人工智能中的编程——第二次作业

问题描述

- 实现全连接层的正向传播和反向传播
- 实现卷积层的正向传播和反向传播
  - 实现 `im2col` 与 `col2im`
- 实现 `max_pool_forward` 与 `max_pool_backward`
- 实现 `softmax_forward`
- 实现 `cross_entropy_forward` 与 `cross_entropy_backward` (with `softmax`)

实现

本轮作业的实现主要在 `csrc/layers/` 与 `csrc/core/kernels` 中。

我的实现分为两个层面：

- 在 `csrc/core/kernels` 中实现指针层面的操作，定义模板结构体来实现不同设备异构计算的统一，可参考 `csrc/core/kernels/ops.h` 与其对应的 `cpp`、`cu` 文件，还可阅读 `csrc/core/kernels/functions` 文件夹内代码
- 在 `csrc/layers/` 中实现张量层面的操作，可参考 `csrc/layers/layers.h` 与其对应的 `cpp` 文件，该层面中调用 `csrc/core/kernels` 中的模板结构体
  - 由于不同的设备的异构计算算子在第一层抽象中已经统一，因此在该层面中不需要再通过 `cpp` 与 `cu` 文件来区分设备类型，仅需使用 `tensor` 类管理设备类型，并调用 `csrc/core/kernels` 中的模板结构体即可

全连接层

签名在 `csrc/layers/layers.h` 中定义：

```cpp
/**
 * @brief forward function for fully connected layer
 *        - Y = XW + b
 */
template <typename Tp>
void fc_forward(
    const tensor::Tensor<Tp>& input,    // X(batch_size, in_features)
    const tensor::Tensor<Tp>& weight,   // W(in_features, out_features)
    const tensor::Tensor<Tp>& bias,     // b(out_features)
    tensor::Tensor<Tp>& output          // Y(batch_size, out_features)
);

/**
 * @brief backward function for fully connected layer
 *        - dX = dY * W^T
 *        - dW = X^T * dY
 *        - db = \sum dY
 */
template <typename Tp>
void fc_backward(
```

```
    const tensor::Tensor<Tp>& input,        // X(batch_size, in_features)
    const tensor::Tensor<Tp>& weight,       // W(in_features, out_features)
    const tensor::Tensor<Tp>& bias,         // b(1, out_features)
    const tensor::Tensor<Tp>& output,       // Y(batch_size, out_features)
    tensor::Tensor<Tp>& grad_input,         // dX(batch_size, in_features)
    tensor::Tensor<Tp>& grad_weight,        // dW(in_features, out_features)
    tensor::Tensor<Tp>& grad_bias,          // db(1, out_features)
    const tensor::Tensor<Tp>& grad_output   // dY(batch_size, out_features)
);
```

函数实现在`csrc/layers/fc_layer.cpp`中，测试在`csrc/layers/tests/test_layer.cpp`中

测试样例使用`pytorch`生成，生成脚本保存在`csrc/layers/tests/`文件夹中

卷积层

签名在`csrc/layers/layers.h`中定义：

```
/**
 * @brief forward function for conv2d layer
 *        - Y = W conv X + b
 */
template <typename Tp>
void conv2d_forward(
    const tensor::Tensor<Tp>& input,    // X(batch_size, in_channels, height, width)
    const tensor::Tensor<Tp>& weight,   // W(out_channels, in_channels, kernel_h, kernel_w)
    const tensor::Tensor<Tp>& bias,     // b(out_channels)
    tensor::Tensor<Tp>& output,         // Y(batch_size, out_channels, height_out, width_out)
    const int pad_h,
    const int pad_w,
    const int stride_h,
    const int stride_w
);

/**
 * @brief backward function for conv2d layer
 *        - dX = dY conv W^T
 *        - dW = dY conv X
 *        - db = \sum dY
 */
template <typename Tp>
void conv2d_backward(
    const tensor::Tensor<Tp>& input,        // X(batch_size, in_channels, height, width)
    const tensor::Tensor<Tp>& weight,       // W(out_channels, in_channels, kernel_h, kernel_w)
    tensor::Tensor<Tp>& grad_input,         // dX(batch_size, in_channels, height, width)
    tensor::Tensor<Tp>& grad_weight,        // dW(out_channels, in_channels, kernel_h, kernel_w)
    tensor::Tensor<Tp>& grad_bias,          // db(1, out_channels)
```

```
    const tensor::Tensor<Tp>& grad_output,  // dY(batch_size, out_channels, height_out, width_out)
    const int pad_h,
    const int pad_w,
    const int stride_h,
    const int stride_w
);
```

函数实现在`csrc/layers/conv2d_layer.cpp`中，测试在`csrc/layers/tests/test_layer.cpp`中

其指针层面计算在`csrc/core/kernels/ops.h`中定义，可参考`csrc/core/kernels/ops.h`及其对应的`cpp`、`cu`文件

