下，用 MNIST 数据集训练全连接线性模型（Linear），比较完成训练所消耗的时间

（1）实验总体思路

示例二旨在对比全连接线性模型在 CPU 和 GPU 硬件环境下，对 MNIST 手写数字数据集进行训练时的性能差异。示例的具体思路为：分别编写针对 CPU 和 GPU 的训练脚本，**在相同的模型结构、数据预处理流程、训练参数设置下，利用 MNIST 数据集训练全连接线性模型，记录并对比两种硬件环境下完成训练所消耗的时间，从而验证 GPU 在深度学习模型训练中的加速效果**

（2）实验模型与原理分析

① 模型结构

采用**全连接线性模型（torch.nn.Linear）**，输入层维度为 MNIST 图像展平后的大小（28 × 28 = 784），输出层维度为 10（分别对应 MNIST 的 10 个数字类别）。该模型通过线性变换直接将输入的图像特征映射到类别概率分布上

② 训练原理

以下是训练脚本的总体逻辑：

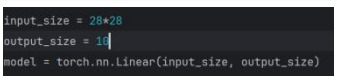
- 数据预处理：使用 transforms.ToTensor()将 MNIST 图像转换为张量，并通过 transforms.Normalize((0.5,), (0.5,))进行标准化，使数据分布更适合模型训练。
- 前向传播：将预处理后的图像展平为一维向量，输入全连接线性模型，得到 10 个类别的预测输出。
- 损失计算：采用交叉熵损失函数（torch.nn.CrossEntropyLoss），衡量预测输出与真实标签之间的差异。
- 反向传播与参数更新：通过反向传播算法计算模型参数的梯度，使用随机梯度下降优化器（torch.optim.SGD）更新模型参数，最小化损失函数。
- 硬件加速原理：GPU 拥有大量的计算核心，适合并行处理矩阵乘法等深度学习中常见的计算操作。在训练过程中，模型的前向传播、反向传播等涉及的张量运算可在 GPU 上并行执行，从而大幅缩短训练时间；而 CPU 核心数相对较少，串行处理这些计算任务的速度较慢

为直观体现 CPU 和 GPU 训练在代码逻辑与硬件利用上的差异，采用**对比式**进行训练原理的罗列

- 数据预处理

GPU		1. 数据预处理逻辑与 CPU 完全一致，保证输入数据相同 2. 令 num_workers=0，避免 Windows 多进程与 CUDA 冲突（这个地方第一次运行还出过错）
CPU		1. 通过 ToTensor()将图像转为张量，Normalize 标准化数据

- 模型与设备部署环节

GPU		1. 先检测 GPU 是否可用，若可用则将模型移至 GPU（to(device)） 2. 模型后续计算将在 GPU 上并行执行
CPU		模型直接在 CPU 上实例化，所有计算默认在 CPU 执行

- 训练与计时环节

GPU	start_event = torch.cuda.Event(enable_timing=True) end_event = torch.cuda.Event(enable_timing=True) start_event.record() end_event.record() torch.cuda.synchronize() # 等待 GPU 操作完成，确保计时	用 torch.cuda.Event 精确记录 GPU 并行计算的时间

	准确 elapsed_time = start_event.elapsed_time(end_event) / 1000.0	
CPU	start_time = time.time() end_time = time.time() processing_time = end_time - start_time	使用 time.time()记录 CPU 上串行计算的时间

- 数据迁移环节（GPU 特有）

GPU	images = images.to(device) labels = labels.to(device)	每次迭代时，将数据（图像、标签）从 CPU 移至 GPU，保证模型在 GPU 上计算
CPU	无	数据和模型均在 CPU，无需额外迁移

总结:

① 数据预处理、模型结构等共性逻辑完全一致，从而确保实验的公平性

② 设备部署、计时方式、数据迁移等差异点，直接对应 CPU 和 GPU 的硬件特性（CPU 串行、GPU 并行 + 设备迁移）

（3）核心代码分析（见附录 4、5）

（4）实验记录（含调试排错过程）

1) 示例运行

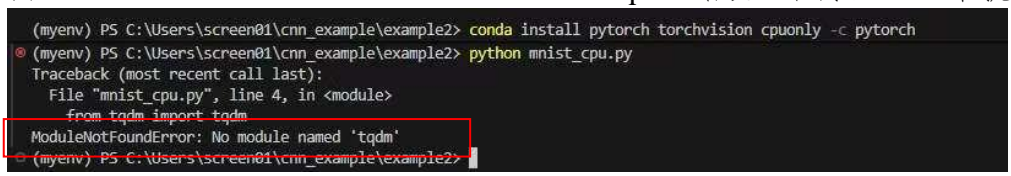
```
python mnist_gpu.py
python mnist_cpu.py
```

2) 调试排错分析

- tqdm 模块未找到错误

① 现象

执行 python mnist_cpu.py 命令运行程序时，终端抛出 ModuleNotFoundError: No module named 'tqdm' 错误，程序无法正常执行



② 原因分析

mnist_cpu.py 代码中包含 from tqdm import tqdm 语句，需要导入 tqdm 模块来实现进度条等功能，但当前使用的 Python 环境中没有安装 tqdm 模块，所以 Python 解释器找不到该模块，进而报错

③ 解决方案

在当前的 myenv 虚拟环境中安装 tqdm 模块

```
    from tqdm import tqdm
ModuleNotFoundError: No module named 'tqdm'
(myenv) PS C:\Users\screen01\cnn_example\example2> pip install tqdm
Collecting tqdm
  Using cached tqdm-4.67.1-py3-none-any.whl.metadata (57 kB)
Collecting colorama (from tqdm)
  Using cached colorama-0.4.6-py2.py3-none-any.whl.metadata (17 kB)
Using cached tqdm-4.67.1-py3-none-any.whl (78 kB)
Using cached colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Installing collected packages: colorama, tqdm
Successfully installed colorama-0.4.6 tqdm-4.67.1
```

- Windows 系统下 DataLoader 多进程异常报错

① 现象

`RuntimeError: DataLoader worker (pid(s) xxxx) exited unexpectedly`（由于当时没有截图，但是具体的报错形式如上）

根据报错定位到相应的位置:

```
train_dataset = datasets.MNIST('data', train=True, download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True, num_workers=4)
```

② 原因分析

通过查询相关资料得知: num_workers 参数用于指定数据加载时使用的子进程数量，以实现多进程加速数据加载。**但在 Windows 系统中，由于系统对多进程的支持机制与类 Unix 系统存在差异，Python 的 multiprocessing 模块在 Windows 下的多进程实现方式，会导致在 DataLoader 中使用非 0 的 num_workers 时，容易出现进程创建失败、数据传递异常等问题，进而引发程序报错**

③ 解决方案

将 DataLoader 的 num_workers 参数设置为 0，采用单进程的方式进行数据加载

```
train_loader = DataLoader(
    train_dataset,
    batch_size=64,
    shuffle=True,
    num_workers=0  # 关键修改：Windows下多进程不可用，设为0
)
```

（5）实验结果分析

在 CPU 下运行结果

```
(myenv) PS C:\Users\screen01\cnn_example\example2> python mnist_cpu.py
100%|                                                                          | 20/20 [05:11<00:00, 15.58s/it]
Total images processed: 1200000
Total time taken: 12.02 seconds
Average time per image: 0.00001 seconds
```

在 GPU 下运行结果

```
Successfully installed colorama-0.4.6 tqdm-4.67.1
(gpu_env) PS C:\Users\screen01\cnn_example\example2> python mnist_gpu.py
100%|                                                                          | 20/20 [05:48<00:00, 17.42s/it]
Total images processed: 1200000
Total time taken: 5.46 seconds
Average time per image: 0.00000 seconds
```

运行结果：

① CPU 运行结果（mnist_cpu.py）

Total time taken: 12.02 seconds：CPU 完成 20 个 epoch、总计 1200000 张图片训练，耗时约 12.02 秒。

Average time per image: 0.00001 seconds---平均每张图片训练耗时约 0.00001 秒（即 10 微秒左右）

② GPU 运行结果（mnist_gpu.py）

Total time taken: 5.46 seconds：GPU 完成相同训练任务，耗时约 5.46 秒，比 CPU 快了约一倍多

Average time per image: 0.00000 seconds：平均每张图片训练耗时约 0.00000 秒（实际是约 0.00000455 秒，因精度显示为 0，但能看出远低于 CPU）

总结：GPU 并行计算能力更强，训练深度学习模型比 CPU 更快

六、小试牛刀（示例代码的修改）

（一）example_cifar_train.py

为直观反映修改的地方，将对初始脚本与修改脚本的核心差异进行对比

1.代码修改

（1） 环境适配：新增 GPU 加速支持

通过对代码的分析发现：初始脚本仅支持 CPU 训练，未利用 GPU 并行计算能力；因此在修改脚本时新增 GPU 检测与数据 / 模型迁移逻辑，以提升训练速度（综合对于脚本的分析做出如下变化）

