This document presents the IR as of October 17th 2018. See the JIT OVERVIEW.md for more up-to-date info

PyTorch uses an SSA-based IR, which is built of multiple entities: - `Graph` is generally the outermost container for the program representation. At the moment all programs are mostly pure (modulo special operators like `prim::Print` or `prim::PythonOp`), but this will change in the future. - Then, there are `Block`s, which you can treat as functions. They have a list of inputs and outputs, and a (topologically ordered) list of `Node`s. Every `Graph` has a top-level `Block` which is (using C lingo) like a `main` function. - `Node`s, in turn, represent calls to functions (with possibly multiple arguments and returns). - Finally, every single intermediate value that appears in the program is represented using a `Value` - the root `Block`s have a list of input values, and every `Node` takes them as inputs, and returns some more as outputs. Every Value has a `Type` associated with it.

> **Ownership model:** All of the structures above are conventionally passed around as pointers. All `Node`s and `Value`s are owned by the `Graph` in which they appear (and in particular they can't be shared between different `Graph`s). `Type`s are pointed to only from `Value`s, so they are wrapped in `shared_ptr`s.

Every `Block` has a canonical `Node` ordering (a doubly-linked `node_list_`), which determines both the display and the actual order in which operations will get executed once it's compiled into the JIT interpreter bytecode. It is a responsibility of the programmer to ensure that all `Node`s they create appear somewhere in that list, and that **the list is a valid topological ordering**.

With this amount of background, let's take a look at an example. Consider this Python program:

```python
def f(a, b):
  c = a + b
  d = c * c
  e = torch.tanh(d * c)
  return d + (e + e)
```

If we were to translate it into the IR, it could be represented as such a `Graph`:

> How the actual translation from Python to the IR works will be descrbed later in this document.

```
graph(%0 : Double(2)
      %1 : Double(2)) {
  %2 : int = prim::Constant[value=1]()
  %3 : Double(2) = aten::add(%0, %1, %2)
  %4 : Double(2) = aten::mul(%3, %3)
  %5 : Double(2) = aten::mul(%4, %3)
  %6 : Double(2) = aten::tanh(%5)
  %7 : Double(2) = aten::add(%6, %6, %2)
  %8 : Double(2) = aten::add(%5, %7, %2)
  return (%8);
}
```

This is the canonical textual representation of the IR. You should be able to easily find (almost all) of the elements we discussed above. - `graph` is the `Graph` - `%x` are `Values` - `%x : Double(2)` is a type annotation of `Value` `%x` (see below for a list of supported types). - `%x : T1, %y : T2 = namespace::name(%z, %w)` is a `Node` which represents the `namespace::name`operator (this name is usually refered to as the `Nodes` *kind*). It takes `%z` and `%w` `Values` as inputs, and returns two outputs (`%x`, `%y`) of types `T1` and `T2` respectively.

Finally, nodes can have extra pieces of information assigned to them, which are called *attributes*. You can see that it's used in the `prim::Constant` node, which returns the `value` attribute when it's called. There's a fixed list of types you can attach: - `int64_t` - `double` - `Tensor` - `Graph` (useful for e.g. slicing subgraphs that are meant to be fused) - `std::string` - and lists of them (not nested)

Supported types

JIT supports a number of builtin types (and the list is fixed): - `int`, `float`, `bool` - scalars - `Dynamic` - tensors *without any* static information available - `Float(*, *)` - tensors with *partial* static information available (note that not all of it is shown in the textual output). It includes: - data type (`Byte`, `Short`, `Int`, `Long`, `Half`, `Float`, `Double`) - number of dimensions - the device on which their data will reside - boolean indicating if they will need to be differentiated (usually false for evaluation, and true for training) - `Float(1, 3, 224, 224)` - tensors with *full* static information available. **Note that as of today this is almost unused, and you should not assume that those details will be ever present.** It includes: - all of the above - sizes - strides

Control flow

`Blocks` can also be embedded inside `Nodes`, and are used to implement control flow combinators. You can treat them as lambda expressions passed in as arguments (no other way of passing functions by value exists). They can take and return multiple values and close over the lexical environment of the surrounding block (every `Graph` has a default top-level `Block`).

There are two combinators used today.

`prim::If` Implements a conditional statement. The general semantics of this node are as follows:

```
%y_1, ..., %y_r = prim::If(%condition)
  block0() { # TRUE BRANCH, never takes arguments, has to return r outputs
    %t_1, ..., %t_k = some::node(%a_value_from_outer_block)
    -> (%t_1, ..., %t_r)
  }
  block1() { # FALSE BRANCH, never takes arguments, has to return r outputs
    %f_1, ..., %f_m = some::node(%a_value_from_outer_block)
    -> (%f_1, ..., %f_r)
  }
```

Values corresponding to `%y_1, ..., %y_r` will become either `%t_1, ..., %t_r`, or `%f_1, ..., %f_r` depending on the value of `%condition` at runtime (you can see that the node kind of acts as a Phi node in conventional SSA).

