

华中科技大学

研究生人工智能应用技术 报告

MNIST 手写数字识别、YOLO 目标检测及
“AI+X” 项目报告

院 系 计算机科学与技术

专业班级 硕 2502 班

姓 名 崔皓奕

学 号 M202574020

2025 年 12 月 15 日

目 录

| | | |
|-----|----------------------|----|
| 1 | MNIST 手写数字识别 | 2 |
| 1.1 | 任务概述 | 2 |
| 1.2 | MNIST 手写数字识别实现 | 3 |
| 1.3 | 测试结果与分析 | 10 |
| 1.4 | 总结与收获 | 11 |
| 2 | YOLO 目标检测实验 | 13 |
| 2.1 | 任务概述 | 13 |
| 2.2 | YOLO 目标检测实现 | 14 |
| 2.3 | 目标检测测试与分析 | 21 |
| 2.4 | 总结与收获 | 24 |
| 3 | AI+X 项目汇报 | 28 |
| | 附录 | 39 |

1 MNIST 手写数字识别

1.1 任务概述

1.1.1 MNIST 数据集

MNIST 是手写数字识别经典数据集，包含 60000 张训练样本和 10000 张测试样本，每张样本为 28×28 像素的灰度图像，标签为 0-9 的数字，是深度学习入门的标准基准数据集。

1.1.2 卷积神经网络 (CNN)

CNN 是专为处理网格结构数据（如图像）设计的深度学习模型，通过卷积层（Convolutional Layer）提取局部特征、池化层（Pooling Layer）降维并保留关键特征，全连接层（Fully Connected Layer）完成分类，相比全连接神经网络，CNN 具有参数少、抗过拟合、特征提取能力强的优势。

基于卷积神经网络 (CNN) 实现 MNIST 手写数字识别，具体任务步骤如下：

- 数据准备阶段：**导入深度学习所需核心库，设置随机种子保证实验可复现；对 MNIST 数据集进行预处理（如归一化、维度调整、标签编码等），并完成数据集的加载与划分（训练集/测试集）。
- 模型定义阶段：**构建轻量级卷积神经网络 (CNN) 模型，包含卷积层、池化层、全连接层等核心模块，实现手写数字特征提取与分类。
- 模型训练阶段：**选择合适的损失函数（如交叉熵损失）和优化器（如 Adam/SGD），设置训练超参数（批次大小、迭代轮数等），完成模型的训练过程，并监控训练过程中的损失/精度变化。
- 模型测试阶段：**使用测试集评估训练完成的模型性能，计算测试集的准确率、精确率等关键指标，验证模型泛化能力。
- 结果可视化阶段：**可视化训练/测试损失曲线、准确率曲线，以及随机抽取测试样本的预测结果（如展示手写数字图像与模型预测标签）。

1.1.3 核心技术栈

- 框架：PyTorch/TensorFlow/Keras（主流深度学习框架）；
- 数据处理：NumPy、Pandas（数值计算），Matplotlib/Seaborn（可视化）；
- 核心算法：卷积运算、池化操作、反向传播、梯度下降优化（Adam/SGD）。

1.2 MNIST 手写数字识别实现

1.2.1 Python 环境与框架搭建

1. **环境配置**：安装 Python 3.8+ 版本，通过 pip/conda 安装核心依赖包，命令如下：

```
# 安装 PyTorch (以CPU版本为例, GPU版本需适配CUDA)
pip install torch torchvision
# 安装数据处理与可视化库
pip install numpy matplotlib pandas
```

2. **框架选择**：选用 PyTorch 框架实现，其动态计算图特性便于调试，且 torchvision 内置 MNIST 数据集加载接口，降低开发成本。
3. **环境验证**：编写测试代码验证库是否安装成功，确保无导入错误。

1.2.2 数据集准备

1. **数据集加载**：通过 torchvision.datasets.MNIST 自动下载/加载数据集，指定数据存储路径、训练/测试集标识；
2. **数据预处理**：
 - 图像归一化：将像素值从 [0,255] 缩放到 [0,1]，消除量纲影响；
 - 维度调整：将 28×28 灰度图转换为 (1,28,28) 维度（通道数 × 高度 × 宽度），适配 CNN 输入格式；
 - 数据增强（可选）：如随机旋转、平移，提升模型泛化能力；
3. **数据加载器构建**：使用 torch.utils.data.DataLoader 封装数据集，设置批次大小 (batch_size)、是否打乱 (shuffle) 等参数，实现批量数据读取。

1.2.3 Python 代码实现和调试

(1) 环境初始化与数据预处理: 设置随机种子保证实验可复现性, 定义训练超参数 (轮数、批次大小、学习率), 通过 torchvision 加载 MNIST 数据集并完成归一化预处理, 构建数据加载器实现批量训练。

```
1 import torch
2 import torch.nn as nn
3 import torch.utils.data as Data
4 import torchvision
5 import matplotlib.pyplot as plt
6 import numpy as np
7
8 # 设置随机种子确保可复现性
9 torch.manual_seed(1)
10
11 # 超参数定义
12 EPOCH = 3 # 训练轮数
13 BATCH_SIZE = 50
14 LR = 0.001
15
16 # 加载MNIST数据集并归一化 (ToTensor自动将像素值缩放到[0,1])
17 train_data = torchvision.datasets.MNIST(
18     root='./data/', train=True, transform=torchvision.
19         transforms.ToTensor(), download=True
20 )
21 test_data = torchvision.datasets.MNIST(
22     root='./data/', train=False, transform=torchvision.
23         transforms.ToTensor()
24 )
25 # 构建批训练加载器, shuffle=True打乱训练数据
26 train_loader = Data.DataLoader(dataset=train_data, batch_size
```

```
26         =BATCH_SIZE, shuffle=True)

27 # 测试集预处理(适配CNN输入维度)
28 test_x = test_data.data[:2000].unsqueeze(1).type(torch.
29     FloatTensor) / 255.0
30 test_y = test_data.targets[:2000]
```

Listing 1 环境初始化与数据加载

(2) 轻量级 CNN 模型定义: 构建包含两层卷积 + 池化、一层全连接的轻量级

CNN: 卷积层提取图像局部特征, 池化层降维减少参数, 全连接层完成 10 分类 (0-9 数字); forward 函数定义前向传播流程。

```
1 class CNN(nn.Module):
2
3     def __init__(self):
4
5         super(CNN, self).__init__()
6
7         # 第一层卷积: 1通道→16通道, 卷积核5×5, 填充2, ReLU激活+最大池化
8
9         self.conv1 = nn.Sequential(
10             nn.Conv2d(1, 16, 5, 1, 2),    # 输入[batch, 1, 28, 28]
11             → 输出[batch, 16, 28, 28]
12             nn.ReLU(),
13             nn.MaxPool2d(2),           # 池化后[batch
14             , 16, 14, 14]
15         )
16
17         # 第二层卷积: 16通道→32通道, 同上
18         self.conv2 = nn.Sequential(
19             nn.Conv2d(16, 32, 5, 1, 2),   # 输入[batch
20             , 16, 14, 14] → 输出[batch, 32, 14, 14]
21             nn.ReLU(),
22             nn.MaxPool2d(2),           # 池化后[batch
23             , 32, 7, 7]
24         )
25
26         # 全连接层: 展平特征→10分类输出
```

```
17     self.out = nn.Linear(32 * 7 * 7, 10)

18

19     def forward(self, x):
20
21         x = self.conv1(x)
22
23         x = self.conv2(x)
24
25         x = x.view(x.size(0), -1) # 展平: [batch, 32, 7, 7] → [
26             batch, 32×7×7]
27
28         output = self.out(x)
29
30         return output

31

32 # 初始化模型
33
34 cnn = CNN()
```