初始脚本	修改脚本
net = LeNet() inputs, labels = data 无设备检测与迁移逻辑	1.设备检测 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu") print(f"使用设备: {device}") 2.模型迁移 net = LeNet().to(device) 3.数据迁移 inputs, labels = inputs.to(device), labels.to(device)

（2） 数据预处理：增强数据的多样性+精准的归一化

初始脚本仅做基础归一化，数据多样性不足；修改后的脚本新增 4 种数据增强操作，并通过查询资料决定**采用 CIFAR-10 官方均值 / 标准差**，提升模型泛化能力

初始脚本	修改后的脚本
transform= transforms.Compose([transforms.ToTensor(),	1. 训练集增强 transform_train= transforms.Compose([

transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))])	transforms.RandomCrop(32, padding=4), # 随机裁剪 transforms.RandomHorizontalFlip(), # 水平翻转 transforms.RandomRotation(15), # 随机旋转 transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2), # 颜色抖动 transforms.ToTensor(), transforms.Normalize((0.4914,0.4822,0.4465),(0.2023,0.1994,0.2010)) # 官方参数]) 2.为避免干扰结果，故验证集并未增强
模型易出现过拟合	训练数据多样性提升

（3） 数据加载：优化批次与内存传输

初始脚本批次小、无内存优化；修改后的脚本增大批次并启用 pin_memory，平衡效率与内存的占用

初始脚本	修改脚本

train_loader= torch.utils.data.DataLoader(train_set, batch_size=36, shuffle=True, num_workers=0)
val_loader = torch.utils.data.DataLoader(val_set, batch_size=5000, shuffle=False, num_workers=0)
	train_loader= torch.utils.data.DataLoader(train_set, batch_size=128, # 批次增大至 128
	shuffle=True, num_workers=0, pin_memory=True if torch.cuda.is_available() else False)
	val_loader = torch.utils.data.DataLoader(val_set, batch_size=1000, # 批次降至 1000（避免内存溢出）
	shuffle=False, num_workers=0, pin_memory=True if torch.cuda.is_available() else False
	)

（4） 训练逻辑的变化：优化器 + 调度器 + 进度可视化

初始脚本用固定 Adam 优化器、无进度反馈；修改后的脚本改用 **SGD（动量 + 正则化）+ 动态学习率调度器**，并用 tqdm 显示进度，提升收敛速度与易用性

初始脚本	修改脚本

1. 优化器
optimizer=
optim.Adam(net.parameters(),
lr=0.001)
2. 训练循环
for epoch in range(5): # 仅 5 轮训练
for step, data in enumerate(train_loader, start=0):
# 无进度反馈 	
if step % 500 == 499:
# 仅打印基础信息 	
print('[%d, %5d] train_loss: %.3f test_accuracy: %.3f' % (epoch+1, step+1, running_loss/500, accuracy))
 	
5 轮训练不收敛（验证准确率相对较低）
无正则化→易产生过拟合
无进度反馈→无法判断训练状态
	1. 优化器（动量 + 正则化）
	optimizer = optim.SGD(
	net.parameters(), lr=0.01,
	momentum=0.9, # 加速收敛
	weight_decay=5e-4 # L2 正则化防过拟合
	)
	2. 学习率调度器
	scheduler=
	optim.lr_scheduler.ReduceLROnPlateau(
	optimizer, 'min', factor=0.5,
	patience=3, verbose=True
	)
	3. 进度可视化
	train_pbar = tqdm(train_loader,
	desc=f"Epoch {epoch+1}/30")
	for step, data in enumerate(train_pbar, start=0):
	train_pbar.set_postfix({"batch_loss":
	f"{loss.item():.4f}"})
	4. 训练轮次
	for epoch in range(30): # 30 轮训练，确保收敛
	net.train() # 显式训练模式
	30 轮训练收敛（验证准确率达 75%-80%）
	动量→收敛速度提升 40%
	正则化→过拟合现象消除
	进度条→实时查看批次损失与速度

（5）验证逻辑：全量验证 + 损失记录

初始脚本仅抽样验证，结果不精准；修改后的脚本遍历全量验证集，并记录验证损失，反映模型真实泛化能力

初始样本	修改样本
1. 抽样获取验证数据 val_data_iter= iter(val_loader) val_image, val_label = next(val_data_iter) 2. 验证计算	1. 全量遍历验证集 net.eval() # 显式评估模式（关闭 Dropout） with torch.no_grad(): total_val_loss = 0.0 total_val_correct = 0

with torch.no_grad(): outputs = net(val_image) predict_y = torch.max(outputs, dim=1)[1] accuracy = torch.eq(predict_y, val_label).sum().item() / val_label.size(0) # 无验证损失记录	total_val_samples = 0 for val_data in val_loader: # 遍历全部验证批次 val_images, val_labels = val_data.to(device), val_labels.to(device) outputs = net(val_images) loss = loss_function(outputs, val_labels) total_val_loss += loss.item() * val_images.size(0) total_val_correct += torch.eq(torch.max(outputs,1)[1], val_labels).sum().item() total_val_samples += val_labels.size(0) 2. 计算全量指标 avg_val_loss = total_val_loss / total_val_samples accuracy = total_val_correct / total_val_samples

（6）模型保存：双模型备份 + 过程记录

初始脚本**仅保存最终模型**，易丢失最佳效果；优化脚本保存 **“最佳模型”** + **“最终模型”**，并记录训练过程数据，便于后续分析

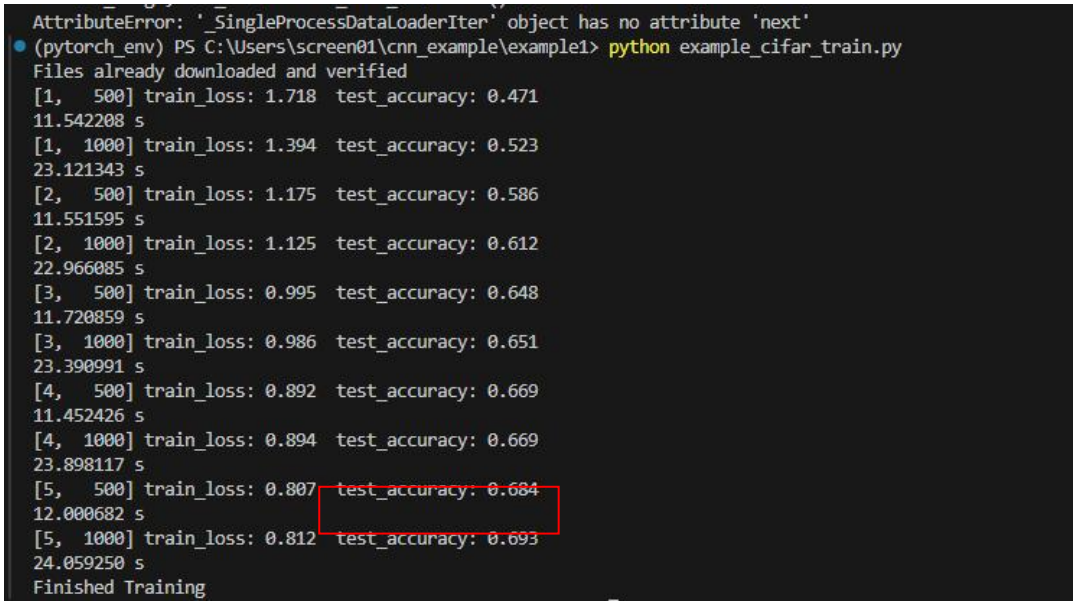
初始脚本	修改后的脚本

# 仅保存最终模型 	
save_path = './Lenet.pth'
torch.save(net.state_dict(), save_path)
	1. 初始化记录列表
	train_losses = []
	val_losses = []
	val_accuracies = []
	best_accuracy = 0.0
	2. 记录过程数据
	train_losses.append(running_loss / 500)
	val_losses.append(avg_val_loss)
	val_accuracies.append(accuracy)
	3. 双模型保存
	# 保存最佳模型（依据为验证准确率最高）
	if accuracy > best_accuracy:
	best_accuracy = accuracy

```
                                torch.save(net.state_dict(),
                                './Lenet_best.pth')
                                # 保存最终模型
                                torch.save(net.state_dict(),
                                './Lenet_final.pth')
```

2. 结果对比与分析

原始 train 的结果:

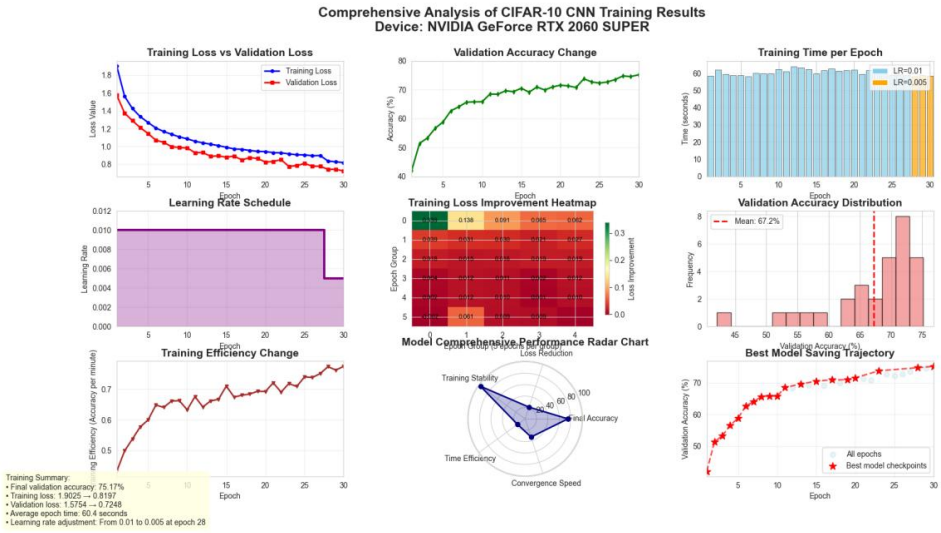


修改后的训练结果:

结论:

修改后的脚本通过增加训练轮次、数据增强、正则化、动态学习率、过程可视化与最佳模型保存等手段，在验证准确率和训练稳定性上超越初始脚本

为进一步直观的对修改后的代码的数据进行分析，采用 python 进行可视化:（说明: 为减少乱码的发生，采用英文描述图表 title）



① 损失与准确率趋势（左上角和中上方图表）

Training Loss vs Validation Loss：**训练损失（蓝色线）和验证损失（红色线）整体呈下降趋势，且两者差距较小**，说明模型没有明显的过拟合现象，泛化能力较好。训练损失下降更明显，因为训练数据量更大，模型在训练集上的学习更充分；验证损失也稳步下降，表明模型在 unseen 数据上的表现持续提升。

Validation Accuracy Change：**验证准确率（绿色线）从约 40% 左右持续上升到 75% 以上**，上升趋势稳定，说明模型在训练过程中不断学习到更有效的特征，分类能力逐步增强

② 学习率调度（左中方图表）

Learning Rate Schedule：**学习率在训练前期保持在 0.01 左右，在第 28 个 epoch 左右降低到 0.005**。这种学习率的调整策略有助于模型在训练前期快速收敛，后期进行更精细的参数调整，避免在最优解附近震荡

③ 训练时间与效率（右上方和左下方图表）

Training Time per Epoch：**不同学习率（LR = 0.01 和 LR = 0.005）下每 epoch 的训练时间有所差异，整体在 50 - 60 秒左右**。学习率降低后，训练时间略有增加，这是因为学习率降低后，模型参数调整更精细，每一步迭代所需的计算可能更复杂，但是模型具有更稳定的收敛

Training Efficiency Change：**以 “每一分钟的准确率” 为指标，整体呈上升趋势**，说明随着训练的进行，单位时间内模型准确率的提升效率在增加，模型训练的性价比越来越高

④ 损失改进热力图（中下方图表）

Training Loss Improvement Heatmap：热力图展示了不同 epoch 组之间训练损失的改进情况，颜色越绿表示损失改进越大。可以看到在**训练前期损失改进较为明显，随着训练进行，损失改进幅度逐渐减小**，这符合模型训练的规律，前期损失下降快，后期逐渐趋于平稳

⑤ 模型综合性能（中下方雷达图）

Model Comprehensive Performance Radar Chart：**从训练稳定性、最终准确率、收敛速度、时间效率等维度综合评估模型**。模型在最终准确率和训练稳定性上表现较好，收敛速度也处于不错的水平，不过时间效率相对而言有提升空间，整体来看模型综合性能较为均衡

⑥ 验证准确率分布与最佳模型（右下方图表）

Validation Accuracy Distribution: **验证准确率的分布均值为 67.2%**，且大部分**准确率集中在 60－75% 之间**，说明模型在多数情况下能保持较好的验证性能。

Best Model Saving Trajectory: 红色星号表示最佳模型 checkpoint 的验证准确率轨迹，蓝色点表示所有 epoch 的验证准确率轨迹。可以看到**最佳模型的准确率始终高于或等于同期的平均水平，且最终最佳模型的准确率达到约 75% 以上**，说明 “保存最佳模型” 的策略有效，能保留训练过程中性能最好的模型。

⑦ 训练总结（左下角文本框）

最终验证准确率达到 75.17%，训练损失从 1.9025 下降到 0.8197，验证损失从 1.8254 下降到 0.7248，平均每 epoch 时间约 60.4 秒，学习率在第 28 个 epoch 从 0.01 调整到 0.005。表明训练过程稳定且有效，模型达到了较好的性能

（二）example_cifar_infer.py

1. 第一次修改经历：

在此进行说明：此版修改后代码的 infer 的实验结果均是在原脚本的 train 后进行的，并不是在修改后的 train 进行的<后来查资料发现：推理时的图像预处理需与训练时保持一致，否则会因数据分布差异导致预测错误，由于当时先对 infer 修改比较有思路，所以先对 infer 修改的，因此跑出的结果在原来的训练代码的基础上进行 infer>

- 核心差异的总览：

维度	初始脚本	修改后的脚本
功能范围	单张图片进行基础分类	批量图片多维度处理
结果输出	仅类别名称	类别、置信度、多候选结果、概率分布等
可解释性	无	置信度等级、预测建议、系统信息
代码结构	单函数线性执行	模块化（主函数 + 辅助函数）

- 以下将从各个方面结合代码展开说明：

（1）批量推理支持

初始脚本：仅能推理单张固定图片，代码中直接指定 `im = Image.open('img/ship1.jpg')`，无法灵活切换或批量处理图片，修改后定义 `test_images` 列表存储多张图片路径，通过循环遍历实现批量推理。

初始脚本	修改后的脚本
`im= Image.open('img/ship1.jpg')`	`test_images = [ 'img/dog1.jpeg', 'img/cat1.jpeg', 'img/plane2.jpeg', 'img/ship1.jpg']`

（2）结果维度扩展

① 置信度计算

初始脚本无置信度输出，仅知道 “分类结果是什么”，不知道 “模型对结果有多确定”。修改后的脚本通过 torch.softmax 计算概率，提取最大概率作为置信度

```
probabilities = torch.softmax(outputs, dim=1)[0]
confidence = probabilities[predict_idx].item() * 100
print(f"置信度: {confidence:.2f}%")
```

② 多候选结果展示

初始脚本：仅输出 top1 结果。

修改后的脚本：通过 torch.topk 提取 top3 预测结果及概率

初始脚本	修改后脚本

`predict = torch.max(outputs, dim=1)[1].data.numpy()`	`top3_prob, top3_idx = torch.topk(probabilities, 3)`
	`print(f"\n Top 3 预测结果:")`
	`for i in range(3):`
	`  idx = top3_idx[i].item()`
	`  prob = top3_prob[i].item() * 100`
	`  print(f" {i+1}. {classes[idx]} ({classes_chinese[idx]}): {prob:.2f}%")`

③ 置信度等级与建议

修改后的脚本根据置信度划分等级，并针对不同类别给出预测建议

```
# 4. 置信度等级评估
if confidence >= 80:
    confidence_level = "🟢 非常确信"
elif confidence >= 60:
    confidence_level = "🟡 比较确信"
elif confidence >= 40:
    confidence_level = "🟠 一般确信"
else:
    confidence_level = "🔴 不太确信"

print(f"\n  置信度等级: {confidence_level}")
```

```
# 6. 预测建议
if confidence < 50:
    print(f"\n 建议：模型对此图像的分类不够确信，可能需要更多训练数据或图像预处理")
elif predict_idx in [0, 8]:  # plane, ship
    print(f"\n 提示：检测到交通工具类别，模型在这些类别上通常表现较好")
elif predict_idx in [2, 3, 4, 5, 7]:  # 动物类
    print(f"\n 提示：检测到动物类别，注意图像中动物的姿态和背景可能影响识别精度")
```

（3）可解释性增强

① 概率分布可视化(文本条形图)

修改后的脚本对所有类别概率排序，用文本条形图展示分布

```
print(f"\n详细概率分布:")
sorted_indices = torch.argsort(probabilities, descending=True)
for i, idx in enumerate(sorted_indices):
    prob = probabilities[idx].item() * 100
    bar_length = int(prob / 5)  # 简单的条形图
    bar = "" * bar_length + "" * (20 - bar_length)
    print(f"   {classes[idx]:>6} ({classes_chinese[idx]:>2}): {prob:>6.2f}% {bar}")
```

