自制大模型推理框架-第3次课程-算子类的设计

动手写大模型推理框架-算子类的设计

视频

算子类的设计.mkv

重录后：重录后-算子类的设计.mkv 声音更清晰

引言

在深度学习中，算子通常指的是在神经网络中对数据执行数学运算的函数。这些运算可以是简单的，如加法、乘法，也可以是复杂的，如卷积、池化、归一化等。

算子是构建神经网络模型的基础，它们定义了数据在网络中的流动方式以及如何通过这种流动进行特征提取和学习。根据算子内部参数的有无，我们大致可以将算子分为两大类：

1. 带参数的，例如卷积算子，全连接算子，rmsnorm算子等
2. 不带参数的，例如sigmoid算子，softmax算子等

另外我们还需要为量化算子预留设计空间（后续课程中会实现），但是下文中主要考虑的是带参（fp32）和不带参的两类算子。以后我们实现的所有算子类型都要继承于这两个基类型。

算子基类

我们先来看其中的类内变量，这些变量包括表示**算子类型**的 `layer_type` ，表示**算子处理数据类型**的 `data_type` ，以及表示算子所属设备类型的 `device_type` 。此外，我们还有相应的方法来返回这些类型，例如 `device_type()` 和 `layer_type()` 等分别获取算子类型和设备类型。

另外还有一些纯虚方法我们放到 `BaseLayer` 的派生类 `Layer` 中去实现，项目中算子的实现有fp32类型的，也有int8类型的，如果用设备类型来分，我们的算子(Layer)既有 `cpu` 上的实现，也有 `gpu` 上的实现。

```
1   class BaseLayer {
2    public:
3     explicit BaseLayer(base::DeviceType device_type, LayerType layer_type,
4                        base::DataType data_type, std::string layer_name = "");
5
6     base::DataType data_type() const;
7
8     LayerType layer_type() const;
9
10    ...
11    ...
12
13    const std::string& get_layer_name() const; // 返回层的名字
14
15    void set_layer_name(const std::string& layer_name); // 设置层的名称
16
17    base::DeviceType device_type() const; // 返回层的设备类型
18
19    void set_device_type(base::DeviceType device_type);
20
```

```
 protected:
  std::string layer_name_; // 层名
  LayerType layer_type_ = LayerType::kLayerUnknown; // 层类型
  base::DataType data_type_ = base::DataType::kDataTypeUnknown; // 层数据类型
  base::DeviceType device_type_ = base::DeviceType::kDeviceUnknown;
};
```

不带参（权重）算子类的设计

> 不带参的算子就是那些add，softmax等没有权重的算子

设置输入输出的部分

Y = x

我们来看看 `BaseLayer` 第一个不带参数的派生类 `Layer` ，我们来看看其中的类内变量 `inputs_` 和 `outputs` ，它们将于用于存放每个算子中的输入和输出张量。

```
class Layer : public BaseLayer {
 public:
  explicit Layer(base::DeviceType device_type, LayerType layer_type,
                 std::string layer_name = "");
  // set_input(0,x1)
  // set_input(1,x2)
  void set_input(int32_t idx, const tensor::Tensor& input) override; // 传入输入

  // set_output(0,y)
  void set_output(int32_t idx, const tensor::Tensor& output) override; // 传入输出

  const tensor::Tensor& get_input(int32_t idx) const override; // 获取输入

  const tensor::Tensor& get_output(int32_t idx) const override; // 获取输出

  size_t input_size() const override; // 获取输入的个数

  size_t output_size() const override; // 获取输出的个数

  void reset_input_size(size_t size);

  void reset_output_size(size_t size);

  virtual void to_cuda();

 private:
```

```
27    std::vector<tensor::Tensor> inputs_;  // 存放输入的数组
28    std::vector<tensor::Tensor> outputs_; // 存放输出的数组
29  };
```

比如我们需要对输入做 `add` 运算，那么我们就需要先通过 `set_input` 方法将输入放到 `inputs_` 变量中，放入的方法见 `set_input` 。比如有个add层，@0和@1两个输入数，set_input()

在 `set_input` 方法中我们需要指定这是该算子的第几个(`idx`)输入， `input` 是具体的输入张量，关于张量的概念，我们会在下一节课程中讲解。 **`set_input` 方法中我们需要先检查 `idx` 是否超出了该算子输入的个数限制**，比如 `ReLU` 算子就只能有一个输入，另外还要再检查该输入的数据类型(data type)是否和算子的设备类型(device type)是一致的。

同时也需要将用于存放的张量通过 `set_output` 方法放到 `outputs_` 变量中，方法的方法见 `set_output` 。

```
1  void Layer::set_input(int32_t idx, const tensor::Tensor& input) {
2    CHECK_GE(idx, 0);
3    CHECK_LT(idx, inputs_.size());
4    if (!input.is_empty()) {
5      CHECK(input.device_type() == device_type_);
6    }
7    this->inputs_.at(idx) = input;
8  }
```