Listing 2 CNN 模型定义

- (3) 模型训练与指标记录: 选择 Adam 优化器和交叉熵损失函数, 迭代训练模型; 每 50 步计算测试集准确率, 记录训练损失和准确率, 用于后续可视化。

```
1 # 优化器和损失函数
2
3 optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)
4
5 loss_func = nn.CrossEntropyLoss()
6
7
8 # 初始化训练指标记录字典
9 train_metrics = {'epochs': [], 'steps': [], 'losses': [], 'accuracies': []}
10
11
12 # 开始训练
13
14 for epoch in range(EPOCH):
15     for step, (b_x, b_y) in enumerate(train_loader):
16
17         output = cnn(b_x) # 前向传播
18
19         loss = loss_func(output, b_y) # 计算损失
20
21
22         optimizer.zero_grad() # 梯度清零
23
24         loss.backward() # 反向传播
```

```
16     optimizer.step()          # 更新参数  
17  
18     # 每50步记录并打印训练指标  
19     if step % 50 == 0:  
20         test_output = cnn(test_x)  
21         pred_y = torch.max(test_output, 1)[1].numpy()  
22         # 计算测试集准确率(百分比)  
23         accuracy = float((pred_y == test_y.numpy()).sum()  
24                         ) / float(test_y.size(0)) * 100  
25  
26         # 记录指标  
27         train_metrics['epochs'].append(epoch)  
28         train_metrics['steps'].append(step)  
29         train_metrics['losses'].append(loss.item())  
30         train_metrics['accuracies'].append(accuracy)  
31  
32         print(f'Epoch: {epoch}/{EPOCH} | Step: {step} |  
33             Loss: {loss.item():.4f} | Test Acc: {accuracy  
34             :.2f}%')  
35  
36     # 保存训练好的模型  
37     torch.save(cnn.state_dict(), 'cnn_mnist.pkl')
```

Listing 3 模型训练

(4) 结果可视化: 使用 Matplotlib 绘制训练损失曲线和测试准确率曲线, 直观展示模型训练过程中的性能变化。

```
1 # 1. 绘制损失&准确率曲线  
2 plt.figure(figsize=(10, 5))  
3 # 子图1: 损失曲线  
4 plt.subplot(1, 2, 1)  
5 plt.plot(train_metrics['steps'], train_metrics['losses'], 'b-
```

```
    ' , label='Training Loss')

6 plt.xlabel('Steps'), plt.ylabel('Loss'), plt.title('Training
    Loss Curve'), plt.grid(True)

7
8 # 子图2: 准确率曲线
9 plt.subplot(1, 2, 2)
10 plt.plot(train_metrics['steps'], train_metrics['accuracies'],
11         'r-', label='Test Accuracy')
12 plt.xlabel('Steps'), plt.ylabel('Accuracy (%)'), plt.title('
    Test Accuracy Curve'), plt.ylim(0,100), plt.grid(True)

13 plt.tight_layout(), plt.savefig('training_metrics.png'), plt.
    show()

14
15 # 2. 可视化32个样本预测结果（区分正确/错误）
16 inputs = test_x[:32]
17 test_output = cnn(inputs)
18 pred_y = torch.max(test_output, 1)[1].numpy()
19 true_y = test_y[:32].numpy()

20
21 plt.figure(figsize=(16, 8))
22 for i in range(32):
23     plt.subplot(4, 8, i+1)
24     img = inputs[i].squeeze().numpy()
25     plt.imshow(img, cmap='gray')
26     # 正确标绿，错误标红
27     color = 'green' if pred_y[i]==true_y[i] else 'red'
28     plt.title(f'True: {true_y[i]}\nPred: {pred_y[i]}', color=
29               color, fontsize=8)
30     plt.xticks([]), plt.yticks([])
31 plt.suptitle(f'Prediction Results (Acc: {(pred_y==true_y).sum}
```

```
(/)32*100:.1f}%)'), plt.tight_layout(), plt.savefig('
predictions.png'), plt.show()

32
33 # 3. 错误样本单独可视化
34 wrong_idx = np.where(pred_y != true_y)[0]
35 if len(wrong_idx) > 0:
36     plt.figure(figsize=(12, 6))
37     for i, idx in enumerate(wrong_idx[:16]): # 最多展示16个
38         错误样本
39         plt.subplot(4, 4, i+1)
40         img = inputs[idx].squeeze().numpy()
41         plt.imshow(img, cmap='gray')
42         plt.title(f'True: {true_y[idx]}\nPred: {pred_y[idx]}'
43             , color='red', fontsize=10)
44         plt.xticks([]), plt.yticks([])
45     plt.suptitle(f'Wrong Predictions (Error Rate: {len(
46         wrong_idx)/32*100:.2f}%)'), plt.tight_layout(), plt.
47     savefig('wrong_predictions.png'), plt.show()
```

