人工智能中的编程大作业：Task 3 报告

自定义卷积网络框架 (mytensor) 实现与评估

学号：2400013169

姓名：董彦嘉

2026 年 1 月 17 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`。实验成功实现了基于 im2col + GEMM 的卷积算子、池化层、激活函数及 Softmax 交叉熵损失函数，并在 Python 端构建了自动微分机制。在 CIFAR-10 数据集上的测试结果显示，自定义框架达到了 56.76% 的测试准确率。此外，本实验还利用该框架在 Tiny ImageNet 数据集上完成了图像分类任务（Bonus），验证了框架的扩展性。尽管功能完整，但相比工业级框架 PyTorch，自定义框架在训练速度（慢约 35%）和最终精度上仍有优化空间。

1 实验背景

本次实验的核心目标是深入理解深度学习框架的底层原理。不同于调用现成的 API，本次任务要求从底层算子出发，使用 CUDA 编写 GPU 加速的核心计算逻辑，利用 pybind11 实现 C++ 与 Python 的交互，并最终在 Python 端构建自动微分引擎和优化器。

2 方法与实现

2.1 框架架构

自定义框架 `mytensor` 采用经典的分层设计：

- **后端 (Backend):** 使用 CUDA C++ 实现。核心计算采用 `im2col` 将卷积转化为矩阵乘法，并利用 cuBLAS 库进行加速。实现了 `Tensor` 类进行显存管理，利用 `std::shared_ptr` 和自定义删除器自动释放 GPU 内存。

- **绑定层 (Binding):** 使用 pybind11 将 CUDA 编译为 Python 可调用的 `.pyd` 扩展模块，导出了 `Tensor`, `Conv2d`, `ReLU` 等核心类。
- **前端 (Frontend):** 使用 Python 实现计算图逻辑。定义了 `Variable` 类封装数据与梯度，实现了 `Function` 基类及其子类（如 `Conv2dFunction`）来构建动态计算图并支持自动反向传播。

2.2 核心算子实现

卷积层采用 `im2col` 算法实现：

```
// tensor_lib.cu: im2col_kernel_batch
// 将图片卷积窗口展平成列，以便将卷积转化为矩阵乘法
__global__ void im2col_kernel_batch(...) {
    // ... (索引计算逻辑)
    if (im_row >= 0 && im_row < H && im_col >= 0 && im_col < W) {
        val = data_im[src_idx];
    }
    data_col[dst_idx] = val;
}
```

反向传播时，利用 `col2im` 和 `atomicAdd` 处理梯度聚合，确保了重叠区域梯度的正确计算。

2.3 网络架构

实验采用类 LeNet 的结构进行 CIFAR-10 分类，包含两层 3x3 卷积（Padding=1）、两层 2x2 最大池化和三层全连接层。对于 Bonus 任务（Tiny ImageNet），加深了网络层数并使用了 Kaiming 初始化策略。

3 实验结果对比

3.1 CIFAR-10 性能对比

我们将自定义框架 `mytensor` 与标准 PyTorch 实现进行了对比。训练参数统一为：Batch Size 64，SGD 优化器，Momentum 0.9，训练 30 Epochs，并在第 15 Epoch 进行学习率衰减。

表 1: 自定义框架 (mytensor) 与 PyTorch 框架性能对比 (CIFAR-10)

指标	自定义框架	PyTorch	差异分析
训练总耗时 (s)	1470.50	1089.53	慢约 35%
单 Epoch 耗时 (s)	∼49.0	∼36.3	-
最终训练 Loss	1.2491	0.5555	收敛较慢
最终测试准确率	56.76%	78.85%	低 22.09%

3.2 Bonus: Tiny ImageNet 分类

在 Task 4 (Bonus) 中，我们使用 `mytensor` 对 Tiny ImageNet 数据集（200 类）进行了训练。

- **数据处理：**自定义了 `TinyImageNetValDataset` 处理验证集加载，并修复了灰度图转 RGB 的问题。
- **训练结果：**经过 50 个 Epoch 的训练，模型达到了 **43.38%** 的验证集准确率。训练速度稳定在 175-183 img/s。

4 讨论与分析

4.1 执行效率分析

实验结果显示，PyTorch 显著快于自定义框架。主要原因包括：

- **显存分配优化**: `mytensor` 在每次前向/反向传播中频繁调用 `cudaMalloc/cudaFree`，产生大量系统开销；而 PyTorch 使用 Caching Allocator 复用显存。
- **算子优化**: PyTorch 的卷积底层调用 cuDNN，针对不同尺寸选择最优算法（如 Winograd）；自定义实现仅使用了基础的 `im2col` + `cublasSgemm`，且未充分利用 Shared Memory 进行访存优化。

4.2 准确率差异分析

自定义框架准确率 (56.76%) 低于 PyTorch (78.85%)，原因可能在于：

- **初始化**: PyTorch 默认使用 Kaiming 初始化适配 ReLU，而 `mytensor` 在 CIFAR-10 任务中使用了简单的均匀分布初始化，导致收敛较慢。在 ImageNet 任务中引入 Kaiming 初始化后情况有所改善。

- **数据增强**: PyTorch 的实现中包含了丰富的数据增强流程，而自定义框架虽然在 Python 端通过 torchvision 实现了增强，但数据加载与 GPU 计算的流水线（Pipeline）并行度不如 PyTorch 的 DataLoader 高效。

5 如何运行

请按照以下步骤复现实验结果：

1. **环境准备：** 确保系统已安装 CUDA Toolkit 和 PyTorch（仅用于获取依赖库路径和数据下载）。

2. **编译扩展：** 在运行训练脚本前，必须先编译 C++ 扩展。为了简化流程，请直接运行项目根目录下的 `heading.ipynb` 文件。该 Notebook 包含了编译 `setup.py` 的必要代码：

   ```
   !python setup.py install
   ```

   运行 `heading.ipynb` 将自动编译 `src/tensor_lib.cu` 和 `src/bindings.cpp` 并安装 `mytensor` 库到当前环境。

3. **运行 Task 3 (CIFAR-10):** 执行以下命令开始训练自定义框架版本：

   ```
   python Task3/train.py
   ```

   如需运行 PyTorch 对比版本，请执行：

   ```
   python Task3/train_pytorch.py
   ```

4. **运行 Bonus (Tiny ImageNet):** 确保数据已下载至 `./data/tiny-imagenet-200`，然后执行：

   ```
   python Task3/train_new.py
   ```