`get_input` 则是 `set_input` 的反过程，从类中获取到第idx个输入张量用于做后续运算。另外还有些关于算子输入、输出张量的辅助函数，例如获取数量的 `input_size` 和 `output_size` 。

调用计算的部分

在用 `set_input` 和 `set_output` 设置好输入和输出张量之后，我们就要**调用最关键的计算方法**了，对于每个算子都需要去重写 `Layer::base_foward` ，例如 `ReLU` 的base_forward方法中需要重写完成以下的过程。

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

同理对于其他类型的算子，例如 `Softmax` 算子，也要重写 `Softmax::base_forward` 。我们举个例子来看一下， `Add` 算子的 `base_forward` 是怎么被重写的，可以看到 `base_forward` 是一个重载函数。

```
1  base::Status base_forward() override; // 每个算子的计算过程都有些不同，所以需要重写base_forward.
2
3  base::Status VecAddLayer::base_forward() {
4    auto status = this->check();
```

```
    if (!status) {
      return status;
    }
    auto input1 = this->get_input(0);
    auto input2 = this->get_input(1);
    auto output = this->get_output(0);
    kernel::get_add_kernel(device_type_)(input1, input2, output, nullptr);
    return base::error::Success();
  }
```

先是用 `get_input` 方法分别取出输入张量，随后再用 `get_output` 取出输出张量，随后再调用计算过程对两个输入中的数据进行一一加和。这里get_input获取到的输入张量是在调用base_forward之前，通过set_input传入并保存的，get_output同理。张量的定义会在下一节课当中讲。

我们再来看一个 `RoPE` 算子中计算函数的重写实现，也就是 `RoPELayer::base_forward()` ，同样是先从 `inputs_` 取出三个输入数据，分别是 `input_q` 和 `input_k` 以及 `input_pos` ，随后我们再调用它的计算过程得到最终的结果。在这里，请同学们先不要纠结计算过程是怎么写的，本节课的目的就是要先理解算子整体的设计，而不是一开始就把头扎到算子具体实现中。

值得说一句的是，以上代码(base_forward)中我们调用了 `check()` 方法，check方法同样是一个重载方法，每个算子派生类都要重写它用于检查调用过程中输入参数数量的合法性，我们先来看一下 `AddLayer` 中check方法的实现。

```
base::Status VecAddLayer::check() const {
  tensor::Tensor input1 = this->get_input(0);
  tensor::Tensor input2 = this->get_input(1);
  int32_t size = input1.size();
  base::Status status;
  status = check_tensor_with_dim(input1, device_type_, data_type_, size);
  if (!status) {
    LOG(ERROR) << "The input tensor 1 error in the add layer.";
    return status;
  }
  // 我的输入是cpu上的数据，而我的layer(device type)是gpu的
  status = check_tensor_with_dim(input2, device_type_, data_type_, size);
  if (!status) {
    LOG(ERROR) << "The input tensor 2 error in the add layer.";
    return status;
  }

  status = check_tensor_with_dim(get_output(0), device_type_, data_type_,
size);
  if (!status) {
    LOG(ERROR) << "The output tensor error in the add layer.";
    return status;
```

```
    }
    return base::error::Success();
  }
```

我们在forward方法中调用了check方法，check方法先是得到两个输入张量用于做检查，分别记作变量 `input1` 和变量 `input2` ，随后我们再检查它们两个的维度是否是一致的，最后就是检查数据类型是否正确，设备类型是否正确。

1. 我们知道算子类本身有一个数据类型为 `data_type_` ，我们这里要检查输入的数据类型是否为 `data_type_` ；
2. 算子类本身也有一个设备类型为 `device_type_` ，我们这里也要检查输入张量的设备类型是否也为 `device_type` 。

为什么要检查？因为我们的算子实现在不同设备上有不同的实现，CPU上有CPU上的算子实现，CUDA上有CUDA的算子实现，所以我们也要确保传入的输入和算子的计算过程是符合的，比如我们如果用CUDA写了一个算子计算过程，那么传入的输入上的数据必定要存储在显存之上。

张量类我们将在下一讲中涉及到。

带参数的算子类设计

带参数的算子类，多了一个类内变量用于存储权重张量：

```

class LayerFp32Param : public Layer {
 public:
  explicit LayerFp32Param(base::DeviceType device_type, LayerType layer_type,
                          std::string layer_name = "");

  size_t weight_size() const;

  void reset_weight_size(size_t size);

  tensor::Tensor& get_weight(int32_t idx);

  const tensor::Tensor& get_weight(int32_t idx) const;

  void set_weight(int32_t idx, const tensor::Tensor& weight);

  void set_weight(int32_t idx, const std::vector<int32_t>& dims, const float*
weight_ptr,
                  base::DeviceType device_type =
base::DeviceType::kDeviceUnknown);

 private:
```

```
21    std::vector<tensor::Tensor> weights_; // 用于额外存放权重数据
22  };
```