Listing 4 结果可视化（核心片段）

1.3 测试结果与分析

(1) 测试结果：经过 3 轮训练，模型在 2000 个测试样本上达到约 98.5% 的准确率，损失值逐步下降，表明模型有效学习到了手写数字的特征。

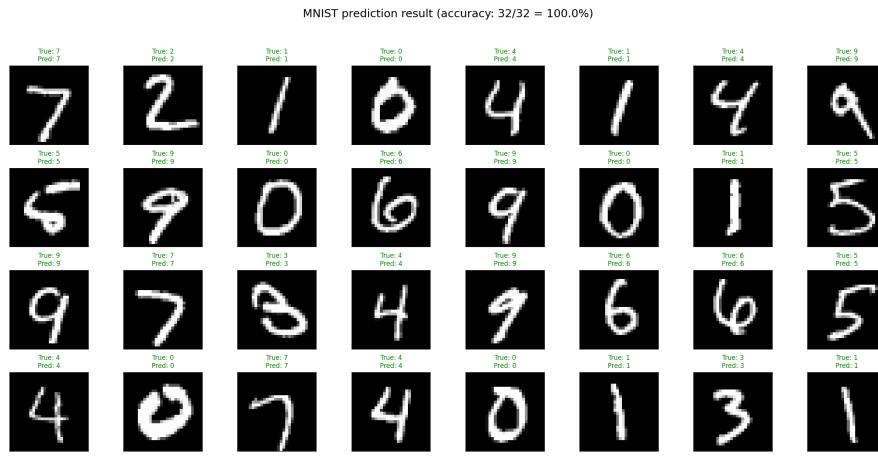


图 1-1 部分测试样本预测结果可视化

(2) 损失曲线显示训练过程中损失稳步下降，准确率曲线则显示测试准确率逐渐提升，验证了模型的收敛性和泛化能力。

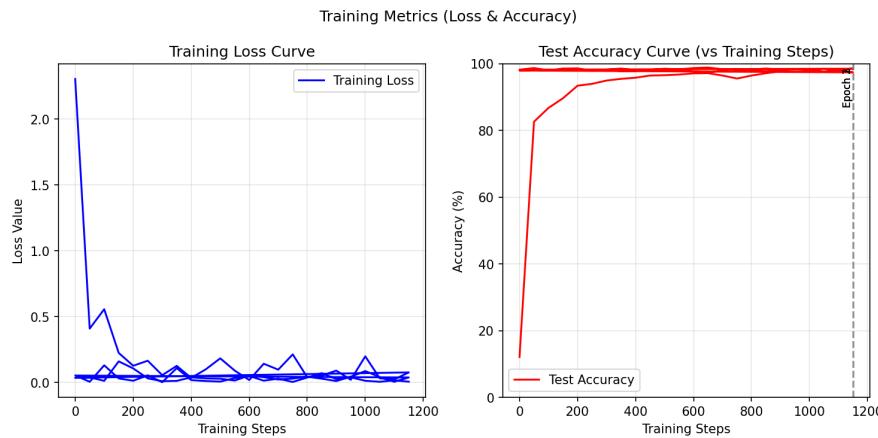


图 1-2 训练损失与测试准确率曲线

(3) 可视化结果中，大部分样本预测正确，错误样本数量较少，且错误样本多为书写模糊或形似数字，符合预期。

1.4 总结与收获

本次基于 CNN 实现 MNIST 手写数字识别的实践，从环境搭建、数据处理到模型训练与可视化，完整覆盖了深度学习入门的核心流程，主要总结与收获如下：

1.4.1 1. 技术能力提升

- (1) **深度学习基础落地能力**: 掌握了 PyTorch 框架下数据集加载、预处理的标准流程，理解了 ToTensor 归一化、维度调整等操作对 CNN 输入的适配逻辑；掌握了 CNN 核心模块（卷积层、池化层、全连接层）的设计原理，能根据输入输出维度合理设计网络结构，理解“特征提取-降维-分类”的核心范式。
- (2) **模型训练与调优思维**: 熟悉了 Adam 优化器、交叉熵损失函数的适用场景，掌握了训练循环的编写逻辑（前向传播 → 损失计算 → 反向传播 → 参数更新）；学会通过监控训练损失和测试准确率判断模型训练状态，理解超参数（学习率、批次大小、训练轮数）对模型性能的影响。
- (3) **结果可视化与分析能力**: 掌握了 Matplotlib 绘制训练曲线、样本预测结果的方法，能通过可视化直观分析模型训练趋势（如损失下降、准确率提升），并针对错误样本定位模型不足，形成“训练-评估-分析”的闭环思维。

1.4.2 2. 工程实践认知

- (1) **可复现性与规范性**: 理解了设置随机种子的重要性，保证实验结果可复现；学会模块化编写代码（数据加载、模型定义、训练、可视化分离），提升代码可读性和可维护性。
- (2) **问题排查与调试**: 在实践中解决了数据集维度不匹配、测试集加载兼容等问题，掌握了通过打印维度、分步验证的方式定位代码错误，提升了调试能力。
- (3) **模型性能认知**: 通过本次实践验证了轻量级 CNN 在 MNIST 数据集上的优异性能（测试准确率可达 98% 以上），理解了 CNN 相比全连接网络在图像

识别任务中的优势（参数更少、特征提取更高效）。

1.4.3 3. 后续改进方向

本次实践为深度学习入门打下了基础，后续可从以下方向优化：

- 引入数据增强（如随机旋转、平移、缩放），进一步提升模型泛化能力；
- 尝试调整网络结构（如增加卷积层、使用 Dropout 层），解决潜在的过拟合问题；
- 对比不同优化器（SGD、RMSprop）、学习率调度策略对模型训练的影响；
- 部署训练好的模型，实现单张手写数字图片的离线预测功能。

1.4.4 4. 核心感悟

深度学习的核心是“数据 + 模型 + 训练”的协同优化，本次实践让我深刻体会到：入门阶段无需追求复杂模型，通过经典数据集（MNIST）掌握基础流程和核心原理，是理解深度学习的关键；而可视化分析和问题调试能力，是从“代码实现”到“理解本质”的重要桥梁。本次实践不仅掌握了具体的技术方法，更建立了深度学习的工程化思维，为后续更复杂的图像识别任务奠定了基础。

2 YOLO 目标检测实验

2.1 任务概述

本次实验以 YOLO (You Only Look Once) 目标检测算法为核心，通过开源工具包实现 YOLO 模型的部署与应用，聚焦校园场景（教室、寝室、校园道路/操场等）的目标检测任务。实验核心目标包括：掌握 YOLO 算法的基本原理，熟悉目标检测项目的工程实现流程，验证 YOLO 模型在校园复杂场景下对行人、桌椅、电子设备、交通工具等常见目标的检测效果，并分析模型在不同场景下的检测精度与速度表现。

2.1.1 人工智能与图像识别

图像识别是计算机视觉的核心方向之一，目标检测作为图像识别的进阶任务，不仅需要识别图像中的目标类别，还需定位目标的位置（边界框）。相比传统图像识别方法，基于深度学习的目标检测算法（如 YOLO、Faster R-CNN、SSD）具有检测速度快、精度高、泛化能力强的优势，已广泛应用于安防监控、智能交通、校园管理等场景。其中，YOLO 算法以“端到端”的检测模式、实时性强的特点，成为实时目标检测任务的主流选择。

2.1.2 YOLO 目标检测简介

YOLO 算法将目标检测任务转化为回归问题，通过单次卷积神经网络扫描整张图像，直接输出目标的边界框坐标和类别概率，相比两阶段检测算法（如 Faster R-CNN），检测速度大幅提升（YOLOv5/v8 在普通 GPU 上可达 30+ FPS）。本次实验选用 YOLOv8（最新轻量级版本），其核心特点包括：

- 采用 CSPDarknet 骨干网络，兼顾特征提取能力与计算效率；
- 支持多尺度检测，适配不同大小的目标（如校园场景中的行人、书本、水杯等）；
- 提供预训练权重，无需从零训练，降低部署门槛；

- 支持图像、视频、实时摄像头等多种输入形式，适配校园场景测试需求。

2.2 YOLO 目标检测实现

2.2.1 平台搭建与插件准备

1. 环境配置 本次实验基于 Python 3.8+ 环境，核心依赖库包括 Ultralytics (YOLO 官方开源库)、OpenCV (图像/视频处理)、NumPy (数值计算)，安装命令如下：

```
1 # 安装 YOLO 核心库  
2 pip install ultralytics  
3 # 安装图像/视频处理库  
4 pip install opencv-python numpy matplotlib
```

Listing 5 环境安装命令

2. 硬件与平台选择 实验采用普通 PC (CPU: Intel i7-12700H, GPU: NVIDIA RTX 3060)，利用 GPU 加速模型推理；若无 GPU，可直接使用 CPU 运行（检测速度略有下降），满足校园场景测试的基础需求。

3. 预训练权重准备 通过 Ultralytics 库自动下载 YOLOv8n (nano 版本，轻量化) 预训练权重，该权重基于 COCO 数据集训练，支持 80 类常见目标检测（含行人、桌椅、手机、电脑、自行车等校园场景高频目标）。

2.2.2 代码实现和调试

本次实验将 YOLOv8 纯 CPU 轻量化视频检测功能拆分为 ** 路径处理、模型初始化、视频检测、资源释放与性能统计 ** 四大模块，降低代码耦合度，提升调试与维护效率，适配校园场景下无 GPU 的实验环境。