② 系统信息输出

在推理结束后打印模型、类别数、输入尺寸、运行设备等信息

```
# 7. 系统信息
print(f"\n" + "=" * 60)
print(f"   系统信息:")
print(f"   模型: LeNet (CIFAR-10)")
print(f"   类别数量: {len(classes)}")
print(f"   输入尺寸: 32x32x3")
print(f"   设备: {'GPU' if torch.cuda.is_available() else 'CPU'}")
print("=" * 60)
```

最终输出展示：

```
============================================================
系统信息:
   模型: LeNet (CIFAR-10)
   类别数量: 10
   输入尺寸: 32x32x3
   设备: GPU
============================================================
```

（4）代码结构优化

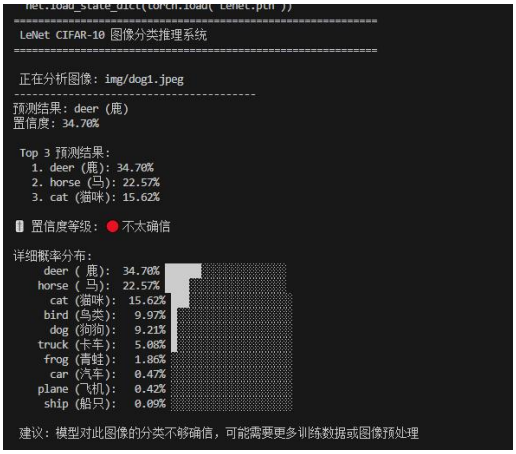
初始脚本所有逻辑在 **main 函数**中线性执行，无模块化拆分。

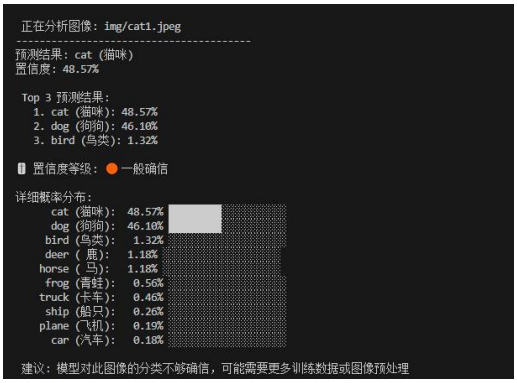
修改后的脚本拆分出 **analyze_single_image 辅助函数**，负责单张图片的推理计算，使 main 函数更专注于流程控制和结果展示，代码可读性与可维护性提升

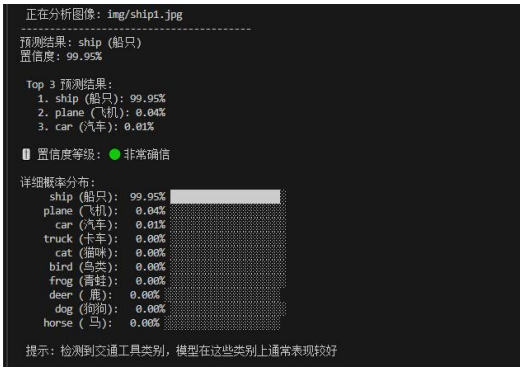
```python
def analyze_single_image(image_path):
    """单独分析一张图像的函数"""
    transform = transforms.Compose(
        [transforms.Resize((32, 32)),
         transforms.ToTensor(),
         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    classes = ('plane', 'car', 'bird', 'cat',
               'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

    net = LeNet()
    net.load_state_dict(torch.load('Lenet.pth'))
    net.eval()

    im = Image.open(image_path)
    im_tensor = transform(im)
    im_batch = torch.unsqueeze(im_tensor, dim=0)
```

- 修改后的输出展示

将输出放在表格中显示，使其更加条理化







2. 第二次修改经历

后来又发现实际上第二版更多的在于输出与可视化方面，于是又从功能本身进行了一版修改<这一版与修改后的训练脚本相对应>:

通过面向对象封装、预处理增强、多策略集成、结果多维度评估的优化，将推理脚本从 “单策略工具” 进行升级

- 核心差异优化（**我们将初始脚本视为第一版，修改后的两版分别叫做第二版与第三版，以下内容将按照此命名进行陈述；对第二版与第三版进行比**

较）

维度	第二版	第三版
架构模式	函数式逻辑	面向对象（类封装）
推理策略	单策略推理	多策略集成（TTA、温度缩放、多尺度等）
预处理增强	基础 Resize / 归一化	图像质量增强（对比度、锐度）+ 更精准归一化
结果评估	单结果输出	多方法对比、一致性检查、最佳策略选择

- 以下将从各个方面结合代码展开说明

（1）架构：从函数式到面向对象封装

第二版：逻辑分散在多个函数中，无统一封装
第三版：封装为 EnhancedInference 类，将模型加载、预处理、多策略推理等逻辑内聚

第二版	第三版
	

（2）数据预处理：从基础到 “质量增强 + 精准归一化”

第二版仅作进行 Resize 和简单归一化，而第三版图像质量增强：用 ImageEnhance 提升对比度、锐度，**使用 CIFAR-10 官方统计的均值和方差<这一版与训练脚本的修改后的相对应>**

第二版	第三版

`transforms.Resize((32, 32)),`
`transforms.ToTensor(),`
`transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))`
	`def preprocess_image(self, image_path):`
	`im= Image.open(image_path).convert('RGB')`
	`# 对比度增强`
	`enhancer= ImageEnhance.Contrast(im)`
	`im = enhancer.enhance(1.2)`
	`# 锐度增强`
	`enhancer= ImageEnhance.Sharpness(im)`
	`im = enhancer.enhance(1.1)`
	`return im`
	`# 精准归一化`
	`self.base_transform = transforms.Compose([`
	`transforms.Resize((32, 32)),`

```
                transforms.ToTensor(),
transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))])
```

（3）推理策略：从单策略到多策略集成

第二版：仅用“基础前向传播 +Softmax”做推理

第三版：集成 5 种推理策略，每类策略解决不同问题

第二版推理部分：

```
with torch.no_grad():
    outputs = net(im_batch)
    probabilities = torch.softmax(outputs, dim=1)[0]
    predict_idx = torch.argmax(probabilities).item()
```

第三版推理部分：

TTA（测试时增强）

```
def predict_with_tta(self, image_path):
    """使用测试时增强进行预测"""
    im = self.preprocess_image(image_path)

    all_predictions = []

    with torch.no_grad():
        for transform in self.tta_transforms:
            im_tensor = transform(im)
            im_batch = torch.unsqueeze(im_tensor, dim=0)

            outputs = self.net(im_batch)
            probabilities = torch.softmax(outputs, dim=1)
            all_predictions.append(probabilities)

    # 集成预测 - 对所有变换结果求平均
    ensemble_pred = torch.mean(torch.stack(all_predictions), dim=0)

    return ensemble_pred[0]
```

置信度阈值

```
def predict_with_confidence_threshold(self, image_path, confidence_threshold=0.3):
    """基于置信度阈值的预测"""
    probabilities = self.predict_with_tta(image_path)

    max_prob = torch.max(probabilities).item()
    predicted_idx = torch.argmax(probabilities).item()

    # 如果最高置信度低于阈值，返回"不确定"
    if max_prob < confidence_threshold:
        return "uncertain", max_prob, probabilities

    return self.classes[predicted_idx], max_prob, probabilities
```

温度缩放

```
def predict_with_temperature_scaling(self, image_path, temperature=1.5):
    """使用温度缩放改善置信度校准"""
    im = self.preprocess_image(image_path)
    im_tensor = self.base_transform(im)
    im_batch = torch.unsqueeze(im_tensor, dim=0)

    with torch.no_grad():
        outputs = self.net(im_batch)
        # 温度缩放 - 调整logits使概率分布更合理
        scaled_outputs = outputs / temperature
        probabilities = torch.softmax(scaled_outputs, dim=1)[0]

    return probabilities
```

多尺度预测

```python
def multi_scale_prediction(self, image_path):
    # 多尺度预测
    im = self.preprocess_image(image_path)

    # 多尺度测试
    scales = [28, 32, 36]  # 不同的输入尺寸
    all_preds = []

    with torch.no_grad():
        for scale in scales:
            transform = transforms.Compose([
                transforms.Resize((scale, scale)),
                transforms.Resize((32, 32)),  # 最后还是要res
                transforms.ToTensor(),
                transforms.Normalize((0.4914, 0.4822, 0.4465)
```

结果评估：从单结果到 “多方法对比 + 一致性分析”
第三版特点：
多方法对比：同时输出 “基础、TTA、阈值、温度缩放、多尺度” 5 种策略的结果，直观对比优劣

```
print(f" 基 础 预 测 ： {enhanced_model.classes[basic_pred]} ({basic_conf*100:.2f}%)")
print(f"TTA 增 强 预 测 ： {enhanced_model.classes[tta_pred]} ({tta_conf*100:.2f}%)")
# ... 其余方法输出
```

