第三课-计算图的定义

计算图的概念

`KuiperInfer`使用的模型格式是`PNNX`. `PNNX`是`PyTorch Neural Network Exchange`的缩写，它的愿景是将`PyTorch`模型文件直接导出为高效、简洁的计算图。计算图的概念如第一章说的那样，一般包括了以下的几个部分：

1. `Operator`:深度学习计算图中的计算节点。
2. `Graph`: 有多个`Operator`串联得到的有向无环图，规定了各个计算节点（`Operator`）执行的流程和顺序。
3. `Layer`：**计算节点中**运算的具体执行者，`Layer`类先读取输入张量中的数据，然后对输入张量进行计算，得到的结果存放到计算节点的输出张量中，**当然，不同的算子中`Layer`的计算过程会不一致**。
4. `Tensor`: 用于存放**多维数据**的数据结构，方便数据在计算节点之间传递，同时该结构也封装矩阵乘、点积等与矩阵相关的基本操作。

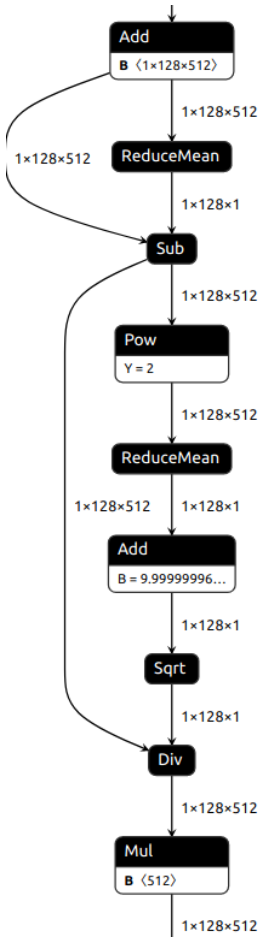
如果你对以上的内容已经没有印象了，可以自行回顾本课程的第一章。作为一种计算图形式，`PNNX`自然也不例外。接下来，我们来探讨一下为什么在出现了`ONNX`计算图之后还需要`PNNX`，以及它解决了什么问题？

PNNX计算图的优势

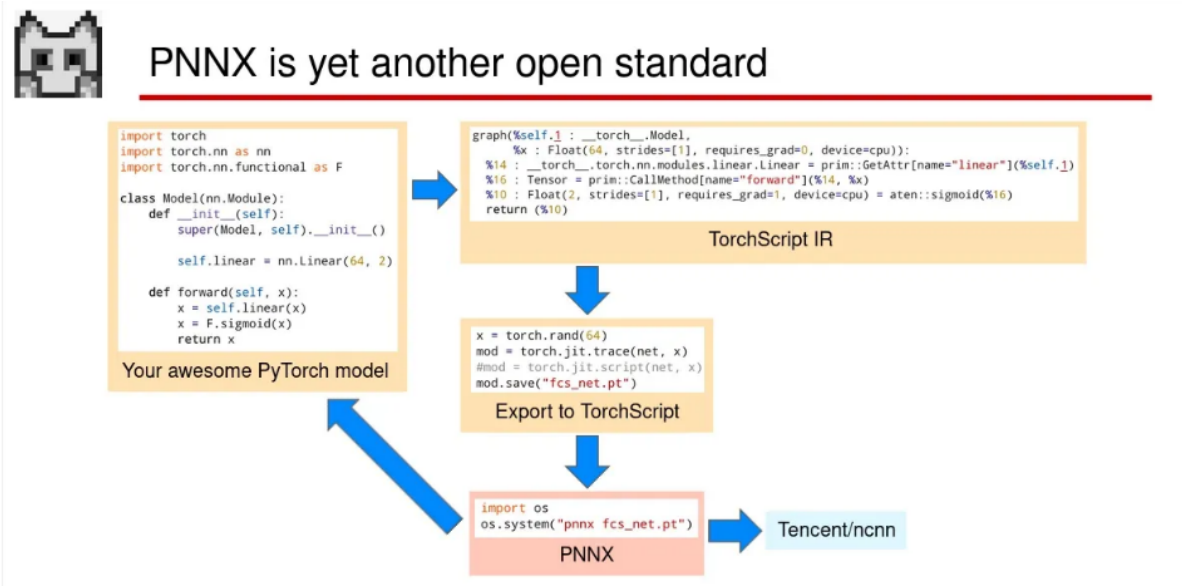
> 参考资料：https://zhuanlan.zhihu.com/p/427620428