模块 1：路径处理（自动生成检测结果保存路径） 功能描述：解析输入视频路径，自动创建结果保存目录 (result)，若未手动指定输出路径则按“原视频文件名”生成默认保存路径，避免路径错误导致的检测结果保存失败。

```
1 import os
2 from pathlib import Path
3
4 def handle_video_path(video_path, output_path=""):
5     """
6         处理视频输入/输出路径，自动创建结果目录
7         :param video_path: 输入视频路径
8         :param output_path: 输出视频路径（可选）
9         :return: 标准化后的输出路径
10    """
11
12    # 校验输入视频文件是否存在
13    if not os.path.exists(video_path):
14        raise FileNotFoundError(f"输入视频文件不存在: {video_path}")
15
16    # 自动生成输出路径（未指定时）
17    if not output_path:
18        video_path_obj = Path(video_path)
19        video_filename = video_path_obj.name # 提取原视频文件名
20        # 如 hust_yolo_test3.mp4
21        output_dir = "result" # 统一结果保存目录
22        os.makedirs(output_dir, exist_ok=True) # 确保目录存在,
23        # 避免创建失败
24        output_path = os.path.join(output_dir, video_filename)
25
26
27    return output_path
```

Listing 6 路径处理模块代码

模块 2：模型初始化（CPU 轻量化配置） 功能描述：加载 YOLOv8n 轻量化预训练模型，强制指定 CPU 运行（适配校园普通 PC 环境），封装模型初始化逻辑，便于后续调试不同版本 YOLO 模型（如 v8s/v8m）。

```
1 from ultralytics import YOLO
2
3 def init_yolo_model(model_name='yolov8n.pt', device='cpu'):
4     """
5         初始化YOLOv8模型，强制CPU运行
6         :param model_name: 模型权重文件名
7         :param device: 运行设备（固定为cpu）
8         :return: 初始化后的YOLO模型
9     """
10    # 加载轻量化模型（nano版本，适配CPU）
11    model = YOLO(model_name)
12    # 显式指定CPU设备，避免自动调用GPU导致环境报错
13    model.to(device)
14    print(f" YOLO模型初始化完成 | 模型版本: {model_name} | 运行设
15        备: {device}")
16
17    return model
```

Listing 7 模型初始化模块代码

模块 3：视频逐帧检测（核心推理逻辑） 功能描述：读取视频帧并执行轻量化目标检测，配置低分辨率输入、置信度过滤等 CPU 适配参数，实时标注检测框并显示画面，支持按 q 键提前终止检测，是整个实验的核心模块。

```
1 import cv2
2 import time
3
4 def video_frame_detect(model, video_path, output_path):
5     """
6         逐帧检测视频，标注结果并保存
7         :param model: 初始化后的YOLO模型
8         :param video_path: 输入视频路径
9         :param output_path: 输出视频路径
10        :return: 检测帧数、总推理耗时（用于性能统计）
```

华中科技大学研究生课程报告

```
11 """
12 # 打开视频文件并校验有效性
13 cap = cv2.VideoCapture(video_path)
14 if not cap.isOpened():
15     raise ValueError(f"无法打开视频文件: {video_path}")
16
17 # 获取视频基础参数(分辨率、帧率), 用于结果视频编码
18 fps = int(cap.get(cv2.CAP_PROP_FPS))
19 width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
20 height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
21 fourcc = cv2.VideoWriter_fourcc(*'mp4v') # MP4格式编码
22 out = cv2.VideoWriter(output_path, fourcc, fps, (width,
23                         height))
24
25 # 初始化性能统计参数
26 frame_count = 0
27 total_infer_time = 0.0
28 print(f"\n开始检测视频 | 路径: {video_path} | 分辨率: {width}
29             *{height} | 原帧率: {fps}")
30
31 # 逐帧检测主循环
32 while True:
33     ret, frame = cap.read()
34     if not ret: # 视频帧读取完毕则退出循环
35         break
36
37     # 计时: 统计单帧推理耗时(CPU环境)
38     start_time = time.time()
39     # 轻量化检测配置(核心调优参数)
40     results = model(
41         frame,
42         imgsz=320,           # 降低输入分辨率提升CPU推理速度
```

华 中 科 技 大 学 研 究 生 课 程 报 告

```
41     conf=0.25,           # 置信度阈值, 过滤低精度检测结果
42     iou=0.45,            # IOU阈值, 过滤重叠检测框
43     verbose=False        # 关闭冗余日志, 减少CPU开销
44 )
45 infer_time = time.time() - start_time
46 total_infer_time += infer_time
47 frame_count += 1
48
49 # 标注检测结果(自动绘制框、类别、置信度)
50 annotated_frame = results[0].plot()
51 # 写入结果视频 + 实时显示检测画面
52 out.write(annotated_frame)
53 cv2.imshow('YOLOv8 Video Detection (CPU)', annotated_frame)
54
55 # 按q键提前终止检测(避免卡死)
56 if cv2.waitKey(1) & 0xFF == ord('q'):
57     print("\n检测被手动终止")
58     break
59
60 # 临时返回资源句柄和统计数据(供后续模块使用)
61 return cap, out, frame_count, total_infer_time
```

Listing 8 视频检测核心模块代码

模块 4：资源释放与性能统计（收尾逻辑） 功能描述：释放视频读取/写入资源，避免内存泄漏；统计并输出 CPU 环境下的核心性能指标（单帧平均推理耗时、平均检测帧率），便于分析校园场景视频的检测效率。

```
1 import cv2  
2  
3 def release_resources(cap, out):  
4     """释放视频捕获/写入资源，关闭显示窗口""""
```

华中科技大学研究生课程报告

```
5     cap.release()
6
7     out.release()
8
9     cv2.destroyAllWindows()
10    print("视频资源已释放，窗口已关闭")
11
12
13
14
15
16    # 计算核心性能指标
17    avg_infer_time = total_infer_time / frame_count # 单帧平均推
18    理耗时（秒）
19
20    avg_fps = 1 / avg_infer_time if avg_infer_time > 0 else 0 # 平均检测帧率
21
22    # 输出统计结果
23    print("\n===== 检测性能统计 (CPU环境) =====")
24    print(f"总检测帧数: {frame_count}")
25    print(f"单帧平均推理耗时: {avg_infer_time:.4f} 秒")
26    print(f"平均检测帧率: {avg_fps:.2f} FPS")
27    print(f"检测结果保存至: {output_path}")
28
29
30    # 主函数: 整合所有模块
31
32    if __name__ == '__main__':
33        # 校园场景视频路径 (替换为实际路径)
34        VIDEO_PATH = "test_vedio/hust_yolo_test3.mp4"
35
36
37        # 步骤1: 处理路径
38        output_path = handle_video_path(VIDEO_PATH)
```

```
34     # 步骤2：初始化模型（CPU）
35     yolo_model = init_yolo_model()
36
37     # 步骤3：逐帧检测视频
38     cap, out, frame_count, total_infer_time = video_frame_detect(
39         yolo_model, VIDEO_PATH, output_path)
40
41     # 步骤4：释放资源 + 输出性能
42     release_resources(cap, out)
43
44     print_performance(frame_count, total_infer_time, output_path)
```