`im2col`与`col2im`

签名在`csrc/core/kernels/ops.h`中定义，可参考`csrc/core/kernels/ops.h`及其对应的`cpp`、`cu`文件

对应测试在`csrc/core/kernels/tests/test_ops_cpu.cpp`与`csrc/core/kernels/tests/test_ops_gpu.cu`中

测试样例使用`pytorch`生成，生成脚本及对应样例保存在`csrc/core/kernels/tests/`文件夹中

池化层

签名在`csrc/layers/layers.h`中定义：

```
/**
 * @brief forward function for max pooling layer
 */
template <typename Tp>
void max_pool_forward(
    const tensor::Tensor<Tp>& input,    // X(batch_size, channels, height, width)
    tensor::Tensor<int>& mask,          // mask(batch_size, channels, height_out, width_out)
    tensor::Tensor<Tp>& output,         // Y(batch_size, channels, height_out, width_out)
    const int kernel_h,
    const int kernel_w,
    const int pad_h,
    const int pad_w,
    const int stride_h,
    const int stride_w
);

/**
 * @brief backward function for max pooling layer
 */
template <typename Tp>
void max_pool_backward(
    tensor::Tensor<Tp>& grad_input,        // dX(batch_size, channels, height, width)
    const tensor::Tensor<int>& mask,       // mask(batch_size, channels, height_out, width_out)
```

```
    const tensor::Tensor<Tp>& grad_output, // dY(batch_size, channels, height_out, width_out)
    const int kernel_h,
    const int kernel_w,
    const int pad_h,
    const int pad_w,
    const int stride_h,
    const int stride_w
);
```

函数实现在`csrc/layers/pooling_layer.cpp`中，测试在`csrc/layers/tests/test_layer.cpp`中

其指针层面计算在`csrc/core/kernels/ops.h`中定义，可参考`csrc/core/kernels/pooling_ops.h`及其对应的`cpp`、`cu`文件

测试样例使用`pytorch`生成，生成脚本及对应样例保存在`csrc/layers/tests/`文件夹中

`softmax`层

签名在`csrc/layers/layers.h`中定义：

```
/**
 * @brief forward function for softmax layer
 *        - Y = softmax(X)
 */
template <typename Tp>
void softmax_forward(
    const tensor::Tensor<Tp>& input,    // X(batch_size, num_classes)
    tensor::Tensor<Tp>& output          // Y(batch_size, num_classes)
);
```

函数实现在`csrc/layers/softmax_layer.cpp`中，测试在`csrc/layers/tests/test_layer.cpp`中

其指针层面计算在`csrc/core/kernels/functions/softmax.h`中定义，可参考`csrc/core/kernels/functions/softmax.h`及其对应的`cpp`、`cu`文件

对应测试在`csrc/core/kernels/tests/test_functions_cpu.cpp`与`csrc/core/kernels/tests/test_functions_gpu.cu`中

测试样例使用`pytorch`生成，生成脚本保存在`csrc/core/functions/tests`文件夹中

cross_entropy层

签名在`csrc/layers/layers.h`中定义：

```
/**
 * @brief loss function for cross entropy
 *       - loss = -\sum y_i * log(p_i)
 */
template <typename Tp>
void cross_entropy_forward(
    const tensor::Tensor<Tp>& input,    // X(batch_size, num_classes)
    const tensor::Tensor<int>& target,  // t(batch_size)
    tensor::Tensor<Tp>& output          // z(1)
);
```

```
11
12 /**
13  * @brief backward function for cross entropy
14  *        - dX_i = p_i - y_i
15  */
16 template <typename Tp>
17 void cross_entropy_backward(
18     const tensor::Tensor<Tp>& input,    // X(batch_size, num_classes)
19     const tensor::Tensor<int>& target,  // t(batch_size)
20     tensor::Tensor<Tp>& grad            // dX(batch_size, num_classes)
21 );
```

函数实现在`csrc/layers/softmax_layer.cpp`中，测试在`csrc/layers/tests/test_layer.cpp`中

其指针层面计算在`csrc/core/kernels/functions/cross_entropy.h`中定义，可参考`csrc/core/kernels/functions/cross_entropy.h`及其对应的`cpp`、`cu`文件

对应测试在`csrc/core/kernels/tests/test_functions_cpu.cpp`与`csrc/core/kernels/tests/test_functions_gpu.cu`中

测试样例使用`pytorch`生成，生成脚本保存在`csrc/core/functions/tests`文件夹中

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。

测试点共计126个，其中：

- CPU测试点61个
- GPU测试点65个

可进入根目录下的`scripts`文件夹，运行：

```
1 bash all_test.sh
```

若想分别测试`cpu`与`gpu`版本，可进入`scripts`文件夹，运行

```
1 bash test.sh --cpu
2 bash test.sh --gpu
```

若使用windows系统进行测试，按照下面步骤：

编译

```
1 mkdir build
2 cd build
3 cmake -DTEST=ON -DCUDA=OFF ..
4 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
```