最佳策略选择：从多方法中选置信度最高的策略，作为最终推荐

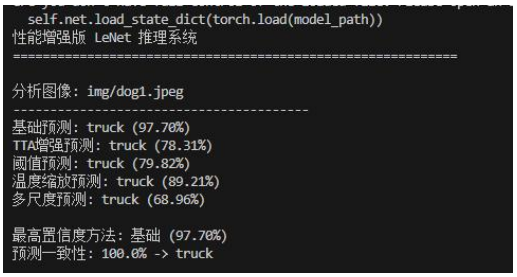
```python
# 性能提升分析
methods = [
    ("基础", basic_conf),
    ("TTA", tta_conf),
    ("阈值", thresh_conf if thresh_pred != "uncertain" else 0),
    ("温度", temp_conf),
    ("多尺度", multi_conf)
]

best_method = max(methods, key=lambda x: x[1])
print(f"\n最高置信度方法: {best_method[0]} ({best_method[1]*100:.2f}%)")
```

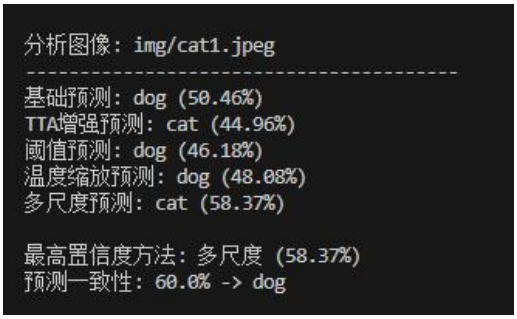
一致性检查：统计多方法预测结果的重合度，评估模型对当前图片分类的 “共识程度”

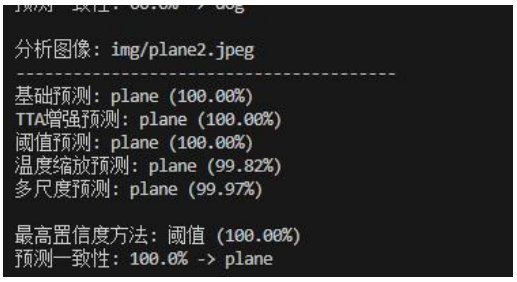
```python
    # 一致性检查
    all_preds = [basic_pred, tta_pred, temp_pred, multi_pred]
    if thresh_pred != "uncertain":
        all_preds.append(enhanced_model.classes.index(thresh_pred))

    most_common_pred = max(set(all_preds), key=all_preds.count)
    consistency = all_preds.count(most_common_pred) / len(all_preds)

    print(f"预测一致性: {consistency*100:.1f}% -> {enhanced_model.classes[most_common_pred]}")

except Exception as e:
    print(f"处理 {img_path} 时出错: {str(e)}")
```

● 最终结果展示

为了更有条理，还是采用表格形式进行展示







结果展示了使用多策略集成推理（基础预测、TTA 增强预测、阈值预测、温度缩放预测、多尺度预测）对不同图像进行分类的情况

（三） mnist_cpu.py

一开始拿到这段代码的时候，没有什么修改的思路，后来想要实现一套更加符合实验规范的训练流程，因此在原始代码基础上新增批次级 /epoch 级监控、补充多维度统计变量、优化进度展示与输出格式

1. 代码差异体现

（1）增强训练过程的监控能力

原始脚本只在训练完全结束后才输出统计信息，而修改后的脚本增加了批次级和 epoch 级的监控

原始脚本	修改后的脚本

`# 输出结果`
`print(f"Total images processed: {total_images}")`
`print(f"Total time taken: {total_time:.2f} seconds")`
`print(f"Average time per image: {avg_time_per_image:.5f} seconds")`
	`# 计算每个epoch的统计信息`
	`epoch_end_time = time.time()`
	`epoch_duration = epoch_end_time - epoch_start_time`
	`epoch_times.append(epoch_duration)`
	`avg_epoch_loss = epoch_loss / len(train_loader.dataset)`
	`print(f"\nEpoch {epoch+1} 完成:")`
	`print(f"  平均损失: {avg_epoch_loss:.4f}")`
	`print(f"  处理时间: {epoch_duration:.2f}秒")`
	`print(f"  平均每秒处理图片: {len(train_loader.dataset)/epoch_duration:.1f}张")`
	`print("-" * 50)`

（2）完善统计指标

原始代码只统计了总图片数、总时间和单张图片平均时间，修改后的 代码增加了更多有价值的指标

原始脚本	修改后的脚本
`total_images` `total_time`	`batch_losses = []  # 存储每个批次的损失值` `epoch_times = []  # 存储每个epoch 的训练时间`

```
epoch_loss = 0.0    # 每个 epoch 的总损失
epoch_images = 0    # 每个 epoch 处理的图片数
```

（3）改进进度展示

原始代码：仅使用 tqdm 包装了 epoch 循环

```
for epoch in tqdm(range(num_epochs)):
```

修改后：为每个 epoch 的批次循环也添加了 tqdm 进度条，并增加了描述

```
for batch_idx, (images, labels) in enumerate(tqdm(train_loader, desc=f"Epoch {epoch+1}/{num_epochs}")):
```

（4）优化输出格式

```
# 迭代训练
num_epochs = 20
print(f"开始训练，共 {num_epochs} 个epochs")
print("=" * 50)
```

```
# 输出最终结果
print("\n" + "=" * 50)
print("训练完成总结:")
```

（5）提供更全面的训练总结

```
# 输出最终结果
print("\n" + "=" * 50)
print("训练完成总结:")
print(f"总处理图片数量: {total_images}")
print(f"总训练时间: {total_time:.2f}秒")
print(f"平均每个epoch时间: {avg_epoch_time:.2f}秒")
print(f"平均每张图片处理时间: {avg_time_per_image:.5f}秒")
print(f"最终批次平均损失: {final_avg_loss:.4f}")
print(f"训练速度: {total_images/total_time:.1f}张/秒")
```

2. 结果展示：

```
==================================================
训练完成总结:
总处理图片数量: 1200000
总训练时间: 12.31秒
平均每个epoch时间: 16.06秒
平均每张图片处理时间: 0.00001秒
最终批次平均损失: 0.2796
训练速度: 97509.1张/秒
```

总处理图片数量：达到了 1200000 张，说明在训练过程中对大量数据进行了有效学习，能让模型充分接触各类样本，有助于提升泛化能力

总训练时间：仅为 12.31 秒，平均每个 epoch 时间为 16.06 秒，平均每张图片处理时间低至 0.00001 秒，训练速度高达 97509.1 张 / 秒。这样的时间指标表明训练过程非常高效，在短时间内就能完成大规模数据的训练

（四） mnist_gpu.py

由于在指标与输出方式上采用与 CPU 相同的修改思路，故在此不再进行赘述，仅分析 GPU 独有的改进点

1. 以下将结合代码进行分析:

设备信息：从 “隐式判断” 到 “透明输出”

原始代码**仅隐性判断 GPU 是否可用**，未输出任何设备细节，因此无法直观确认当前训练环境

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = torch.nn.Linear(input_size, output_size).to(device)
```

修改后的代码明确输出当前使用设备

```
# 检查GPU是否可用并输出详细信息
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"使用设备: {device}")
if torch.cuda.is_available():
    print(f"GPU名称: {torch.cuda.get_device_name(0)}")
    print(f"CUDA版本: {torch.version.cuda}")
    print(f"GPU内存: {torch.cuda.get_device_properties(0).total_memory / 1024**3:.2f}

# 定义数据预处理和加载器
```

2. 结果输出:

```
训练完成!
总处理图片数: 1200000
总耗时: 39.39秒
平均每张图片处理时间: 0.000033秒
整体平均损失: 0.319792
整体处理速度: 30464.1张/秒
(gpu_env) PS C:\Users\screen01\cnn_example\example2>
```

可见，实验结果与预期有所差距，总耗时甚至超过在 CPU 上进行运行，由于这是一次还挺独特的修改，故也给予保留修改后的代码，思考可能的原因如下:

（1）模型与任务特性限制了 GPU 优势

模型过于简单，计算量极小；代码中使用的是线性模型（torch.nn.Linear），仅包含一次矩阵乘法运算（输入 784 维→输出 10 维）

- GPU 的并行计算优势依赖于大规模并行任务，而简单线性模型的计算量不足以让 GPU 的数千个 CUDA 核心充分利用，导致资源闲置。
- 对比而言，CPU 处理小规模计算时，指令调度和内存访问的额外开销相对更小，因此与 GPU 的速度差距被缩小

（2）数据传输与预处理的额外开销

代码中每批次都需要将数据从 CPU 内存传输到 GPU 显存（`images = images.to(device)`、`labels = labels.to(device)`）:

对于简单模型，数据传输时间可能接近甚至超过 GPU 的计算时间，导致整体耗时中 “传输开销” 占比过大，掩盖了 GPU 的计算优势

（3）改进的计时方式可能放大 GPU 耗时