以往我们将训练好的模型导出为`ONNX`结构之后，模型中的一个复杂算子不仅经常会被拆分成多个细碎的算子，而且为了将这些细碎的算子拼接起来完成原有算子的功能，通常还需要一些称之为“胶水算子”的辅助算子，例如`Gather`和`Unsqueeze`等。

你还认得出下图的算子，其实是被拆分后`LayerNorm`算子吗？当然`ONNX`使用多个小算子去等价一个复杂算子的设计，也是为了用尽可能少的算子去兼容更多的训练框架。



但是过于细碎的计算图会不仅不利于推理的优化。另外，拆分的层次过于细致，也会导致算法工程师难以将导出的模型和原始模型进行结构上的相互对应。为了解决以上说到的问题，我们选用`NCNN`那推理框架的计算图格式之一`PNNX`, 那么`PNNX`给我们带来了什么呢？下图是它在模型部署中的位置。



1. 使用模板匹配（`pattern matching`）的方法将匹配到的子图用对应等价的大算子替换掉，例如可以将上图子图中的多个小算子（已经在`TorchScript`中被拆分的）重新替换为`LayerNorm`算子。或者在对`PyTorch`模型导出时，也可以自定义某个`nn.Module`不被拆分；
2. 在`PyTorch`中编写的简单算术表达式在转换为`PNNX`后，会保留表达式的整体结构，而不会被拆分成许多小的加减乘除算子。例如表达式`add(mul(@0,@1),add(@2,@3))`不会被拆分为两个`add`算子和一个`mul`算子，而是会生成一个表达式算子`Expression`；
3. `PNNX`项目中有大量图优化的技术，包括了算子融合，常量折叠和消除，公共表达式消除等技术。
    - 算子融合优化是一种针对深度学习神经网络的优化策略，通过将多个相邻的计算算子合并为一个算子来减少计算量和内存占用。以卷积层和批归一化层为例，我们可以把两个算子合并为一个新的算子，也就是将卷积的公式带入到批归一化层的计算公式中：

$$Conv = w * x_1 + b$$

$$BN = \gamma \frac{x_2 - \hat{u}}{\sigma^2 + \epsilon} + \beta$$

其中x_1和x_2依次是卷积和批归一化层的输入，w是卷积层的权重，b是卷积层的偏移量，$\hat{u}$和σ依次是样本的均值和方差，ϵ为一个极小值。带入后有：

$$Fused = \gamma \frac{(w * x + b) - \hat{u}}{\sigma^2 + \epsilon} + \beta$$

- 常量折叠是将在编译时期间将**表达式中的常量计算出来，然后将结果替换为一个等价的常量**，以减少模型在运行时的计算量。
- 常量移除就是将计算图中不需要的常数（**计算图推理的过程中未使用**）节点删除，从而减少计算图的文件和加载后的资源占用大小。
- 公共表达式消除优化是一种针对计算图中重复计算的优化策略，**它可以通过寻找并合并重复计算的计算节点，减少模型的计算量和内存占用。**

  公共子表达式检测是指**查找计算图中相同的子表达式**，公共子表达式消除是指**将这些重复计算的计算节点合并为一个新的计算节点**，从而减少计算和内存开销。举个例子：

  ```
  X = input(3,224,224)
  A = Conv(X)
  B = Conv(X)
  C = A + B
  ```

  在上方的代码中，`Conv(X)`这个结果被计算了两次，公共子表达式消除可以将它优化为如下代码，这样一来就少了一次卷积的计算过程。

  ```
  X = input(3, 224,224)
  T = Conv(X)
  C = T + T
  ```

