【我与TVM二三事 前篇（1）】relay parse

此文会跟读torch前端到tvm.IRModule构建完成的流程，梳理相关知识点。

Tvm支持大量前端格式，选择torch是因为对算法人员而言上手容易，并且torchvision提供傻瓜式的加载模型的接口，可以集中精力到追读tvm解析的流程上。

跟读使用以下代码：

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_parse/demo.py>

起作用的代码就一句，relay.frontend.from\_pytorch，tvm的tutorial中对每个训练框架都会有一个from\_XXX的函数，功能就是把计算图中的信息保存成relay格式的信息。Relay格式信息的载体是tvm.IRModule，整个过程基本围绕这个结构展开。

调用流程：

relay.frontend.from\_pytorch

1 tvm.IRModule(初始化容器，用于保存relay的信息)

2 Prelude(加载辅助函数)

2.1 import\_from\_std(加载基础函数)

2.2 tensor\_array\_ops.register(加载tensorarray相关函数)

3 PyTorchOpConverter(构建converter，用于算子解析)

4 create inputs && params

4.1 \_get\_relay\_input\_vars(构建inputs)

4.2 convert\_params(构建params)

5 converter.convert\_operators(转换算子)

6 set the IRModule

6.1 analysis.free\_vars(确定无依赖参数，例如inputs，params)

6.2 tvm.relay.Function(用Function包装DAG计算过程)

6.3 transform.RemoveUnusedFunctions(简单优化去除无用代码)

@ tvm/python/tvm/relay/frontend/pytorch.py

def from\_pytorch(script\_module, input\_infos, custom\_convert\_map=None, default\_dtype="float32"):

mod = tvm.IRModule()

prelude = Prelude(mod)

converter = PyTorchOpConverter(prelude, default\_dtype)

…

outputs = \_get\_relay\_input\_vars(

graph, input\_infos, prelude, default\_dtype=default\_dtype, is\_module=is\_module

)

param\_vars, tensors, packed\_param\_map = convert\_params(graph, params)

…

ret = converter.convert\_operators(\_get\_operator\_nodes(graph.nodes()), outputs, ret\_name)[0]

…

for arg in \_analysis.free\_vars(ret):

if arg.name\_hint not in tvm\_params.keys():

data\_inputs.append(arg)

else:

func\_args.append(arg)

func\_args = data\_inputs + func\_args

mod["main"] = tvm.relay.Function(func\_args, ret)

return transform.RemoveUnusedFunctions()(mod), tvm\_params

# Step 1 创建IRModule

简单理解IRModule就是起始与终结之羁绊，其实就是类似全局的map的东西，作为各阶段信息的载体。通过python和C++的接口最终调用的是IRModule的构造函数。

@ tvm/src/ir/module.cc

IRModule::IRModule(tvm::Map<GlobalVar, BaseFunc> functions,

tvm::Map<GlobalTypeVar, TypeData> type\_definitions,

std::unordered\_set<String> import\_set, parser::SourceMap source\_map) {

auto n = make\_object<IRModuleNode>();

n->functions = std::move(functions);

n->type\_definitions = std::move(type\_definitions);

…

data\_ = std::move(n);

}

第一步初始化只是进行属性设定，没有添加functions和type\_definitions，所以打印一下mod会发现是空。这种初始化方式是TVM常用的对Object进行初始化的方法，TVM中常用的Object，ObjectRef和ObjectPtr可以理解为shared\_ptr的一种实现，详细说明可以参考[深入理解TVM：Object家族（二）](https://zhuanlan.zhihu.com/p/432851987)，写的很详细易懂。

# Step 2 加载辅助函数

@ tvm/python/tvm/relay/frontend/pytorch.py

def from\_pytorch(script\_module, input\_infos, custom\_convert\_map=None, default\_dtype="float32"):

…

prelude = Prelude(mod)

## Step 2.1 加载基础函数

@ tvm/python/tvm/relay/prelude.py

class Prelude:

def \_\_init\_\_(self, mod=None):

if mod is None:

mod = IRModule()

self.mod = mod

self.load\_prelude()

…

def load\_prelude(self):

…

self.mod.import\_from\_std("prelude.rly")