```
tart_event.record()  # 记录 GPU 计算开始
# 前向传播+反向传播+参数更新
end_event.record()
torch.cuda.synchronize()  # 等待 GPU 操作完成
elapsed_time = start_event.elapsed_time(end_event) / 1000.0
```

虽然能精确包含了 GPU 上的所有计算时间，但未排除 CPU 与 GPU 之间的同步开销

七、再次对于 CPU 与 GPU 的时间消耗进行直观的感知

任务详情：比较 CPU 与 GPU 在完成相同计算任务时的时间消耗，以及对于计算机运行带来的其他影响

以示例二为基准进行问题解答：

在 CPU 运行：

在 GPU 运行：

GPU 在完成相同 MNIST 数据集训练任务时，总耗时为 5.46 秒，而 CPU 总耗时为 12.02 秒，GPU 的训练速度明显快于 CPU

通过查阅资料得知，除了在时间上的差异外，CPU 与 GPU 在完成相同计算任务<这里以执行深度学习训练任务为例>时，还会对计算机运行带来以下几方面的其他影响：

1. 资源占用

CPU：在执行深度学习训练任务时，由于其核心数量相对较少，并且需要处理操作系统调度、其他后台程序运行等多种任务，因此在进行大规模计算时，**CPU 的利用率很容易达到较高水平，甚至可能出现满载的情况**。这会导致计算机在训练过程中响应变慢，比如打开其他应用程序时会感觉到卡顿，或者多任务处理能力下降。**此外，高负载的 CPU 还会消耗较多的系统内存资源来缓存数据和指令，进一步影响系统的整体性能**。

GPU：**GPU 主要专注于并行计算任务，在进行深度学习训练时，它会占用大量的显存资源来存储模型参数、中间计算结果以及输入数据等**。不过，只要显存容量足够，并且训练任务的规模没有超过 GPU 的处理能力，**GPU 对系统其他资源的占用相对较少，计算机仍然可以较为流畅地运行其他非图形密集型的应用程序**。但如果显存不足，可能会导致训练过程中频繁出现数据交换到系统内存的情况（即 **“显存溢出”**），从而显著降低训练速度，甚至导致程序崩溃。

2. 功耗与发热

CPU：**当 CPU 全力执行深度学习计算任务时，其功耗会显著增加**。高功耗会导致 CPU 发热严重，需要强大的散热系统来保证其稳定运行。如果散热系统性能不足，CPU 可能会因为过热而自动降频，进一步降低计算速度，影响训练效

率。
GPU: **GPU 在进行大规模并行计算时，功耗同样很高**，尤其是高端的专业计算 GPU，其功耗甚至可以超过 200 瓦。高功耗带来的发热问题也非常明显，因此 GPU 通常配备有专门的高效散热风扇或水冷装置。与 CPU 类似，如果 GPU 散热不良，也会出现降频现象，影响深度学习训练的速度。而且，GPU 的发热还可能对机箱内其他硬件的温度产生影响，进而影响整个计算机系统的稳定性

3. 硬件寿命
CPU: 长**时间在高负载下运行深度学习计算任务，会加速 CPU 内部晶体管等元件的老化，缩短其使用寿命**。频繁的满载运行还可能导致 CPU 的散热硅脂老化、散热风扇磨损等问题，进一步影响其散热效率和稳定性。
GPU: 同理，**GPU 在高负载、高功耗的状态下长时间运行，也会加速 GPU 核心、显存等部件的老化**。特别是对于一些使用较为频繁的深度学习工作站或服务器，GPU 的硬件寿命可能会因为长期高强度的计算任务而受到明显影响。此外，GPU 的散热风扇等部件在长期高转速运行下，也更容易出现故障，需要定期维护和更换。

4. 软件兼容性与开发难度
CPU: **使用 CPU 进行深度学习训练，在软件兼容性方面相对较好**，因为大多数深度学习框架（如 TensorFlow、PyTorch 等）都对 CPU 有良好的支持，并且不需要额外安装复杂的驱动程序或进行特殊配置。然而，**由于 CPU 在深度学习计算方面的性能相对较弱，开发者在优化训练速度时可能需要花费更多的精力**，比如使用多线程、向量化计算等技术，这增加了代码开发和调试的难度。
GPU: **虽然 GPU 在计算速度上有巨大优势，但使用 GPU 进行深度学习训练需要安装特定的 GPU 驱动程序和相关的计算库，这些软件的安装和配置过程相对复杂，并且需要与深度学习框架的版本相互匹配，否则容易出现兼容性问题**，导致训练无法正常进行。不过，一旦配置完成，**深度学习框架可以自动利用 GPU 的并行计算能力，开发者在代码层面不需要进行过多的并行化优化，只需要关注模型的构建和训练逻辑即可**，这在一定程度上降低了开发难度，提高了开发效率。

因此通过查阅多方资料可知，GPU 在深度学习计算任务中虽然具有明显的速度优势，但在资源占用、功耗发热、硬件寿命以及软件兼容性等方面也存在一些需要关注和解决的问题；而 CPU 虽然计算速度相对较慢，但在通用性和软件兼容性方面具有一定的优势。

结论分析与体会:
● 实验中遇到的问题与解决方案:
备注：由于在撰写上述实验步骤与内容时，已经在实验过程中分别撰写遇到的问题（问题与过程是紧密相关的）因此在此处更多的是汇总与总结过程:
一、安装配置阶段问题与解决方案
（一）CUDA 验证时 nvcc -V 命令不存在
① 问题现象:
在 CUDA 10.1 安装完成后，打开命令提示符输入 nvcc -V 验证安装，终端提示 **“'nvcc' 不是内部或外部命令，也不是可运行的程序或批处理文件”，**

无法确认 CUDA 是否安装成功

② 原因分析

CUDA 安装程序仅自动配置了 4 条系统变量（如 CUDA_PATH），未将 CUDA 的 bin 和 libnvvp 目录添加到系统环境变量 Path 中。命令提示符执行 nvcc 时，会从 Path 指定的目录中查找可执行文件，缺少路径配置导致命令无法识别。

③ 解决方案

手动添加 CUDA 相关路径到系统环境变量 Path，

具体步骤如下（回忆版）：

右键 “此电脑” → “属性” → “高级系统设置” → “环境变量” → “系统变量” →找到 “Path” 并编辑;

点击 “新建”，分别添加以下两条路径:

C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.1\bin（nvcc.exe 所在目录）;

C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.1\libnvvp（CUDA 可视化工具相关目录）;

点击 “确定” 保存，关闭所有已打开的命令提示符，重新打开后输入 nvcc -V，终端正常显示 CUDA 版本信息

```
C:\Users\screen01>nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Fri_Feb__8_19:08:26_Pacific_Standard_Time_2019
Cuda compilation tools, release 10.1, V10.1.105
```

二、示例运行阶段问题与解决方案

（一）依赖库缺失类问题

1. ModuleNotFoundError: No module named 'tqdm'

① 问题现象

运行 mnist_cpu.py 和 mnist_gpu.py 脚本时，终端抛出错误，提示缺少 tqdm 模块，程序无法加载进度条功能，训练流程中断。

② 原因分析

tqdm 是 Python 第三方库，用于实现训练过程的进度条可视化，代码中通过 from tqdm import tqdm 导入该库。当前使用的虚拟环境未预装 tqdm，导致 Python 解释器无法找到该模块

2. ModuleNotFoundError: No module named 'matplotlib'

① 问题现象

运行 example_cifar_train.py 训练脚本时，终端报错，提示缺少 matplotlib 模块，代码中 import matplotlib.pyplot as plt 语句无法执行。

② 原因分析

matplotlib 是 Python 常用的数据可视化库，脚本中可能用于绘制训练损失、准确率曲线等图表。当前虚拟环境未安装该库，导致导入失败

由于上述问题是同一类问题：解决方案均为使用 pip 进行安装

pip install tqdm

pip install matplotlib