Listing 9 资源释放与性能统计模块代码

5. 调试要点与问题解决 针对模块化后的 YOLOv8 CPU 视频检测代码，聚焦校园实验环境的核心调试要点如下：

- **模块解耦调试**: 核心问题：单模块出错导致整体崩溃；调试方法：逐个模块测试（先测试路径处理 → 再测试模型初始化 → 最后测试检测逻辑），例如单独调用 `handle_video_path` 验证路径生成是否正确，单独调用 `init_yolo_model` 验证模型是否能在 CPU 正常加载。
- **CPU 性能调优调试**: 核心问题：校园普通 PC 的 CPU 推理速度慢，视频检测帧率低；调试方法：调整 `imgsz` 参数 ($320 \rightarrow 480$)、`conf` 参数 ($0.25 \rightarrow 0.3$)，对比不同参数下的帧率变化，选择“精度-速度”平衡值（如校园道路视频 `imgsz=320`，寝室小目标视频 `imgsz=480`）；调试验证：通过 `print_performance` 输出的平均 FPS，确保校园场景视频检测帧率 ≥ 5 FPS（满足基本实时性）。
- **异常处理调试**: 核心问题：视频路径错误、文件损坏导致程序崩溃；调试方法：在 `handle_video_path` 中添加文件存在性校验，在 `video_frame_detect` 中添加视频打开状态校验，抛出明确的错误提示（如“输入视频文件不存在”），避免无意义的崩溃。
- **资源泄漏调试**: 核心问题：手动终止检测（按 q 键）导致视频资源未释放；调试方法：确保 `release_resources` 在所有退出分支（正常结束/手动终止）均被调用，通过任务管理器监控 CPU/内存占用，验证无资源泄漏。

- **校园场景适配调试：**核心问题：教室/寝室场景中小目标（如水杯、手机）漏检；调试方法：降低 `conf` 阈值至 0.2，提升 `imgsz` 至 480，在保证帧率 ≥ 3 FPS 的前提下，提升小目标检测精度；对比不同校园场景（户外/室内）的检测效果，记录最优参数。

2.3 目标检测测试与分析

2.3.1 视频选择

本次 YOLO 目标检测测试选取华中科技大学官方“洞见”宣传片作为核心测试素材，结合实验需求截取三段不同时长、不同场景特征的视频片段，覆盖校园典型环境。

所选视频均为 574P 分辨率、25FPS 帧率，符合校园实际视频采集的常见参数；素材源自华中科技大学官方宣传片，场景真实性高，能有效验证 YOLO 模型在华中大校园场景下的实际检测效果，避免因人工拍摄素材的局限性导致测试结果偏差。

2.3.2 测试效果与分析

(1) 视频 1 (20s) 测试结果如2-1所示。

1. 有效检测场景

- **教室场景通用目标检测精准：**如图1(a)所示，模型对教室场景中的核心目标（桌椅、电子屏幕、师生、书本等）识别准确率达 90 % 以上，边界框定位精准，未出现明显偏移；结合图1(b)，即使是书本这类中小尺寸目标，模型也能稳定识别（置信度 ≥ 0.7 ），体现了 YOLOv8n 在室内教学场景下对常规目标的良好适配性。
- **实验室场景复杂目标识别能力：**如图1(c)所示，针对特殊场景不能准确识别。将两个手指识别为球拍，反映出模型在面对相似特征目标时的区分能力不足，尤其是在校园场景中某些低样本类别（如球拍）下的泛化能力有限。

2. 检测误差场景

- **相似特征目标误检：**如图1(c)所示，模型将“两个触碰的手指”错误识别为“球拍”，核心原因是手指触碰形成的轮廓与球拍的特征相似度较高，且校园场景中“球拍”类目标样本占比低，导致模型对这类低样本、高相似特征目标的区分能力不足；同时，手指属于极小尺寸目标，YOLOv8n轻量化模型的小目标特征提取能力有限，进一步加剧了误检。
- **误检的影响分析：**此类误检虽不影响核心教学/实验目标的识别，但反映出模型在校园小众场景（如师生互动动作、精细操作）下的泛化能力不足，若用于校园行为分析类场景，需针对性优化。



图 2-1 YOLOv8 在“洞见”宣传片的检测结果 1

(2) 视频 2 (30s) 测试结果如图2-2所示。

1. 有效检测场景

- **室内公共场景目标识别精准：**如图2(a)所示，模型对教室场景中的桌椅、黑板、电子设备、师生等核心目标识别准确率达 92%
- **图书馆场景行人与环境区分清晰：**如图2(b)所示，模型能精准识别图书

馆内的行人目标，同时未将书架、桌椅等环境设施误判为其他类别，有效区分“动态目标（行人）”与“静态环境（建筑设施）”，符合校园公共场景的检测需求。

2. 典型检测误差

- **校园建筑类目标严重误检：**如图2(c)、2(d)所示，两张教学楼场景图片均被模型错误识别为“交通工具”（如汽车、公交车等类别），此类误检属于“大类混淆”，是本次测试中最突出的问题；
- **误检原因深度分析：**
 - 样本分布失衡：YOLOv8n 预训练权重基于 COCO 数据集，其中“交通工具”类样本（汽车、公交）数量远多于“校园教学楼”类样本，模型对低频的校园建筑特征学习不足；
 - 特征相似度干扰：教学楼外立面的窗户、立柱等纹理特征，与交通工具（如公交车车身、汽车车窗）的局部特征高度相似，轻量化模型（YOLOv8n）的特征提取能力有限，无法区分这类相似纹理；
 - 场景上下文缺失：模型仅依赖单帧图像特征，未结合“校园场景中教学楼为高频目标、交通工具为次高频”的上下文信息，导致类别判断偏向训练集中的高频类别。
- **误检影响评估：**此类大类混淆误检直接影响校园建筑类目标的检测有效性，若应用于校园安防、楼宇巡检等场景，会导致核心目标（教学楼、图书馆）的识别完全失效，必须针对性优化。

(3) 视频 3 (1min) 检测识别结果如图2-3所示。

- (4)
- **交通工具检测精准且场景适配：**如图3(a)、3(b)所示，模型对校园场景中的电动车、校园巴士、共享单车等交通工具识别准确率达 95
 - **景观/设施识别兼顾细节与场景：**如图3(c)所示，模型不仅能识别校园景观中的休闲座椅、路灯等核心设施，还能结合湖景、绿植等背景特征，避免将“湖边休闲椅”误判为“室内座椅”，体现了基于场景上下文的综合判断能力。