-> @ tvm/src/ir/module.cc（path="tvm/python/tvm/relay/std/prelude.rly"）

void IRModuleNode::Import(const String& path) {

if (this->import\_set\_.count(path) == 0) {

…

std::string file\_contents{std::istreambuf\_iterator<char>(src\_file),

std::istreambuf\_iterator<char>()};

auto mod\_to\_import = parser::ParseModule(path, file\_contents, GetRef<IRModule>(this));

Update(mod\_to\_import);

}

}

这一段大概理解就是填充了mod中的一些基础函数，加载之后会调用InferType，定义在tvm/src/relay/transforms/type\_infer.cc 中，对加载进mod的variables进行类型检测。加载使用parser系统从文件加载模块，这个过程没有仔细研究，留坑。

TVM中IRModule中保存的函数以def @作为前缀，结构是Function/FunctionNode，其他的是一些辅助的type，struct之类的结构定义。基础函数以及结构形式如下：

type List[A] {

Cons(A, List[A]),

Nil,

}

…

def @size[A](%t: Tree[A]) -> int32 {

match (%t) {

Rose(\_, %sub\_trees: List[Tree[A]]) => {

%7 = @map(@size, %sub\_trees) /\* …/prelude.rly \*/ /\* ty=List[int32] \*/;

%8 = @sum(%7) /\* …/prelude.rly \*/ /\* ty=int32 \*/;

add(1 /\* ty=int32 \*/, %8) /\* …/prelude.rly \*/ /\* ty=int32 \*/

},

}

}

…

使用案例：@ tvm/python/tvm/relay/frontend/pytorch.py

def convert\_to\_list\_adt(self, py\_lst):

…

\_, cons, nil = self.prelude.mod.get\_type("List")

adt\_lst = nil()

for elem in reversed(py\_lst):

adt\_lst = cons(elem, adt\_lst)

return adt\_lst

完成List of torch Tensor到List of tvm Tensor的转换

## Step2.2 添加tensor\_array相关函数

@ tvm/python/tvm/relay/prelude.py

class Prelude:

def load\_prelude(self):

…

for dtype in [

"float32",

…

]:

tensor\_array\_ops = TensorArrayOps(self, dtype)

tensor\_array\_ops.register()

添加tensor处理的函数，形式如：

type tensor\_float32\_t {

tensor\_nil\_float32,

tensor0\_float32(float32),

tensor1\_float32(Tensor[(?), float32]),

tensor2\_float32(Tensor[(?, ?), float32]),

tensor3\_float32(Tensor[(?, ?, ?), float32]),

tensor4\_float32(Tensor[(?, ?, ?, ?), float32]),

tensor5\_float32(Tensor[(?, ?, ?, ?, ?), float32]),

tensor6\_float32(Tensor[(?, ?, ?, ?, ?, ?), float32]),

}

def @tensor\_array\_unstack\_tensor4\_helper\_float32(%i: int32, %up: int32, %t: Tensor[(?, ?, ?, ?), float32]) -> List[tensor\_float32\_t[]] {

%0 = equal(%i, %up);

if (%0) {

Nil

} else {

%1 = take(%t, %i, axis=0);

%2 = add(%i, 1);

%3 = tensor3\_float32(%1);

%4 = @tensor\_array\_unstack\_tensor4\_helper\_float32(%2, %up, %t);

Cons(%3, %4)

}

}

…

定义了一个tensor可以接受的尺寸以及处理函数。这里可以看到每种tensor按照维度从0到6进行了注册，也就是说TVM的Tensor维度最多为6。解析过程中高于此维度的tensor会出现无法解析的报错。使用案例：

@ tvm/python/tvm/relay/frontend/pytorch.py

def map\_tensor\_array\_constructor(self, adt\_lst, shape):

static\_tensor\_array\_ops = StaticTensorArrayOps(self.prelude, "float32", shape)

static\_tensor\_array\_ops.register()

tensor\_create = self.prelude.get\_tensor\_ctor\_static("tensor\_constructor", "float32", shape)

return self.prelude.map(tensor\_create, adt\_lst)

完成tensor array之间的转换

# Step3 创建converter

@ tvm/python/tvm/relay/frontend/pytorch.py

def from\_pytorch(script\_module, input\_infos, custom\_convert\_map=None, default\_dtype="float32"):