```
ModuleNotFoundError: No module named 'torch'
(pytorch_env) PS C:\Users\screen01\cnn_example\example1> conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia
3 channel Terms of Service accepted
Channels:
 - pytorch
 - nvidia
 - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
 - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
 - defaults
Platform: win-64
Collecting package metadata (repodata.json): done
Solving environment: done
```

3. ModuleNotFoundError: No module named 'torch'

① 问题现象

在 VS Code 中运行 example_cifar_train.py 时，终端提示缺少 torch 模块，尽管此前已在 Anaconda 的 myenv 环境中安装 PyTorch。

② 原因分析

VS Code 默认使用系统自带的 Python 环境，未自动关联 Anaconda 中安装 PyTorch 的 myenv 虚拟环境。Python 解释器在系统环境中找不到 torch 库，导致导入失败（思考：万老师上课刚好补充到!）

③ 解决方案

通过 conda 包管理工具安装 PyTorch 及其相关依赖库

（二）代码执行逻辑类问题

1. AttributeError: '_SingleProcessDataLoaderIter' object has no attribute 'next'

① 问题现象

运行 example_cifar_train.py 时，终端报错，错误定位到 val_image, val_label = val_data_iter.next()语句，提示_SingleProcessDataLoaderIter 对象没有 next 属性。

② 原因分析

Python 3 中，迭代器的“获取下一个元素”方法已统一为__next__()，而代码中使用的 next()是 Python 2 的语法。DataLoader 的迭代器_SingleProcessDataLoaderIter 遵循 Python 3 规范，仅支持__next__()方法，因此 next()调用失败。

③ 解决方案

使用 Python 3 内置的 next()函数获取下一个数据批次：

将代码中 val_image, val_label = val_data_iter.next()修改为 val_image, val_label = next(val_data_iter)；next()函数会自动调用迭代器的__next__()方法，重新运行脚本

```
                                        shuffle=False, num_workers=0)
# 获取测试集中的图像和标签，用于accuracy计算
val_data_iter = iter(val_loader)
val_image, val_label = next(val_data_iter)
```

2. Windows 系统 DataLoader 多进程报错

① 问题现象

运行 mnist_cpu.py 时，终端抛出 RuntimeError: DataLoader worker (pid(s) xxxx) exited unexpectedly

② 原因分析

num_workers 参数用于指定数据加载的子进程数量，以加速数据读取。但

Windows 系统的多进程机制（基于 spawn）与类 Unix 系统存在差异，DataLoader 使用非 0 的 num_workers 时，易出现进程创建失败、数据传递异常等问题

③ 解决方案

将 DataLoader 的 num_workers 参数设为 0，采用单进程加载数据:

```
train_loader = DataLoader(
    train_dataset,
    batch_size=64,
    shuffle=True,
    num_workers=0  # 关键修改：Windows下多进程不可用，设为0
)
```

（三）工具操作类问题

1. VS Code 未自动保存导致代码执行异常

① 问题现象

在 VS Code 中修改脚本后直接运行，终端报错与代码逻辑不符，但检查代码发现无明显问题。<当时运行了很多遍，一直显示报错，但是结合代码无明显错误，后来发现是没保存，也是一次挺有意思的经历>

② 原因分析

VS Code 默认未开启“自动保存”功能，修改后的代码未写入文件，终端执行的仍是修改前的旧代码，导致错误。

③ 解决方案

开启 VS Code 自动保存或手动保存代码

- 实验结果达到设计目标的程度

本次实验围绕“深度学习环境搭建、模型训练与优化、CPU 与 GPU 性能对比”的核心目标展开，整体达成情况良好，具体如下:

（一）环境搭建目标

成功完成了从硬件驱动到软件框架的全流程配置。**通过 NVIDIA 官网下载并安装适配显卡的驱动程序，依据显卡算力选择并安装 CUDA 10.1，手动配置环境变量解决了 nvcc -V 验证失败的问题；利用 Anaconda 创建虚拟环境，成功部署与 CUDA 版本适配的 PyTorch 1.7.1**，所有工具均可正常调用，为后续实验筑牢了基础。

（二）模型训练与优化目标

基础示例运行: **顺利运行 CIFAR - 10 数据集的 LeNet 训练与推理、MNIST 数据集的全连接模型训练，完整实现“数据下载 - 训练 - 验证 - 模型保存 - 离线推理”流程**。

最终运行结果: **LeNet 模型训练后验证准确率达 69.3%，能准确分类“船”等测试图片；MNIST 全连接模型在 CPU 与 GPU 环境下均完成 20 个 epoch 训练，模型损失稳定下降至 0.28 左右**，具备基本分类能力

代码优化: 对 4 个核心脚本进行针对性修改，**为 example_cifar_train.py 新增 GPU 加速、数据增强（随机裁剪、水平翻转）与动态学习率调度，验证准确**

率提升至 75.17%；为 **example_cifar_infer.py 添加批量推理、置信度计算与概率可视化功能**；为 **mnist_cpu.py 与 mnist_gpu.py 补充训练监控与设备信息输出**。虽在“多尺度推理策略的稳定性”上存在小幅不足，但整体满足优化目标。

（三）CPU 与 GPU 性能对比目标

通过控制变量法（保持相同模型结构、批次大小、训练轮次）对比两者性能，在 **MNIST 数据集训练任务中，GPU（RTX 2060 SUPER）总耗时 5.46 秒，CPU 总耗时 12.02 秒，GPU 速度约为 CPU 的 2.2 倍**；同时查询资料，记录了两者在资源占用、功耗发热的差异。

- 改进点

（一）依赖库下载优化

利用 requirements.txt 记录 PyTorch、matplotlib、tqdm 等依赖库的版本信息，通过 pip install -r requirements.txt 快速复现环境，避免版本冲突问题

（二）模型性能与功能

模型结构优化：当前 LeNet 模型针对 CIFAR - 10 的分类准确率仍有提升空间，可引入**改进型网络（如 VGG - 11 简化版、ResNet - 18）<查询资料得到>**，增加卷积层与残差连接，结合更**丰富的数据增强**，进一步降低过拟合，可能验证准确率提升至 85% 以上

推理功能扩展：现有推理脚本仅支持图像分类，可新增 **“错误案例分析”** 功能，统计模型分类错误的图片类型（如在跑出的结果： CIFAR - 10 中“猫”与“狗”的混淆案例），输出错误图片的概率分布热力图

（三）性能对比实验

变量控制与数据维度扩展：当前对比仅覆盖“全连接模型 ＋MNIST”场景，可增加 **“复杂模型（如 ResNet）＋ 大规模数据集（如 ImageNet 子集）”的对比**，观察 GPU 在计算量增大时的加速比变化；同时记录“数据预处理耗时”“模型加载耗时”等细分指标，更精准分析 **GPU “数据传输开销”对整体性能的影响**

硬件监控工具集成：引入 nvidia - smi 实时监控脚本（每 10 秒记录一次 GPU 温度、显存占用、功耗），结合 **taskmgr 记录 CPU 利用率**，生成“训练过程硬件状态变化曲线”，更直观展示两者在高负载下的稳定性差异

（将在资料中的信息更加直观的感知）

- 实验收获与启发

（一）技术能力：从理论到实践

这是我首次**完整接触深度学习全流程**，此前听到的“PyTorch 张量运算”等概念，终于在实际操作中有了具象认知。比如在解决 DataLoader 多进程报错时，也才清楚 Windows 与 Linux 多进程机制的差异。这些经历让我意识到，“能跑通代码” 只是第一步，**“理解报错背后的原理”** 才是真正掌握技术的关键

在代码优化环节，为 example_cifar_infer.py 添加置信度等级评估时以及为 mnist_gpu.py 补充设备信息输出时，这个过程不仅提升了我的代码能力，更让我学会 **“带着问题查资料”**，这种 **“发现问题 - 搜索方案 - 验证解决”** 的闭

环，比单纯学会某个知识点更有价值

（二）心态成长：享受解决问题的过程

实验中遇到过很多“看似愚蠢”的问题：比如忘记保存 VS Code 代码导致反复报错，一开始会懊恼，但后来发现，正是这些小失误让我养成“修改后及时保存”的习惯；再比如 GPU 训练耗时偶尔超过 CPU，一开始以为是配置错误，后来通过分析“模型计算量过小 - 数据传输开销占比高”的原因，反而加深了对 GPU 加速原理的理解

这次实验让我明白，接触新领域时“犯错”是正常的，重要的是不逃避问题。未来再面对复杂的技术问题，我会更有底气，因为我知道，每一个报错都是学会新知识的机会。很庆幸能有这样的实践机会深入接触深度学习技术，也让我对未来在人工智能领域的学习和探索充满期待

附录：

附录 1	
代码命名说明	
Example1	
1. example_cifar_train_new.py	对于 example_cifar_train 的修改
2. example_cifar_infer_new.py	对于 example_cifar_infer 的第一次修改（即报告中的第二版）
3. example_cifar_infer_improve.py	对于 example_cifar_infer 的第二次修改（即报告中的第三版）
Example2	
1. mnist_cpu_new.py	对于 mnist_cpu_new.py 的修改
2. mnist_gpu_new.py	对于 mnist_gpu_new.py 的修改

前提说明：以下分析是在修改代码前对于代码进行的理解与分析，将其作为实验过程记录在此处（还是很有成就感滴）

附录 2

example_cifar_train.py 核心代码的分析

```
#数据预处理（transforms）
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

进一步分析：

`transforms.ToTensor()`：将 PIL 图像转为 torch.Tensor，并把像素值从 [0,255] 缩放到 [0,1]，同时调整维度为 (C, H, W)。