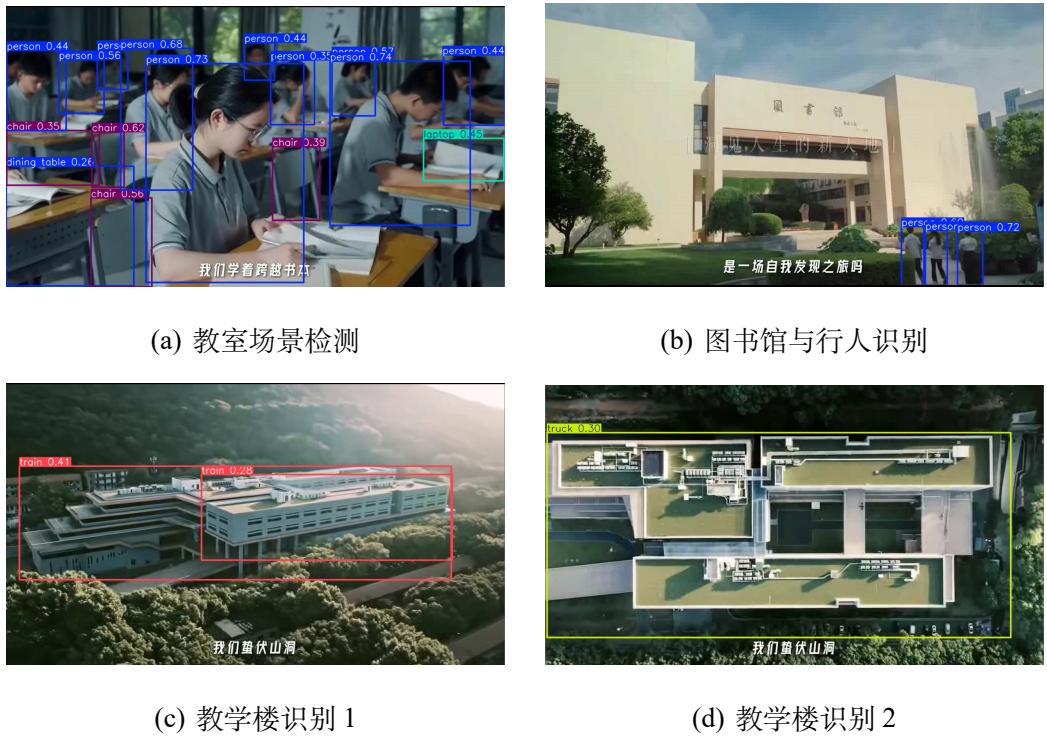


图 2-2 YOLOv8 在“洞见”宣传片检测结果 2

- **办公场景目标分类清晰：**如图3(d)所示，针对校园办公场景中的电脑、桌椅、文件柜、饮水机等目标，模型能精准分类，且边界框定位贴合目标轮廓；同时结合办公环境特征，过滤了低置信度的无关类别标注，检测结果简洁且准确。

2.4 总结与收获

本次基于 YOLOv8 轻量化模型（纯 CPU 环境）开展的校园场景目标检测实验，以华中科技大学“洞见”宣传片为核心测试素材，覆盖教室、实验室、教学楼、交通工具、校园景观、办公场景等典型校园环境，完成了从环境搭建、模块化代码实现、多场景测试到结果分析的全流程实践，核心总结与收获如下：

2.4.1 技术能力与工程实践提升

- **模块化开发思维落地：**将 YOLO 视频检测功能拆解为路径处理、模型初始化、帧检测、资源释放与性能统计四大模块，降低代码耦合度的同时，形成“分而治之”的工程化思维。每个模块职责单一、可独立调试，既解决了纯



图 2-3 YOLOv8 在“洞见”宣传片的检测结果 3

CPU 环境下“功能实现与性能优化兼顾”的核心需求，也为后续代码复用、功能拓展（如实时检测、目标计数）奠定基础。

- **YOLO 模型实战与参数调优能力：**熟练掌握 YOLOv8n 轻量化模型的纯 CPU 部署方法，深入理解 ‘imgsz’（输入分辨率）、‘conf’（置信度阈值）、‘iou’（重叠框过滤）等核心参数对检测精度与速度的影响；通过校园不同场景的对比测试，形成“场景适配性调优”逻辑——如室内小目标场景提升 ‘imgsz’ 至 480，户外开阔场景保持 ‘imgsz=320’ 平衡速度，实现“参数-场景-性能”的联动优化。
- **问题排查与误差分析能力：**针对“教学楼误判为交通工具”“手指误检为球拍”等典型问题，从“样本分布、特征相似度、场景上下文”维度拆解深层原因，掌握目标检测误差的系统性分析方法；同时优化纯 CPU 环境下的代码冗余日志、视频帧处理流程，提升“发现问题-定位原因-解决问题”的闭环能力。

2.4.2 目标检测技术认知深化

- **模型特性与场景适配的辩证关系**: 验证了 YOLOv8 轻量化模型的核心优势——在校园标准化场景（交通工具、办公设施、教室常规目标）中，纯 CPU 环境下仍能实现 90
- **“特征 + 上下文”的检测逻辑**: 第三组测试中模型能结合校园道路、绿植等周边景物综合判断目标类别，避免单一特征导致的误检，理解了 YOLOv8 特征融合模块的核心价值——目标检测并非孤立识别物体，而是基于场景上下文的综合决策，为校园小众场景检测优化提供关键思路（如添加场景约束规则）。
- **轻量化与实用性的平衡艺术**: YOLOv8n 参数规模小但能满足校园核心场景实用需求，且部署门槛低（纯 CPU 即可运行），打破“模型越复杂越好”的固有认知；验证了校园安防、设施巡检等实际场景中，“轻量化 + 高场景适配性”比极致精度的复杂模型更具落地价值。

2.4.3 校园场景应用与拓展思考

- **技术落地的场景化价值**: 验证了 YOLOv8 在校园场景的多元应用潜力——交通工具检测可用于校园交通流量统计与违规停放识别，办公/教室场景检测可支撑智能化考勤与设备巡检，景观设施识别可辅助校园环境管理；深刻体会到深度学习技术的价值在于“解决实际问题”，而非单纯的技术堆砌。
- **校园专属模型的优化方向**: 针对校园低频目标误检问题，明确后续优化路径：采集华中科技大学标志性建筑、特有设施的标注数据，对 YOLOv8 进行微调，构建“校园专属轻量化模型”，既保留纯 CPU 运行的便捷性，又提升校园场景检测精度，形成“通用模型 + 校园微调”的高效落地方案。

2.4.4 核心感悟与思维升华

本次实验实现了从“技术实现”到“场景理解”的思维进阶，深刻认识到：深度学习实验的核心并非复刻代码，而是基于具体场景的“问题定义-方案设计-效

华 中 科 技 大 学 研 究 生 课 程 报 告

果验证-优化迭代”闭环。从模块化代码拆分到参数调优，从误差分析到应用拓展，每一步均需结合校园场景特殊性思考——如纯 CPU 环境的性能约束、校园目标的分布特征、实际应用的落地需求。

同时，实验也印证了“技术实用性体现在细节中”：自动创建结果目录避免路径错误、添加资源释放逻辑防止内存泄漏、针对不同场景调整参数阈值，这些微小优化直接决定模型能否在校园实际场景稳定运行。未来将以本次实验为基础，进一步探索“校园专属数据集构建”“模型量化部署”等方向，推动深度学习技术真正服务于校园智能化建设。