…

converter = PyTorchOpConverter(prelude, default\_dtype)

Converter是原始模型转换成Relay IR的核心，其中最重要的部分就是convert\_map，里面记录了不同的pytorch算子转换到Relay IR的核心逻辑，以dict的形式记录，key是torch算子的名称，类似aten::XXX格式。

构建converter的过程中可以用custom\_convert\_map参数传入自定义的处理过程。

# Step4 构建inputs和params

正式开始转换op工作，首先需要处理的是一些入口信息。一般编译的流程都是从inputs信息开始以DFS/BFS方式进行节点的添加，统计记录拓扑结构，如果之前把玩过其他类型编译器的，对此过程应该不陌生。

## Step 4.1 创建inputs

@ tvm/python/tvm/relay/frontend/pytorch.py

def from\_pytorch(script\_module, input\_infos, custom\_convert\_map=None, default\_dtype="float32"):

…

outputs = \_get\_relay\_input\_vars(

graph, input\_infos, prelude, default\_dtype=default\_dtype, is\_module=is\_module

)

对应此demo，即将torch格式的输入以及input的信息转换成tvm格式的输出：

torch graph inputs:

[self.1 defined in (%self.1 : \_\_torch\_\_.torchvision.models.resnet.ResNet, %input.1 : Float(1, 3, 224, 224, strides=[150528, 50176, 224, 1], requires\_grad=0, device=cpu) = prim::Param()

), input.1 defined in (%self.1 : \_\_torch\_\_.torchvision.models.resnet.ResNet, %input.1 : Float(1, 3, 224, 224, strides=[150528, 50176, 224, 1], requires\_grad=0, device=cpu) = prim::Param()

)]

Input\_infos:

[('input0', ((1, 3, 224, 224), 'float32'))]

TVM inputs:

{'input.1': Var(input0, ty=TensorType([1, 3, 224, 224], float32))}

每个input被用Var结构记录，其中包含name\_hint，以及TensorType类型，TensorType用于记录shape和dtype，属于tensor常规属性。

## Step 4.2 创建params

@ tvm/python/tvm/relay/frontend/pytorch.py

def from\_pytorch(script\_module, input\_infos, custom\_convert\_map=None, default\_dtype="float32"):

…

param\_vars, tensors, packed\_param\_map = convert\_params(graph, params)

param\_vars被记录为dict，key为pytorch计算图格式中存储参数的id，value为Var原语类型。tvm\_params则用于存储数据，其中数据类型tvm.nd.array很类似mxnet中的ndarray数据结构（个人感觉可能是因为陈天奇参与了mxnet的开发，mxnet中很多地方可以看到tvm的影子），底层使用了DLTensor进行存储，创建过程如下

tvm/src/runtime/ndarray.cc:

TVM\_REGISTER\_GLOBAL("runtime.TVMArrayAllocWithScope").set\_body([](TVMArgs args, TVMRetValue\* ret) {

int64\_t\* shape\_ptr = static\_cast<int64\_t\*>(static\_cast<void\*>(args[0]));

int ndim = args[1];

std::vector<int64\_t> shape(shape\_ptr, shape\_ptr + ndim);

DataType dtype = args[2];

Device dev = args[3];

Optional<String> mem\_scope = args[4];

auto ndarray = NDArray::Empty(shape, dtype, dev, mem\_scope);

\*ret = ndarray;

});

NDArray NDArray::Empty(std::vector<int64\_t> shape, DLDataType dtype, Device dev,

Optional<String> mem\_scope) {

NDArray ret = Internal::Create(shape, dtype, dev);

ret.get\_mutable()->dl\_tensor.data =

DeviceAPI::Get(ret->device)

->AllocDataSpace(ret->device, shape.size(), shape.data(), ret->dtype, mem\_scope);

return ret;

}

其中核心部分为ret.get\_mutable()->dl\_tensor.data=…这一句进行内存分配，那么dl\_tensor又是什么？查到定义

@ tvm/include/tvm/runtime/ndarray.h

class NDArray::ContainerBase {

public:

…

DLTensor dl\_tensor;

…

}

DLTensor可以参照[通过 DLPack 构建跨框架深度学习编译器](https://blog.csdn.net/yiran103/article/details/81709356)进行一些理解。

