人工智能中的编程——第一次作业 Part2

问题描述

实现 `Tensor` 类（包含构造函数和析构函数）。初始化方式为 `Tensor tensor(shape, device);`，并实现 `cpu()` 与 `gpu()` 两个成员函数，调用方式为：

```
Tensor c = tensor.cpu();
Tensor g = tensor.gpu();
```

在实现 `Tensor` 类的基础上，对 `Tensor` 实例计算 `ReLU` 和 `Sigmoid` 的正向传播和反向传播。

其中，`ReLU` 函数的正向传播定义为：

$$ReLU(x) = max(0, x)$$

`ReLU` 函数的反向传播定义为：

$$\frac{\partial \mathcal{L}}{\partial x_i} = \begin{cases} \frac{\partial \mathcal{L}}{\partial y_i}, & \text{if } x_i > 0 \\ 0, & \text{otherwise} \end{cases}$$

其中：

- $\mathcal{L}$ 为损失函数
- x_i 为输入
- y_i 为 `ReLU` 函数的输出

`Sigmoid` 函数的正向传播定义为：

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

`Sigmoid` 函数的反向传播定义为：

$$\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial \mathcal{L}}{\partial y_i} \cdot y_i \cdot (1 - y_i)$$

其中

- $\mathcal{L}$ 为损失函数
- x_i 为输入
- y_i 为 `Sigmoid` 函数的输出

测试方法

本项目使用`CMake`构建，使用`Google Test`进行单元测试，我已经为项目内**所有**模块编写了单元测试，`CPU`代码实现可以通过`GitHub Action`自动测试，也可以手动编译进行测试。

编译

```
mkdir build
cd build
cmake -DTEST=ON -DCUDA=OFF ..
make
```

- `-DTEST`：是否开启测试，默认为`OFF`
- `-DCUDA`：是否开启`CUDA`支持，默认为`OFF`

测试

编译后请进入`build`目录，执行：

```
ctest --verbose --output-on-failure -C Debug -T test
```

如需进行`GPU`代码测试，请重新编译：

```
cmake -DTEST=ON -DCUDA=ON ..
make
ctest --verbose --output-on-failure -C Debug -T test
```

实现

文件结构

```
.
├── .github
│   └── workflows
│       └── unittest.yml                # GitHub Action自动测试
├── CMakeLists.txt
├── LICENSE
├── README.md
├── pyproject.toml                      # 项目数据
├── csrc                                # C++源码
│   ├── CMakeLists.txt
│   ├── core
│   │   ├── CMakeLists.txt
│   │   ├── device
```

```
│   │   │   ├── CMakeLists.txt
│   │   │   ├── device.cpp              # device类
│   │   │   ├── device.h
│   │   │   └── tests                   # 测试代码
│   │   ├── kernels
│   │   │   ├── CMakeLists.txt
│   │   │   ├── activation
│   │   │   │   ├── CMakeLists.txt
│   │   │   │   ├── relu.cpp            # relu-cpu
│   │   │   │   ├── relu.cu             # relu-gpu
│   │   │   │   ├── relu.h
│   │   │   │   ├── sigmoid.cpp         # sigmoid-cpu
│   │   │   │   ├── sigmoid.cu          # sigmoid-gpu
│   │   │   │   ├── sigmoid.h
│   │   │   │   └── tests               # 测试代码
│   │   │   ├── ops.cpp                 # CPU数组算子
│   │   │   ├── ops.cu                  # GPU数组算子
│   │   │   └── ops.h
│   │   └── memory
│   │       ├── CMakeLists.txt
│   │       ├── memory.cpp              # CPU内存操作
│   │       ├── memory.cu               # GPU内存操作
│   │       ├── memory.h
│   │       └── tests                   # 测试代码
│   ├── error
│   │   └── error.h                     # 错误处理，定义异常
│   ├── macros.h                        # 项目宏定义
│   └── tensor
│       ├── CMakeLists.txt
│       ├── operators
│       │   ├── CMakeLists.txt
│       │   ├── tensor_activation.cpp   # 实现对tensor激活函数
│       │   ├── tensor_activation.h
│       │   └── tests                   # 测试代码
│       ├── tensor.cpp                  # 实现Tensor类
│       ├── tensor.h
│       └── tests                       # 测试代码
├── third_party                         # 第三方库
├── docs                                # 作业文档
│   └── HW01-helper.md
└── tinytorch                           # Python包(TODO)
    └── __init__.py
```