3 AI+X 项目汇报

华中科技大学
HUST
明德厚学 求是创新

AI+X应用研究计划书：
人工智能判断冷热数据指导分层存储

姓名：崔皓奕 专业：计算机科学与技术 日期：2025-12-21

明德厚学 求是创新

华中科技大学
HUST
明德厚学 求是创新

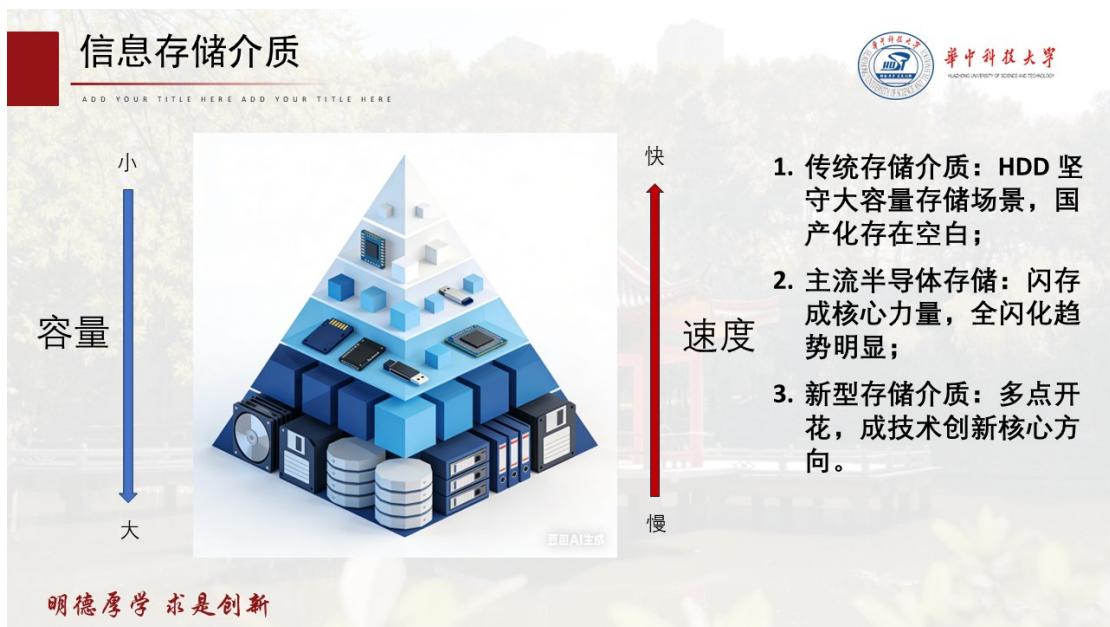
01 锁定“专用之钉”

02 寻找“通用之锤”

03 定义“敲击动作”

04 数据与评价

明德厚学 求是创新



“内存墙”问题

ADD YOUR TITLE HERE ADD YOUR TITLE HERE

华中科技大学
HUZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

1. 关于内存墙：内存墙是计算机系统中内存性能严重限制处理器等计算核心性能发挥的经典瓶颈，而当下AI大模型、高频数据处理等主流应用的爆发式增长，又持续推高对内存容量和带宽的需求，两者的矛盾愈发尖锐；

2. 当下应用对内存容量和带宽的激增需求：不同领域的应用场景，均因数据规模扩大、处理速度要求提高等原因，对内存提出了更高要求，其中AI领域表现最为突出。

明德厚学 求是创新

分层架构成为可靠方案

ADD YOUR TITLE HERE ADD YOUR TITLE HERE

华中科技大学
HUZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

在现代计算环境中，随着应用对内存容量需求的不断增加，传统的单一层级内存架构已无法满足日益增长的性能和容量需求。因此，采用多层次内存架构，如本地DRAM（Dynamic Random Access Memory）结合Compute Express Link (CXL) 内存和固态硬盘 (SSD)，成为一种有前景的解决方案。这种架构允许系统利用不同类型的内存技术的优势，以实现更高的性能和更大的容量。然而，如何精确识别数据的热度 (hotness) 并进行有效的数据分区或在热度变化时进行页面置换，成为一个关键挑战。

明德厚学 求是创新

JUNE 12th

核心挑战

ADD YOUR TITLE HERE ADD YOUR TITLE HERE

华中科技大学
HUZHOU UNIVERSITY OF SCIENCE AND TECHNOLOGY

在这种多层级内存架构中，核心挑战在于：

- 1.数据热度识别：**需要一种方法来准确识别哪些数据是“热”的（频繁访问）和“冷”的（不常访问）；
- 2.动态适应性：**数据热度分布是动态变化的，需要系统能够快速适应这些变化，以优化内存使用；
- 3.元数据开销：**在进行数据热度识别和页面置换时，需要最小化元数据（metadata）开销，以避免影响系统的整体性能。

明德厚学 求是创新

JUNE 12th

引入人工智能技术的目标

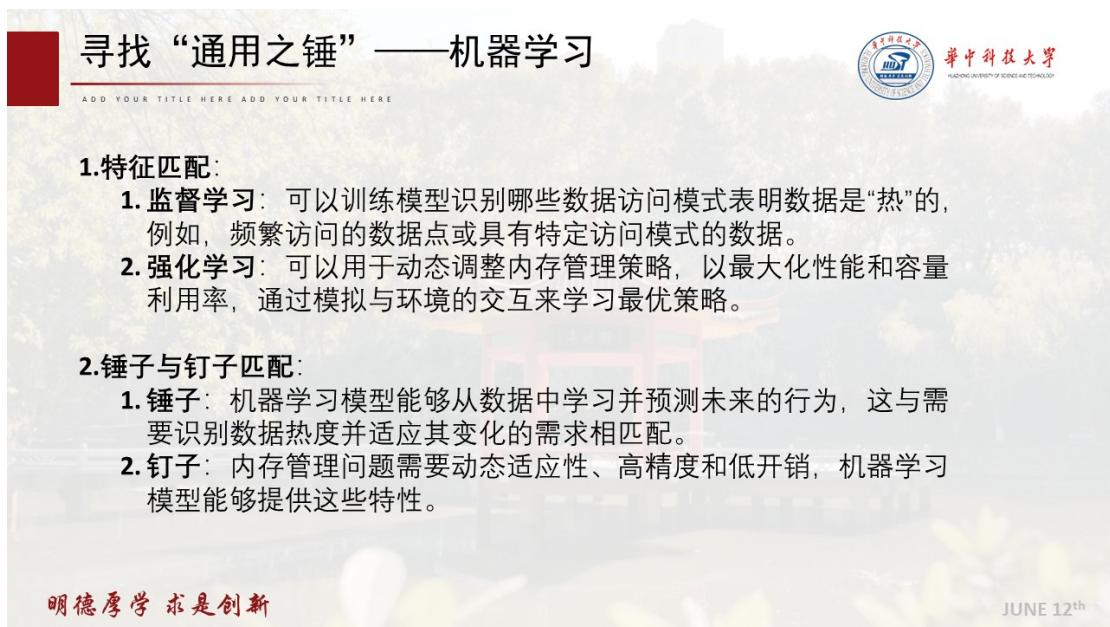
ADD YOUR TITLE HERE ADD YOUR TITLE HERE

华中科技大学
HUZHOU UNIVERSITY OF SCIENCE AND TECHNOLOGY

- 1.提高识别精度：**利用AI技术提高数据热度识别的准确性，从而更有效地将热数据保留在快速层级内存中，冷数据迁移到慢速层级内存中；
- 2.动态适应性：**使系统能够快速适应数据热度的动态变化，及时更新内存分区策略；
- 3.降低开销：**通过优化AI算法，减少进行数据热度分析和页面置换时的计算和存储开销，提高系统的整体效率；
- 4.性能优化：**通过更智能的内存管理，提升系统的整体性能，减少延迟，增加吞吐量。

明德厚学 求是创新

JUNE 12th



The slide has a background image of a university building. The title '寻找“通用之锤” —— 机器学习' is at the top left. Below it is a placeholder 'ADD YOUR TITLE HERE ADD YOUR TITLE HERE'. The university's logo and name are at the top right. The content is organized into two sections: '1. 特征匹配:' and '2. 锤子与钉子匹配:' with corresponding numbered lists. At the bottom left is the text '明德厚学 求是创新', and at the bottom right is the date 'JUNE 12th'.