综上所述，如果在我们推理框架的底层用`PNNX`计算图，就可以吸收图优化和算子融合的结果，使得推理速度更快更高效。

PNNX计算图的格式

`PNNX`由图结构(Graph), 运算符(Operator)和操作数(Operand)这三种结构组成的，设计非常简洁。

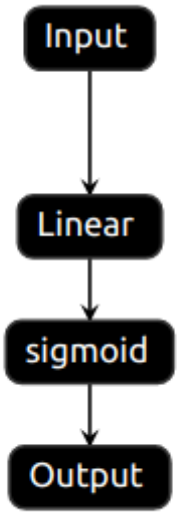
用于测试的PyTorch模型

为了帮助同学们更好地掌握**封装后的计算图格式**，我们准备了一对较小的模型文件`linear.param`和`linear.bin`来做单步调试，它们分别是网络的**结构定义**和**权重文件**。该模型在`PyTorch`中的定义如下：

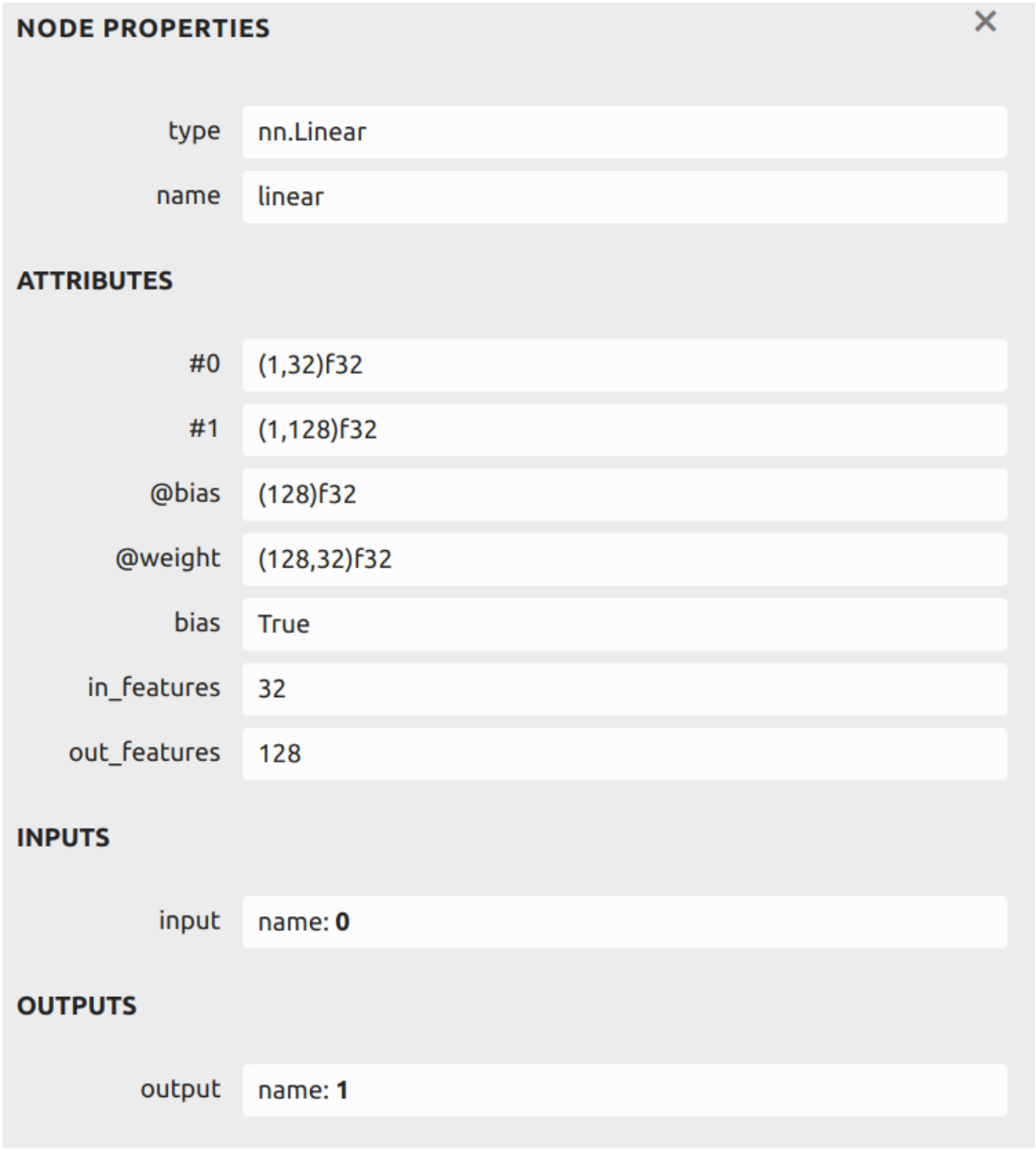
```
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear = nn.Linear(32, 128)

    def forward(self, x):
        x = self.linear(x)
        x = F.sigmoid(x)
        return x
```