其中 `weights_` 变量用于存放权重张量，例如 `matmul` 和 `rmsnorm` 算子中的权重。我们来看看这两个算子是怎么取得权重张量的，我们来看看 `rmsnorm` 算子的计算过程 `base_forward` 实现。

```
1   base::Status RmsNormLayer::base_forward() { // 计算的时候
2     auto status = check();
3     if (!status) {
4       return status;
5     }
6     auto input = this->get_input(0);
7     auto weight = this->get_weight(0);
8     auto output = this->get_output(0);
9     // 得到一个具体的算子计算实现
10    kernel::get_rmsnorm_kernel(device_type_)(input, weight, output,
11                                             cuda_config_ ? cuda_config_-
    >stream : nullptr);
12    return base::error::Success();
13  }
```

我们通过 `get_weight` 方法来取得对应的权重张量，再做相应的运算，同样是先前通过set_weight方法完成对某个算子的权重赋值。 `kernel::get_rmsnorm_kernel(device_type_)` 方法用于根据设备类型获取不同的算子实现。和权重有关的还有其他一些辅助方法，类似获取算子中权重张量的个数 `weight_size` ，重置权重的个数 `reset_weight_size` 等等。

获取不同设备的算子实现

```
1   base::Status VecAddLayer::base_forward() {
2     auto status = this->check();
3     if (!status) {
4       return status;
5     }
6     auto input1 = this->get_input(0);
7     auto input2 = this->get_input(1);
8     auto output = this->get_output(0);
9     kernel::get_add_kernel(device_type_)(input1, input2, output);
10    return base::error::Success();
11  }
12
13  AddKernel get_add_kernel(base::DeviceType device_type) {
14    if (device_type == base::DeviceType::kDeviceCPU) {
```

```
    return add_kernel_cpu; // 返回一个具体的函数指针
  } else if (device_type == base::DeviceType::kDeviceCUDA) {
    return add_kernel_cu;
  } else {
    LOG(FATAL) << "Unknown device type for get a add kernel.";
    return nullptr;
  }
}
```

如果当前的设备类型是 `cpu` ，那么就返回 `add_kernel_cpu` 方法用于运算；反之如果设备是 `cuda` ，那么我们就返回 `add_kernel_cu` 用于在N卡上进行运算。

```
void add_kernel_cpu(const tensor::Tensor& input1, const tensor::Tensor&
input2,
                    const tensor::Tensor& output) {
  arma::fvec input_vec1(const_cast<float*>(input1.ptr<float>()),
input1.size(), false,
                        true);
  arma::fvec input_vec2(const_cast<float*>(input2.ptr<float>()),
input2.size(), false,
                        true);
  arma::fvec output_vec(const_cast<float*>(output.ptr<float>()),
output.size(), false,
                        true);
  output_vec = input_vec1 + input_vec2;
}
```

来看看 `add_kernel_cpu` 中的计算过程实现，该方法接收两个输入并将它们加和放到结果output中。所以链路就是：

1. base_forward调用
2. 在base_forward方法中get input and weight以及输出
3. select kernel，根据设备类型选择算子实现

kernel::get_add_kernel(device_type_)(input1, input2, output); **选中kernel实现后，我们将向kernel传入必要的输入、输出和权重张量**，这里我们根据设备类型是cpu，选中了add_kernel_cpu实现。

4. 向函数指针传入数个input和weight并完成，并将结果放到output中。只简单介绍一下这里的计算流程，因为不是本节课的重点：
   a. 用第一个输入张量赋值input_vec1
   b. 用第二个输入张量赋值input_vec2

c. 随后我们调用Armadillo数学库的+加法计算得到最终的输出值，并保存在output_tensor中。

input_vec1和input_vec都是数学库Armdillo中的向量表示

跟我练

我们来看一下加法算子的整体流程，两个张量是如何被相加并放到输出张量中的。

```
1   TEST(test_add_cu, add1_nostream) {
2     auto alloc_cu = base::CUDADeviceAllocatorFactory::get_instance();
3
4     int32_t size = 32 * 151;
5
6     tensor::Tensor t1(base::DataType::kDataTypeFp32, size, true, alloc_cu);
7     tensor::Tensor t2(base::DataType::kDataTypeFp32, size, true, alloc_cu);
8     tensor::Tensor out(base::DataType::kDataTypeFp32, size, true, alloc_cu);
9
10    set_value_cu(static_cast<float*>(t1.get_buffer()->ptr()), size, 2.f);
11    set_value_cu(static_cast<float*>(t2.get_buffer()->ptr()), size, 3.f);
12
13    kernel::get_add_kernel(base::DeviceType::kDeviceCUDA)(t1, t2, out, nullptr);
14    cudaDeviceSynchronize();
15    float* output = new float[size];
16    cudaMemcpy(output, out.ptr<float>(), size * sizeof(float),
    cudaMemcpyDeviceToHost);
17    for (int i = 0; i < size; ++i) {
18      ASSERT_EQ(output[i], 5.f);
19    }
20
21    delete[] output;
22  }
```

我们先是准备了两个张量，分别是t1和t2，再将两者赋值每个位置的元素均赋值为值2和值3，再调用`get_add_kernel`选择cuda算子对两个输入进行计算，并将结果和“5.f”进行比较。