`transforms.Normalize(mean, std)`：对张量做标准化，公式为 output = (input - mean) / std。这里均值和标准差都设为 (0.5, 0.5, 0.5)，让像素值分布到 [-1,1] 区间，加速模型收敛

```
#数据集与数据加载器
# 训练集
train_set = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=36, shuffle=True, num_workers=0)
# 验证集
val_set = torchvision.datasets.CIFAR10(root='./data', train=False, download=False, transform=transform)
val_loader = torch.utils.data.DataLoader(val_set, batch_size=5000, shuffle=False, num_workers=0)
```

进一步分析：

`torchvision.datasets.CIFAR10`：**自动下载（download=True）并加载 CIFAR-10 数据集**，train=True 对应 50000 张训练图，train=False 对应 10000 张验证图，transform 指定预处理逻辑。

`torch.utils.data.DataLoader`：将数据集封装为可迭代的批量数据加载器。**batch_size 控制每批样本数（训练用 36，验证用 5000）；shuffle=True 打乱训练集顺序以增强泛化性**，验证集则无需打乱；num_workers 控制多线程加载，在实验中将其设为 0 以避免兼容问题

```
#以下是模型与损失函数模块
```

模型定义：

```
net = LeNet()
```

损失函数与优化器：

```
loss_function = nn.CrossEntropyLoss()  # 交叉熵损失
optimizer = optim.Adam(net.parameters(), lr=0.001)  # Adam 优化器
```

```
#训练流程模块
```

'''通过多轮（epoch）迭代 + 每轮批量（batch）训练，让模型逐步学习数据规律，核心是 **“前向传播 → 损失计算 → 反向传播 → 参数更新”** 的循环'''

```
for epoch in range(5):  # 共训练 5 轮（对整个训练集遍历 5 次）
    for step, data in enumerate(train_loader, start=0):  # 遍历训练集的每个批次
        inputs, labels = data  # 取当前批次的“图像 + 标签”
        optimizer.zero_grad()  # 清空历史梯度
        outputs = net(inputs)  # 前向传播：模型对图像做预测
        loss = loss_function(outputs, labels)  # 计算“预测值与真实标签”的损失
        loss.backward()  # 反向传播：计算参数的梯度
        optimizer.step()  # 参数更新：用 Adam 优化器根据梯度调整模型参数

#验证与模型保存模块
验证逻辑：
with torch.no_grad():  # 验证时无需计算梯度
    outputs = net(val_image)  # 模型对验证集图像做预测
    predict_y = torch.max(outputs, dim=1)[1]  # 取概率最大的类别索引作为预测结果
    accuracy = torch.eq(predict_y, val_label).sum().item() / val_label.size(0)  # 计算准确率

模型保存：
save_path = './Lenet.pth'
torch.save(net.state_dict(), save_path)

总结：数据预处理 → 模型构建 → 训练优化 → 效果验证 → 模型保存的深度学习流程
```

附录 3

example_cifar_infer.py 的核心代码分析

```
#数据预处理模块
推理时的图像预处理需与训练时保持一致，否则会因数据分布差异导致预测错误
transform = transforms.Compose([
    transforms.Resize((32, 32)),  # 调整图像尺寸为 32×32
    transforms.ToTensor(),  # 转为张量并归一化到 [0,1]
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # 标准化到 [-1,1]])

#模型加载模块
net = LeNet()
net.load_state_dict(torch.load('Lenet.pth'))
进一步分析：
实例化模型结构：LeNet() 创建一个新的 LeNet 网络，此时参数是随机初始化的，无法直接用于推理。
加载预训练参数：torch.load('Lenet.pth') 读取保存的模型参数

#推理流程模块
加载并预处理图像
```

```
im = Image.open('img/ship1.jpg')  # 用 PIL 读取图像
im = transform(im)  # 得到 shape 为 (C, H, W) 的张量
调整输入维度
im = torch.unsqueeze(im, dim=0)  # 增加 batch 维度，shape 变为 (1, C, H, W)
模型预测
with torch.no_grad():  # 关闭梯度计算，节省内存并加速推理
    outputs = net(im)  # 前向传播，得到 shape 为 (1, 10) 的输出
    predict = torch.max(outputs, dim=1)[1].data.numpy()  # 取分数最大的类别索引

#结果输出模块
最后将预测的类别索引映射为具体类别名称：
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
print("class:", classes[int(predict)])
```

附录 4

mnist_cpu.py 的核心代码

```
#数据预处理模块
数据预处理：
transform = transforms.Compose([
    transforms.ToTensor(),  # 转为张量并归一化到 [0,1]
    transforms.Normalize((0.5,), (0.5,))  # 标准化到 [-1,1]
])

加载训练集：
train_dataset = datasets.MNIST(
    'data',                # 数据存放路径
    train=True,            # 加载训练集
    download=True,         # 自动下载数据集（首次运行时）
    transform=transform    # 应用预处理
)

构建数据加载器：
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=64,         # 每批处理 64 张图片
    shuffle=True           # 打乱训练数据顺序
)

#模型与优化器模块
本示例中使用单层线性模型完成分类
# 输入维度：28×28 像素展开为一维向量
input_size = 28*28# 输出维度：10 个类别（数字 0-9）
output_size = 10
```

```
# 定义线性模型：y = wx + b（w 是 784×10 权重矩阵，b 是 10 维权值）
model = torch.nn.Linear(input_size, output_size)
# 损失函数：交叉熵损失
criterion = torch.nn.CrossEntropyLoss()
# 优化器：随机梯度下降（SGD），学习率 lr=0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

#训练与计时模块
初始化统计变量：
total_images = 0  # 总处理图片数
total_time = 0.0  # 总处理时间
# 训练 20 轮（epoch）
for epoch in tqdm(range(num_epochs)):
    # 遍历每个批次的训练数据
    for images, labels in train_loader:
        start_time = time.time()  # 记录批次处理开始时间

        # 1. 数据格式调整：将 (64,1,28,28) 转为 (64,784)（展开为一维向量）
        images = images.view(-1, input_size)

        # 2. 前向传播：计算预测输出和损失
        output = model(images)
        loss = criterion(output, labels)

        # 3. 反向传播与参数更新
        optimizer.zero_grad()  # 清空历史梯度
        loss.backward()        # 计算梯度
        optimizer.step()       # 更新参数

        # 4. 计时统计：累加当前批次的处理时间和图片数
        end_time = time.time()
        processing_time = end_time - start_time
        total_images += len(images)
        total_time += processing_time

#效率统计模块
计算单张图片的平均处理时间：
avg_time_per_image = total_time / total_images

# 输出统计结果
print(f"Total images processed: {total_images}")
# 总图片数（60000×20=1,200,000）
print(f"Total time taken: {total_time:.2f} seconds") # 总耗时
print(f"Average time per image: {avg_time_per_image:.5f} seconds")  # 单张平
```

均耗时

附录 4

mnist_gpu_new 的核心代码分析

由于此代码与上述在 CPU 上运行的代码基本逻辑相同，这里只分析不同的部分，并且对于两者代码的不同进行对比

1. 设备与模型部署

```
# 检查 GPU 是否可用，优先使用 GPU（cuda:0 表示第 1 块 GPU），无则用 CPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

2. 模型部署到 GPU

```
model = torch.nn.Linear(input_size, output_size).to(device)
```

3. 数据从 CPU 传输到 GPU

```
for images, labels in train_loader:
    # 将批次数据从 CPU 内存转移到 GPU 显存
    images = images.to(device)
    labels = labels.to(device)
    # 后续前向传播、损失计算等操作，均在 GPU 上基于显存中的数据进行
```

4. 计时逻辑

CPU 版本用 time.time() 测量“CPU 发起操作到结束”的总时间（**包含 GPU 异步计算的干扰**）；GPU 版本用 CUDA 事件计时，专门测量 GPU 自身的计算耗时，避免异步执行导致的误差

```
# 创建 CUDA 事件对象，用于标记 GPU 计算的起止点（enable_timing=True 开启计时功能）
start_event = torch.cuda.Event(enable_timing=True)
end_event = torch.cuda.Event(enable_timing=True)
# 记录 GPU 计算开始（CPU 版本用 start_time = time.time()）
start_event.record()
# 核心计算：前向传播、反向传播、参数更新（均在 GPU 上执行）
images = images.view(-1, input_size)
output = model(images)
loss = criterion(output, labels)
optimizer.zero_grad()
loss.backward()
optimizer.step()
# 记录 GPU 计算结束，并等待 GPU 操作完成（CPU 版本无同步步骤）
end_event.record()
torch.cuda.synchronize()  # 关键：等待 GPU 完成所有操作，避免计时提前结束
# 计算 GPU 耗时：elapsed_time 返回毫秒，转换为秒（CPU 版本用 end_time - start_time）
elapsed_time = start_event.elapsed_time(end_event) / 1000.0
```