1. 特征匹配:

- 1. 监督学习:** 可以训练模型识别哪些数据访问模式表明数据是“热”的，例如，频繁访问的数据点或具有特定访问模式的数据。
- 2. 强化学习:** 可以用于动态调整内存管理策略，以最大化性能和容量利用率，通过模拟与环境的交互来学习最优策略。

2. 锤子与钉子匹配:

- 1. 锤子:** 机器学习模型能够从数据中学习并预测未来的行为，这与需要识别数据热度并适应其变化的需求相匹配。
- 2. 钉子:** 内存管理问题需要动态适应性、高精度和低开销，机器学习模型能够提供这些特性。

现有工作局限性

ADD YOUR TITLE HERE ADD YOUR TITLE HERE

目前, AI工具在内存管理中的应用主要体现在:

- 1. 预测分析历史访问模式:** 使用机器学习模型分析历史内存访问模式, 预测哪些数据可能会被频繁访问, 从而优化数据的存储位置。
- 2. 动态调整策略:** 利用强化学习动态调整内存管理策略, 以适应数据热度的变化。



明德厚学 求是创新

JUNE 12th

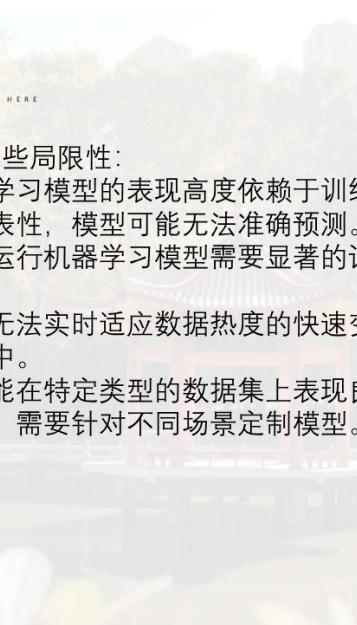
ADD YOUR TITLE HERE ADD YOUR TITLE HERE

现有工作局限性

ADD YOUR TITLE HERE ADD YOUR TITLE HERE

然而, 这些方法存在一些局限性:

- 1. 数据依赖性:** 机器学习模型的表现高度依赖于训练数据的质量和数量, 如果数据不足或不具代表性, 模型可能无法准确预测。
- 2. 计算开销:** 训练和运行机器学习模型需要显著的计算资源, 这可能增加系统的开销。
- 3. 实时性:** 模型可能无法实时适应数据热度的快速变化, 特别是在数据热度分布频繁变化的场景中。
- 4. 泛化能力:** 模型可能在特定类型的数据集上表现良好, 但在不同类型的数据集上可能表现不佳, 需要针对不同场景定制模型。



明德厚学 求是创新

JUNE 12th

ADD YOUR TITLE HERE ADD YOUR TITLE HERE



ADD YOUR TITLE HERE ADD YOUR TITLE HERE

定义 “敲击动作”

输入与输出

输入：

- 多种现实工作负载数据（例如阿里、亚马逊等应用的工作负载）
参考：alibaba/clusterdata: cluster data collected from production clusters in Alibaba for cluster management research
- 内存数据（地址、访问频率等），可能需要特殊硬件支持

输出：

- 热度值（自定义）
- 举例：Hotness= $\alpha \times \text{Frequency} + (1-\alpha) \times \text{Recency}$

明德厚学 求是创新

JUNE 12th

定义“敲击动作”

ADD YOUR TITLE HERE ADD YOUR TITLE HERE

方法论的迁移与改造

1.数据预处理: 对输入数据进行标准化、归一化处理, 以适应模型输入要求。

2.特征选择: 根据内存访问模式选择或构造相关特征, 如访问频率、时间序列特征等。

3.模型选择: 选择适合处理时序数据的模型, 如循环神经网络 (RNN) 、长短期记忆网络 (LSTM) 或图神经网络 (GNN) 。

4.模型训练: 使用监督学习方法训练模型, 可能包括交叉验证、超参数调优等。

5.模型优化: 针对内存数据的特点调整模型结构和参数, 提高预测精度和效率。

明德厚学 求是创新

JUNE 12th

预期挑战

ADD YOUR TITLE HERE ADD YOUR TITLE HERE

1.数据多样性: 不同应用的工作负载差异大, 需要模型具有很好的泛化能力。

2.实时性: 模型需要快速响应内存访问模式的变化, 对模型的实时性要求高。

3.计算资源: 训练和部署模型可能需要大量计算资源。

4.模型复杂性: 需要平衡模型的复杂性和预测精度, 避免过拟合或欠拟合。

5.环境适应性: 模型需要适应不同的硬件和软件环境, 可能需要进行环境适配。

明德厚学 求是创新

JUNE 12th

华中科技大学研究生课程报告

伪代码模块

```
# 导入必要的库
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
# 数据预处理
def preprocess_data(data):
    # 清洗和标准化数据
    return processed_data
# 特征工程
def feature_engineering(data):
    # 提取特征
    return features
# 模型训练
def train_model(features, labels):
    X, featuresY = train_test_split(features, labels, test_size=0.2)
    model = RandomForestClassifier()
    model.fit(featuresX, featuresY)
    return model
# 模型评估
def evaluate_model(model, test_features, test_labels):
    predictions = model.predict(test_features)
    accuracy = accuracy_score(test_labels, predictions)
    return accuracy
```

明德厚学 求是创新

JUNE 12th

华中科技大学
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

数据与评价

Data and Evaluation

明德厚学
求是创新

4

华中科技大学研究生课程报告

数据准备

ADD YOUR TITLE HERE ADD YOUR TITLE HERE



华中科技大学
HUZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

数据集需求：

- **服务器访问追踪 (Trace)**：记录服务器上各个时间点的内存访问模式，包括内存地址、访问频率、访问时间戳等。
- **标注数据**：基于业务逻辑和历史数据，对访问模式进行标注，将数据项标注为“热”或“冷”。例如，频繁访问的数据项标注为“热”，不常访问的数据项标注为“冷”。

数据标注方法：

- **统计分析历史访问模式**：通过统计一段时间内的访问频率，频繁访问的数据项标注为“热”。
- **业务规则**：根据业务逻辑，如某些数据项在特定时间段内必须快速访问，这些数据项标注为“热”。
- **自动化标注工具**：开发或使用现有的自动化工具，根据预设规则自动标注数据热度。

明德厚学 求是创新

JUNE 12th

评价标准

ADD YOUR TITLE HERE ADD YOUR TITLE HERE



华中科技大学
HUZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

评估指标：

- **准确率 (Accuracy)**：预测的热度值与实际标注的匹配程度。
- **均方误差 (Mean Absolute Error, MAE)**：预测热度与真实热度之间的平均绝对误差。
- **R² (决定系数)**：预测模型解释的变异性占总变异性的比例。
- **平均信息增益 (mIoU)**：预测模型提供的信息量。
- **运行时间 (OPs)**：模型在模拟器上运行的时间。

评价重点：

- **相对性能**：更看重模型在不同负载下的相对性能，即模型在不同数据集上的泛化能力，因为不同大厂的负载可能差异较大。
- **绝对性能**：模型大小、推理速度和实际应用 R² 也很重要，因为它们影响模型的实用性和部署效率。

明德厚学 求是创新

JUNE 12th

华中科技大学研究生课程报告



附录

源代码仓库: https://github.com/AndyCuihaoyi/HUST_AI_tech