人工智能编程大作业实验报告：Task 3 自定义卷积网络框架与 PyTorch 性能对比

学号：2400013169　姓名：董彦嘉

2026 年 1 月 16 日

摘要

本实验旨在脱离现有的深度学习框架，基于 CUDA、pybind11 和 Python 自主实现一个卷积神经网络框架 `mytensor`。实验成功实现了卷积层的前向传播与反向传播算子，构建了自动微分机制，并在 CIFAR-10 数据集上完成了图像分类任务。结果显示，自定义框架达到了 54.07% 的测试准确率。虽然成功验证了底层算法的正确性，但在执行效率（比 PyTorch 慢约 8%）和收敛稳定性上与工业级框架仍存在差距。本报告详细对比了两者在算子优化、内存管理及数值稳定性方面的差异。

1　实验背景

本次实验旨在通过实现 CIFAR-10 数据集的图像分类任务，深入理解深度学习框架的底层原理 [2]。实验分为两个主要部分：

1. 使用工业级框架 PyTorch 进行标准实现，作为性能基准（Benchmark）[7]。
2. 基于 CUDA、pybind11 和 Python 自主实现一个卷积网络框架（`mytensor`），并完成前向传播、反向传播及自动微分逻辑 [36, 38]。

2　实验方法与设置

2.1　网络架构

为了公平对比，实验在两个框架中均采用类 LeNet 结构 [37]：

- **卷积层 1**: 3 输入通道, 6 输出通道, 3×3 卷积核, padding=1。
- **池化层 1**: 2×2 Max Pooling。

- **卷积层 2**: 6 输入通道, 16 输出通道, 3×3 卷积核, padding=1。
- **池化层 2**: 2×2 Max Pooling。
- **全连接层**: 三层 MLP（多层感知机），最终输出 10 类概率。

2.2 训练配置

- **硬件环境**: 使用支持 CUDA 的 GPU 进行加速训练 [41]。
- **超参数**: Batch Size = 64，Momentum = 0.9。
- **学习率**: 初始学习率 0.01，第 10 轮后衰减至 0.001。
- **数据集**: CIFAR-10 训练集用于参数优化，测试集用于评估准确率。

3 实验结果对比

根据实际运行输出，汇总性能指标如表 1 所示：

表 1: 自定义框架 (mytensor) 与 PyTorch 框架性能对比

指标	自定义框架 (mytensor)	PyTorch 框架	差异 (PyTorch 为基准)
训练总耗时 (s)	327.35	303.09	+8.0%
最终测试准确率	54.07%	66.78%	-12.71%
收敛平稳性	波动较大	相对平稳	-

4 讨论与分析

4.1 执行效率分析

PyTorch 的训练速度比自定义框架快约 24.26 秒（约 8%）。分析原因如下：

- **算子优化**: PyTorch 底层调用了高度优化的 cuDNN 库，而自定义 CUDA 实现采用了基础的 im2col 算法，在内存访问模式（Memory Access Pattern）和线程块调度上存在提升空间 [43]。
- **内存管理**: PyTorch 拥有成熟的显存池（Caching Allocator）管理机制，减少了频繁申请和释放显存的系统调用开销。
- **数据交互**: 自定义代码中存在 `inputs.numpy()` 调用，这在每一批次都引入了额外的 CPU-GPU 数据拷贝开销。

4.2 准确率与收敛分析

PyTorch 最终达到了 66.78% 的准确率，显著优于自定义框架。

- **权重初始化**: PyTorch 默认使用 Kaiming 或 Xavier 初始化，适配 ReLU 激活函数；而自定义框架使用了简单的随机均匀分布，导致初期训练梯度传递效率较低。
- **数值稳定性**: PyTorch 的 `CrossEntropyLoss` 在数学上进行了 Log-Sum-Exp 技巧优化，避免了溢出风险；自定义框架的 Loss 值（0.03-0.09）与 PyTorch（0.2-2.3）存在数量级差异，说明 Loss 函数的缩放比例或计算细节不一致。

5 结论

本实验成功复现了卷积网络在两个不同层级框架上的训练过程 [53]。实验证明，自主实现的 CUDA 算子与自动微分框架能够完成基本的分类任务，但在极端性能优化和数值稳定性方面与成熟框架仍有差距。未来的改进方向应集中在 CUDA kernel 的共享内存（Shared Memory）优化以及更精细的参数初始化策略上。

A 附录：核心代码实现

A.1 CUDA 卷积算子 (im2col 示意)

```
__global__ void im2col_kernel(const float* data_im, float* data_col,
                              int channels, int height, int width,
                              int ksize, int pad, int stride,
                              int height_col, int width_col) {
    // 计算全局索引
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    const int n = height_col * width_col * channels;

    if (index < n) {
        int w_out = index % width_col;
        int h_out = (index / width_col) % height_col;
        int c_in = index / width_col / height_col;

        // ... (im2col 坐标映射逻辑) ...
    }
}
```

A.2 Python 自动微分调用

```
class Conv2D(Module):
    def __init__(self, in_channels, out_channels, ksize):
        # 权重初始化（随机均匀分布）
        self.weight = Tensor.uniform(out_channels, in_channels, ksize,
    ksize)

    def forward(self, x):
        # 调用 C++ 扩展（mytensor_backend）
        return mytensor_backend.conv2d(x, self.weight)
```