这是一个非常简单的模型，其作用是对输入`x`进行线性映射（从32维到128维），并对输出进行`sigmoid`计算，从而得到最终的计算结果。该模型的网络结构如下图所示（使用`Netron`软件打开），以`Linear`层为例：



- `Linear`层有`#0`和`#1`两个操作数，分别为输入和输出张量。
- `Linear`层有`@weight`和`@bias`两个属性参数，用来存储该层的权重数据（即`weight`和`bias`）。
- `Linear`层有`bias`、`in_features`和`out_features`这三个属性信息。



PNNX中的图结构(Graph)

```
class Graph
{
    Operator* new_operator(const std::string& type, const
std::string& name);
    Operator* new_operator_before(const std::string& type,
const std::string& name, const Operator* cur);

    Operand* new_operand(const torch::jit::Value* v);
    Operand* new_operand(const std::string& name);
    Operand* get_operand(const std::string& name);

    std::vector<Operator*> ops;
    std::vector<Operand*> operands;
};
```

`Gragh`的核心作用是管理计算图中运算符和操作数，我们在以下对运算符和操作数这两个概念进行说明：

1. `Operator`类用来**表示计算图中的运算符（算子）**，比如一个模型中的`Convolution`，`Pooling`等算子；
2. `Operand`类用来表示计算图中的**操作数**，即**与一个运算符有关的输入和输出张量**；
3. `Graph`类的成员函数提供了方便的接口用来**创建和访问操作符和操作数**，以构建和遍历计算图。同时，它也是模型中**运算符（算子）和操作数的集合**。

PNNX中图结构相关的单元测试

```
TEST(test_ir, pnnx_graph_ops) {
  using namespace kuiper_infer;
  /**
   * 如果这里加载失败，请首先考虑相对路径的正确性问题
   */
  std::string
bin_path("course3/model_file/test_linear.pnnx.bin");
  std::string
param_path("course3/model_file/test_linear.pnnx.param");
  std::unique_ptr<pnnx::Graph> graph =
std::make_unique<pnnx::Graph>();
```

```
  int load_result = graph->load(param_path, bin_path);
  // 如果这里加载失败，请首先考虑相对路径(bin_path和param_path)的
正确性问题
  ASSERT_EQ(load_result, 0);
  const auto &ops = graph->ops;
  for (int i = 0; i < ops.size(); ++i) {
    LOG(INFO) << ops.at(i)->name;
  }
}
```

以上代码使用`pnnx::Graph`类的`load`函数传递相应参数来加载模型，并使用单元测试来检查模型是否被成功加载。

随后我们在`for`循环中打印相关的运算符，得到了与模型结构图上一致的输出结果。可以看到，这里输出的运算符和模型结构图上的运算符是一致的。

```
I20230606 13:52:36.940599  3651 test_ir.cpp:23]
pnnx_input_0
I20230606 13:52:36.940616  3651 test_ir.cpp:23] linear
I20230606 13:52:36.940624  3651 test_ir.cpp:23]
F.sigmoid_0
I20230606 13:52:36.940632  3651 test_ir.cpp:23]
pnnx_output_0
```

在以下的代码中，我们**除了输出运算符，还输出了运算符相关的输入输出张量，包括张量相关的名字(name), 形状(shape)等属性**，可以看到输出的结果也和可视化模型结构图保持了一致。

```
TEST(test_ir, pnnx_graph_operands) {
    ...
    ...
    for (int j = 0; j < op->inputs.size(); ++j) {
      LOG(INFO) << "Input name: " << op->inputs.at(j)-
>name
                << " shape: " << ShapeStr(op-
>inputs.at(j)->shape);
    }

    LOG(INFO) << "OP Output";
```

```
    for (int j = 0; j < op->outputs.size(); ++j) {
      LOG(INFO) << "Output name: " << op->outputs.at(j)-
>name
                << " shape: " << ShapeStr(op-
>outputs.at(j)->shape);
    ...
    ...
}
```