模块详情

下面模块均在`csrc`目录下：

- `core` 核心模块，复用次数较多
  - `device` 设备类，用于管理CPU和GPU

    - `memory` 内存模块，包含CPU和GPU的内存操作
    - `kernels` 算子模块，包含CPU和GPU的算子
- `error` 错误处理模块
- `tensor` Tensor类模块
    - `operators` 算子模块，包含Tensor上的激活函数
    - `tensor` Tensor类

Device类

类声明如下，用于管理CPU和GPU：

```
struct BaseDevice {
    virtual bool is_cpu() const = 0;
    virtual bool is_gpu() const = 0;
};

struct CPU : public BaseDevice {
    bool is_cpu() const override;
    bool is_gpu() const override;
};

struct GPU : public BaseDevice {
    bool is_cpu() const override;
    bool is_gpu() const override;
};
```

使用 `struct` 实现主要是考虑了不同设备上的算子不同，为了编译为同一函数，方便接口调用。

在不同设备上使用同类操作，例如 `relu` 时，需要向 `relu` 函数传入 `device` 对象，以便在函数内部判断设备类型，调用不同的算子。

好处是我可以使用完全相同的接口，并且增加了跨平台的兼容性。若主机只有 `CPU`，编译时会自动忽略 `GPU` 相关代码；若主机有 `Nvidia GPU`，则会编译 `GPU` 相关代码。

memory模块

`memory` 模块用于管理CPU和GPU的内存操作，包括内存分配、释放、拷贝等。

接口示例：

```
template <typename Tp, typename Device>
struct malloc_mem_op {
    /// @brief memory allocation for multi-device
    ///
    /// Inputs:
    /// @param device  : the type of device
```

```
    /// @param p_data  : the input pointer
    /// @param size    : the size of the memory
    void operator()(const Device* device, Tp*& p_data, const size_t size);
};
```

这里我使用重载`()`运算符的模板类实现了内存算子，这样我可以使用如下方式调用：

```
device::CPU* cpu {};
float* data;
size_t size = 10;

malloc_mem_op<float, device::CPU>()(cpu, data, size);
```

kernels模块

该模块中我实现了`ReLU`与`Sigmoid`的正向传播和反向传播。

同样的，我在头文件中声明了接口，用上面相同方式将`CPU`与`GPU`实现分别写在`.cpp`文件与`.cu`文件中。

举例说明：

接口：

```
template <typename Tp, typename Device>
struct relu_forward {
    /// @brief relu forward operator for multi-device
    ///
    /// Inputs:
    /// @param device : the type of device
    /// @param output : the output array pointer
    /// @param input  : the input array pointer
    /// @param size   : the size of the array
    void operator()(Device* device, Tp* output, Tp* input, size_t size);
};
```

cpu实现：

```
template <typename Tp>
struct relu_forward<Tp, device::CPU> {
    void operator()(
        device::CPU* device,
        Tp* output,
        Tp* input,
        size_t size
    ) {
        for (int i = 0; i < size; ++i) {
```

```
            output[i] = input[i] > 0 ? input[i] : 0;
        }
    }
};
```

gpu实现：

```
template <typename Tp>
__global__ void
kernel_relu_f(Tp* output, Tp* input, size_t size) {
    CUDA_KERNEL_LOOP(i, size) {
        output[i] = input[i] > 0 ? input[i] : 0;
    }
}

template <typename Tp>
struct relu_forward<Tp, device::GPU> {
    void operator()(
        device::GPU* device,
        Tp* output,
        Tp* input,
        size_t size
    ) {
        kernel_relu_f<Tp><<<CUDA_GET_BLOCKS(size), CUDA_K_THREADS>>>(output,
input, size);
    }
};
```

Tensor类

我按照题目要求实现了`Tensor`类，包含构造函数、析构函数、`cpu()`与`gpu()`两个成员函数。

具体实现可阅读代码`/csrc/tensor/tensor.h`与`/csrc/tensor/tensor.cpp`。

Tensor类的实现并无特色，这里不再赘述。