@ tvm/3rdparty/dlpack/include/dlpack/dlpack.h

typedef struct {

void\* data;

DLDevice device;

int ndim;

DLDataType dtype;

int64\_t\* shape;

int64\_t\* strides;s

uint64\_t byte\_offset;

} DLTensor;

核心就是使用tvm.nd.array进行数据存储，兼容各种格式的ndarray数据。可以打印ndarray的内容，格式和numpy.ndarray十分相似。

# Step5 转换ops

@ tvm/python/tvm/relay/frontend/pytorch.py

def from\_pytorch(script\_module, input\_infos, custom\_convert\_map=None, default\_dtype="float32"):

…

ret = converter.convert\_operators(\_get\_operator\_nodes(graph.nodes()), outputs, ret\_name)[0]

对每个pytorch的node按照kind() 进行查找，找到对应的解析方法，大部分用于计算的函数都定义在PyTorchOpConverter中，例如使用PyTorchOpConverter.convolution()函数解析节点得到：

torch node:

%input.2 : Float(1, 64, 112, 112, strides=[802816, 12544, 112, 1], requires\_grad=1, device=cpu) = aten::\_convolution(%input.1, %25, %26, %29, %32, %35, %36, %39, %40, %41, %42, %43, %44), scope: \_\_module.conv1 # …/torch/nn/modules/conv.py:396:0

tvm node (<class 'tvm.relay.expr.Call'>):

:free\_var %input0: Tensor[(1, 3, 224, 224), float32];

free\_var %conv1.weight: Tensor[(64, 3, 7, 7), float32];

nn.conv2d(%input0, %conv1.weight, strides=[2, 2], padding=[3, 3, 3, 3], channels=64, kernel\_size=[7, 7])

Call原语是最常见的TVM节点类型，一般计算图中计算节点都属于此类行，追踪Call的构建：

@ tvm/python/tvm/relay/frontend/pytorch.py

def convolution(self, inputs, input\_types):

…

conv\_op = \_op.nn.conv2d

…

conv\_out = conv\_op(

data,

weight,

\*\*kwargs

)

可以看到核心的构造调用了\_op.nn.conv2d，此处会再根据python调用C++最终定位到:

@ tvm/src/relay/op/nn/convolution.cc

TVM\_REGISTER\_GLOBAL("relay.op.nn.\_make.conv2d")

.set\_body\_typed([](Expr data, Expr weight, Array<IndexExpr> strides, Array<IndexExpr> padding,

Array<IndexExpr> dilation, int groups, IndexExpr channels,

Array<IndexExpr> kernel\_size, String data\_layout, String kernel\_layout,

String out\_layout, DataType out\_dtype) {

return MakeConv<Conv2DAttrs>(data, weight, strides, padding, dilation, groups, channels,

kernel\_size, data\_layout, kernel\_layout, out\_layout, out\_dtype,

"nn.conv2d");

});

…

template <typename T>

inline Expr MakeConv(Expr data, Expr weight, Array<IndexExpr> strides, Array<IndexExpr> padding,

Array<IndexExpr> dilation, int groups, IndexExpr channels,

Array<IndexExpr> kernel\_size, std::string data\_layout,

std::string kernel\_layout, std::string out\_layout, DataType out\_dtype,

std::string op\_name) {

auto attrs = make\_object<T>();

attrs->strides = std::move(strides);

…

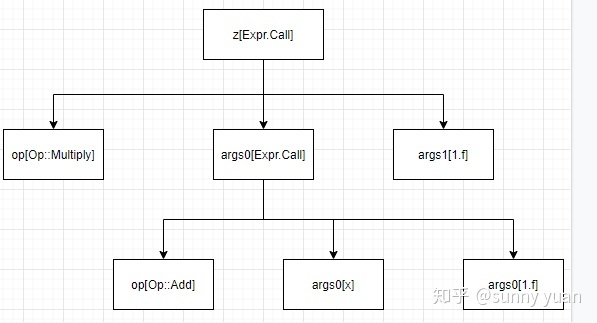
const Op& op = Op::Get(op\_name);

return Call(op, {data, weight}, Attrs(attrs), {});