输出结果：

```
...
...
I20230606 14:05:59.942880  3897 test_ir.cpp:52] OP Name:
linear
I20230606 14:05:59.942904  3897 test_ir.cpp:53] OP Inputs
I20230606 14:05:59.942926  3897 test_ir.cpp:55] Input
name: 0 shape: 1 x 32
I20230606 14:05:59.942950  3897 test_ir.cpp:59] OP Output
I20230606 14:05:59.942974  3897 test_ir.cpp:61] Output
name: 1 shape: 1 x 128
...
...
```

PNNX中的运算符结构(Operator)

有了上面的直观认识，我们来聊聊`PNNX`中的运算符结构。

```
class Operator
{
public:
    std::vector<Operand*> inputs;
    std::vector<Operand*> outputs;

    std::string type;
    std::string name;

    std::vector<std::string> inputnames;
    std::map<std::string, Parameter> params;
    std::map<std::string, Attribute> attrs;
};
```

在PNNX中，`Operator`用来表示一个算子，它由以下几个部分组成：

1. `inputs`：类型为`std::vector<operand>`，表示这个算子在计算过程中所需要的输入操作数`operand`；
2. `outputs`：类型为`std::vector<operand>`，表示这个算子在计算过程中得到的输出操作数`operand`；
3. `type`和`name`类型均为`std::string`，分别表示该运算符号的类型和名称；
4. `params`，类型为`std::map`，用于存放该运算符的所有参数（例如卷积运算符中的`params`中将存放`stride`，`padding`，`kernel size`等信息）；
5. `attrs`，类型为`std::map`，用于存放运算符号所需要的具体权重属性（例如卷积运算符中的`attrs`中就存放着卷积的权重和偏移量，通常是一个`float32`数组）。

PNNX中运算符结构相关的单元测试

```
TEST(test_ir, pnnx_graph_operands_and_params) {
    ...
    ...

    LOG(INFO) << "Params";
    for (const auto &attr : op->params) {
      LOG(INFO) << attr.first;
```

```
    }

    LOG(INFO) << "Weight: ";
    for (const auto &weight : op->attrs) {
      LOG(INFO) << weight.first << " : " <<
ShapeStr(weight.second.shape);
    }
    LOG(INFO) << "----------------------------------------
-----";
  }
}
```

以上代码使用`pnnx::Graph`类的`load`函数传递相应参数来加载模型，并使用单元测试来检查模型是否被成功加载。

随后我们在`for`循环中打印`Linear`运算符，**得到了与可视化模型结构图上一致的权重(`Weight`)信息和参数信息(`bias`, `in_features`, `out_features`).**

```
...
...
I20230606 14:26:40.971652  4186 test_ir.cpp:86] OP Name:
linear
I20230606 14:26:40.971665  4186 test_ir.cpp:87] OP Inputs
I20230606 14:26:40.971673  4186 test_ir.cpp:89] Input
name: 0 shape: 1 x 32
I20230606 14:26:40.971681  4186 test_ir.cpp:93] OP Output
I20230606 14:26:40.971688  4186 test_ir.cpp:95] Output
name: 1 shape: 1 x 128

I20230606 14:26:40.971696  4186 test_ir.cpp:99] Params
I20230606 14:26:40.971704  4186 test_ir.cpp:101] bias
I20230606 14:26:40.971710  4186 test_ir.cpp:101]
in_features
I20230606 14:26:40.971717  4186 test_ir.cpp:101]
out_features

I20230606 14:26:40.971724  4186 test_ir.cpp:104] Weight:
```

```
I20230606 14:26:40.971731  4186 test_ir.cpp:106] bias :
128
I20230606 14:26:40.971738  4186 test_ir.cpp:106] weight :
128 x 32
...
...
```

KuiperInfer对计算图的封装