}

Conv->Call 的创建和Conv op的注册需要一起使用。注册机制对于训练框架和模型编译器而言比较常见，在这里有一片详细介绍：[深入理解TVM：RELAY\_REGISTER\_OP](https://zhuanlan.zhihu.com/p/369433448)

Call结构的理解则可以参考[TVM之设计模式解读（一）--visitor模式](https://zhuanlan.zhihu.com/p/341334406)，这里借用@sunny yuan一张图来形象的表达一下。



conv2d op的注册代码：

@ tvm/src/relay/op/nn/convolution.cc

RELAY\_REGISTER\_OP("nn.conv2d")

.set\_attrs\_type<Conv2DAttrs>()

.set\_num\_inputs(2)

.add\_argument("data", "Tensor", "The input tensor.")

.add\_argument("weight", "Tensor", "The weight tensor.")

.set\_support\_level(2)

.add\_type\_rel("Conv2D", Conv2DRel<Conv2DAttrs>)

.set\_attr<FInferCorrectLayout>("FInferCorrectLayout", ConvInferCorrectLayout<Conv2DAttrs>);

这里的“nn.conv2d”作用等同于ID，在relay lower的时候会用这个id寻找schedule的实现。注册采用经典的链式调用添加属性，很符合注册机的格调。

此处有一个和ONNX类型的编译器不同的地方，尝试print(call\_node)的时候会发现并不仅仅是打印了一个节点，而是所有后向DFS过程中依赖的节点，例如此demo中第一个conv和第二个conv节点分别打印得到的结果是

first conv

%input0: Tensor[(1, 3, 224, 224), float32];

free\_var %conv1.weight: Tensor[(64, 3, 7, 7), float32];

nn.conv2d(%input0, %conv1.weight, strides=[2, 2], padding=[3, 3, 3, 3], channels=64, kernel\_size=[7, 7])

second conv

free\_var %input0: Tensor[(1, 3, 224, 224), float32];

free\_var %conv1.weight: Tensor[(64, 3, 7, 7), float32];

%0 = nn.conv2d(%input0, %conv1.weight, strides=[2, 2], padding=[3, 3, 3, 3], channels=64, kernel\_size=[7, 7]);

free\_var %bn1.weight: Tensor[(64), float32];

free\_var %bn1.bias: Tensor[(64), float32];

free\_var %bn1.running\_mean: Tensor[(64), float32];

free\_var %bn1.running\_var: Tensor[(64), float32];

%1 = nn.batch\_norm(%0, %bn1.weight, %bn1.bias, %bn1.running\_mean, %bn1.running\_var);

%2 = %1.0;

%3 = nn.relu(%2);

%4 = nn.max\_pool2d(%3, pool\_size=[3, 3], strides=[2, 2], padding=[1, 1, 1, 1]);

free\_var %layer1.0.conv1.weight: Tensor[(64, 64, 3, 3), float32];

nn.conv2d(%4, %layer1.0.conv1.weight, padding=[1, 1, 1, 1], channels=64, kernel\_size=[3, 3])

这种DFS形式的输出原因在于Call这个原语类型的打印方式。Call的\_\_str\_\_方法来自于Node @ tvm/python/tvm/base.py：return \_ffi\_api.PrettyPrint(self)，调用链

RelayTextPrinter::PrintFinal ->

RelayTextPrinter::PrintScope ->

RelayTextPrinter::Print ->

RelayTextPrinter::VisitExpr

最终调用

@ tvm/src/printer/text\_printer.cc

String PrettyPrint(const ObjectRef& node) {

Doc doc;

doc << TextPrinter(false, nullptr, false).PrintFinal(node);

return doc.str();

}

…

Doc RelayTextPrinter::VisitExpr(const Expr& expr) {

auto fcheck\_visited = [this](const Expr& expr) { return this->CheckVisited(expr); };

auto fvisit\_leaf = [this](const Expr& expr) { return this->VisitLeaf(expr); };

if (fcheck\_visited(expr)) {

return memo\_[expr];

} else {

ExpandDataflow(expr, fcheck\_visited, fvisit\_leaf);

return memo\_[expr];

}

}

核心过程是RelayTextPrinter::VisitExpr，使用类似visitor的机制进行DFS节点的访问，从而访问所有依赖的节点。Vistior机制可以参考[TVM之设计模式解读（一）--visitor模式](https://zhuanlan.zhihu.com/p/341334406)进行理解。所有节点都进行转换之后得到的是输出的表达式，打印得到如下形式

free\_var %input0: Tensor[(1, 3, 224, 224), float32];

free\_var %conv1.weight: Tensor[(64, 3, 7, 7), float32];

%0 = nn.conv2d(%input0, %conv1.weight, strides=[2, 2], padding=[3, 3, 3, 3], channels=64, kernel\_size=[7, 7]);

free\_var %bn1.weight: Tensor[(64), float32];

free\_var %bn1.bias: Tensor[(64), float32];

free\_var %bn1.running\_mean: Tensor[(64), float32];

free\_var %bn1.running\_var: Tensor[(64), float32];

%1 = nn.batch\_norm(%0, %bn1.weight, %bn1.bias, %bn1.running\_mean, %bn1.running\_var);

…

%91 = nn.dense(%89, %90, units=1000);

free\_var %fc.bias: Tensor[(1000), float32];

add(%91, %fc.bias)

注意此处还不是Function形式，输出是tvm::relay::TupleNode节点后向DFS过程记录下的所有doc stack。

# Step6 填充IRModule

转换之后的Tuple形式需要进行转换，变成Function形式保存到Step1中创建的IRModule中，并进行一些基础的信息过滤。

## Step 6.1 确定无依赖参数（free\_vars）

@ tvm/python/tvm/relay/frontend/pytorch.py

def from\_pytorch(script\_module, input\_infos, custom\_convert\_map=None, default\_dtype="float32"):

…

*f*or arg in \_analysis.free\_vars(ret):

if arg.name\_hint not in tvm\_params.keys():

data\_inputs.append(arg)

else:

func\_args.append(arg)

-> @ tvm/src/relay/analysis/util.cc

class VarVisitor : protected MixedModeVisitor, protected PatternVisitor {

public:

Array<Var> Free(const Expr& expr) {

this->VisitExpr(expr);

Array<Var> ret;

for (const auto& v : vars\_.data) {

if (bound\_vars\_.set.count(v) == 0) {

ret.push\_back(v);

}

}

return ret;

}

…

void VisitExpr\_(const VarNode\* var) final { vars\_.Insert(GetRef<Var>(var)); }

}

以及后向DFS函数的核心部分

@ tvm/src/relay/ir/expr\_functor.cc

void MixedModeVisitor::VisitExpr(const Expr& expr) {

auto fcheck\_visited = [this](const Expr& expr) { return this->CheckVisited(expr); };

auto fvisit\_leaf = [this](const Expr& expr) { return this->VisitLeaf(expr); };

if (visit\_counter\_[expr.get()] < visit\_limit\_) {

ExpandDataflow(expr, fcheck\_visited, fvisit\_leaf);

}

}

@ tvm/src/relay/ir/expr\_functor.h

template <typename FCheckVisited, typename FVisitLeaf, typename FExpandExpr>

void ExpandDataflow(Expr expr, FCheckVisited fcheck\_visited, FVisitLeaf fvisit\_leaf, FExpandExpr fexpand\_expr) {

std::deque<v\_info> stack;

auto fpush\_to\_stack = [&fcheck\_visited, &stack](const Expr& expr) {

if (!fcheck\_visited(expr)) {

stack.emplace\_front(v\_info(expr));

}

};

fpush\_to\_stack(expr);

while (stack.size() > 0) {

v\_info\* front = &stack.front();

if (fcheck\_visited(front->node)) {

stack.pop\_front();

} else if (front->children\_expanded) {

fvisit\_leaf(front->node);

// TODO(d-smirnov): this is for compatibility with current implementation of MixedModeVisitor

stack.pop\_front();

} else {

front->children\_expanded = true;

for (auto e : fexpand\_expr(front->node)) {

fpush\_to\_stack(e);

}

}

}

}

结合起来看就是用后向DFS的顺序visit每个节点，如果是VarNode就记录var，最后返回。VarNode对应的就是Step4中的inputs和params。bounded\_vars\_在此demo中没有遇到，暂时不深入研究。

## Step 6.2 包装function

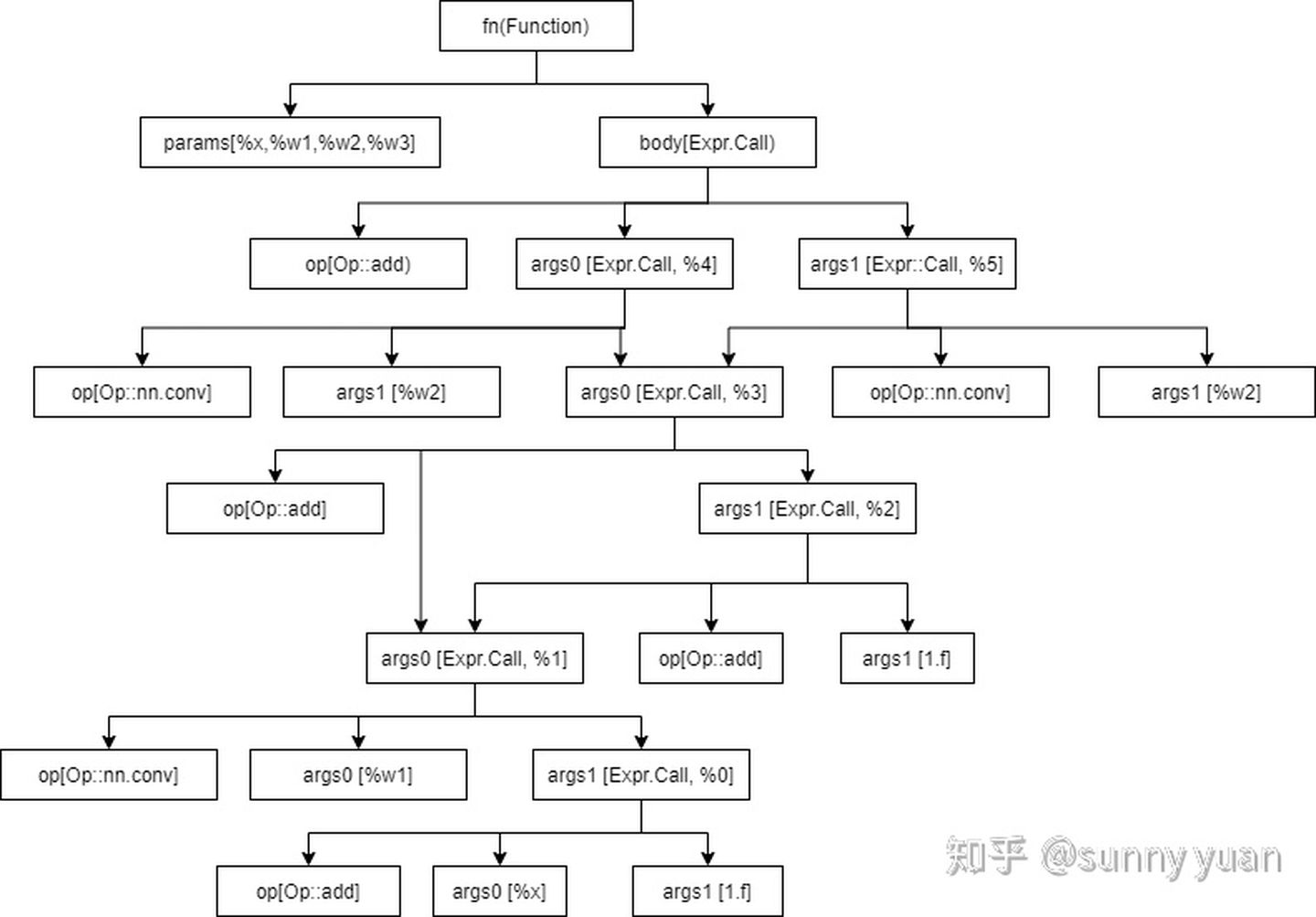
@ tvm/python/tvm/relay/frontend/pytorch.py

def from\_pytorch(script\_module, input\_infos, custom\_convert\_map=None, default\_dtype="float32"):

…

mod["main"] = tvm.relay.Function(func\_args, ret)

这里把ret（TupleNode）和func\_args（step6.2中过滤得到的free\_vars）包装成一个function，可以类比tensorflow的graph定义function，都是使用函数记录整个计算过程。Function对象是TVM的一个核心结构，借用[TVM之Relay IR实践](https://zhuanlan.zhihu.com/p/339348734)这篇文章的一张图直观感受一下。



可以分别打印mod["main"], mod["main"].body, mod["main"].body.op会发现

mod["main"]是一个完整的func定义，结构如

fn (%input0: Tensor[(1, 3, 224, 224), float32], %conv1.weight: Tensor[(64, 3, 7, 7), float32],…){

%0 = nn.conv2d(%input0, %conv1.weight, strides=[2, 2], padding=[3, 3, 3, 3], channels=64, kernel\_size=[7, 7]);

%1 = nn.batch\_norm(%0, %bn1.weight, %bn1.bias, %bn1.running\_mean, %bn1.running\_var);

%2 = %1.0;

%3 = nn.relu(%2);

…

add(%91, %fc.bias)

}

mod["main"].body则是Step5中的输出，即类型是Call。

mod["main"].body.op是add，类型是Op，即function DFS遍历的时候起始的节点。

## Step 6.3 去除无用代码

@ tvm/python/tvm/relay/frontend/pytorch.py

def from\_pytorch(script\_module, input\_infos, custom\_convert\_map=None, default\_dtype="float32"):

…

return transform.RemoveUnusedFunctions()(mod), tvm\_params

第一次使用到transforms这个模块，这个模块在TVM中很重要，但负责对计算图进行修改的功能，可以理解为编译器的中间层。此处涉及的Pass难度不高，可以用来熟悉一下此模块结构。通过python调用C++最终调用到

@ tvm/src/relay/backend/vm/removed\_unused\_funcs.cc

IRModule RemoveUnusedFunctions(const IRModule& module, Array<runtime::String> entry\_funcs) {

std::unordered\_set<std::string> called\_funcs{};

for (auto entry : entry\_funcs) {

auto funcs = CallTracer(module).Trace(entry);

called\_funcs.insert(funcs.cbegin(), funcs.cend());

}

auto existing\_functions = module->functions;

for (auto f : existing\_functions) {

auto it = called\_funcs.find(f.first->name\_hint);

if (it == called\_funcs.end()) {

module->Remove(f.first);

}

}

return module;

}

其中核心的搜索部分是CallTracer,继承自ExprVisitor，对于作为entry的func进行DFS搜索查找依赖的func，有用的func来自GloableVar和Function：

@ tvm/src/relay/backend/vm/removed\_unused\_funcs.cc

void VisitExpr\_(const GlobalVarNode\* op) final {

called\_funcs\_.insert(op->name\_hint);

auto func = module\_->Lookup(op->name\_hint);

VisitExpr(func);

}

void VisitExpr\_(const FunctionNode\* func\_node) final {

auto func = GetRef<Function>(func\_node);

if (visiting\_.find(func) == visiting\_.end()) {

visiting\_.insert(func);

for (auto param : func\_node->params) {

ExprVisitor::VisitExpr(param);

}

ExprVisitor::VisitExpr(func\_node->body);

}

}

可以简单打印一下删除的function：

has entry : main

has func : tensor\_array\_unstack\_tensor5\_helper\_int64(remove)

has func : ta\_split\_helper\_int16(remove)

has func : tensor\_concatenate\_int64(remove)

has func : tensor\_array\_split\_float64(remove)

has func : tensor\_take\_uint8(remove)

has func : tensor\_array\_unstack\_tensor5\_int64(remove)

has func : tensor\_array\_split\_float32(remove)

has func : tensor\_concatenate\_int16(remove)

has func : tensor\_array\_split\_int64(remove)

has func : tensor\_array\_unstack\_tensor3\_helper\_uint8(remove)

…

可见主要删除的是没有用到的function。本demo中最终mod中剩下的函数只有main，也就是relay解析之后的主函数 def @main开头的部分。

# 总结

整个解析过程可以分成3部分：

准备：prelude和构建converter，都是为了将torch的信息转换成relay的信息做准备

转换：算子和参数按照对应的处理方式进行转换

简单优化：用了transform模块，这部分后面章节会细说，此处只是简单的无用信息的剔除。

最终得到的是一个经过初步优化的IRModule。