Here's an example translation of a Python program:

```
def f(a, b, c):
    d = a + b
    if c:
        e = d + d
    else:
        e = b + d
    return e
```

```
graph(%a : Dynamic
      %b : Dynamic
      %c : Dynamic) {
  %2 : int = prim::Constant[value=1]()
  %3 : Dynamic = aten::add(%a, %b, %2)
  %5 : Dynamic = prim::If(%c)
    block0() {
      %6 : int = prim::Constant[value=1]()
      %7 : Dynamic = aten::add(%3, %3, %6)
      -> (%7)
    }
    block1() {
      %8 : int = prim::Constant[value=1]()
      %9 : Dynamic = aten::add(%b, %3, %8)
      -> (%9)
    }
  return (%5);
}
```

`prim::Loop` Implements a looping construct (covers both `while` and `for` loops). A valid instantiation of this node always looks like this:

```
%y_1, ..., %y_r = prim::Loop(%max_trip_count, %initial_condition, %x_1, ..., %x_r)
  block0(%i, %a_1, ..., %a_r) {
    %b_1, ..., %b_m = some::node(%a_value_from_outer_block, %a_1)
    %iter_condition = some::other_node(%a_2)
    -> (%iter_condition, %b_1, ..., %b_r)
  }
```

The simplest way to explain the semantics is to consider this Python-like pseudo-code:

```
y_1, ..., y_r = x_1, ..., x_r
condition = initial_condition
i = 0
while condition and i < max_trip_count:
    a_1, ..., a_r = y_1, ..., y_r

    ############################################################
    # Actual body of the loop
    b_1, ..., b_m = some::node(a_value_from_outside_of_the_loop, a_1)
    iter_condition = some::node(a_2)
    ############################################################

    y_1, ..., y_r = b_1, ..., b_r
    condition = iter_condition
    i += 1
```

> Note that translations of `for` loops simply pass in a constant `true` for both `%initial_condition` and `%iter_condition`, while for `while` loops `%max_trip_count` is set to the largest value of `int64_t`, and `%i` is unused. Those patterns are recognized by our interpreter and optimized accordingly (e.g. `while` loops don't maintain the loop counter).

For example, this program:

```
def f(x):
    z = x
    for i in range(x.size(0)):
        z = z * z
    return z
```

can be translated as:

```
graph(%z.1 : Dynamic) {
  %3 : bool = prim::Constant[value=1]()
  %1 : int = prim::Constant[value=0]()
  %2 : int = aten::size(%z.1, %1)
  %z : Dynamic = prim::Loop(%2, %3, %z.1)
    block0(%i : int, %5 : Dynamic) {
      %z.2 : Dynamic = aten::mul(%5, %5)
      -> (%3, %z.2)
    }
  return (%z);
}
```

Function calls

At the moment there's way to call a `Graph` from another `Graph`, and all function calls appearing in the frontend result in inlining of the callee's body into the caller. In particular recursive function calls are not supported yet. This will be addressed in a future release.

Node overloading

PyTorch IR supports function overloading (but you can't have two overloads that differ only in their return types). For example, `aten::add` name has usually those overloads associated with it (`Scalar` means `float` or `int` in this case): - `aten::add(Tensor self, Tensor other) -> Tensor` - `aten::add(Tensor self, Scalar other) -> Tensor` - `aten::add(int self, int other) -> int` - `aten::add(float self, float other) -> float`

All of the strings above can actually be parsed into `FunctionSchema` objects, which hold all this infomation in a machine-readable way. A `Node` can be queried for its schema using the `schema()` method (it will check the argument types, and will try to match one of the options for its `kind()`).

Note that the chosen overload is not shown in any way in the textual output. If you're unsure which function does a node resolve to, you might need to check the type annotations of its input values.

JIT interpreter bytecode

`Graph`s are data structures written with the ease of manipulation in mind, and are not meant to be interpreted directly. Instead, they are first transformed into `Code` objects, which hold a list of `std::function`s (individual instructions), and some additional metadata regarding register use. Later, `Code` objects can be executed using `InterpreterState` objects.

The JIT interpreter is a simple stack-based VM (with a number of registers to hold local values). There's a single `Stack` used to pass arguments into and out of every instruction. The aforementioned metadata in `Code` describes how to organize stores/loads between registers and the stack.

`Stack` is really an `std::vector<IValue>`, where `IValue` is our custom tagged union type, which is able to represent all kinds of values that the JIT can accept (it's optimized to be small and lean, and only takes 16B).

There are many tricks that can be applied in the interpreter to make it faster, but we haven't seen it becoming a bottleneck this far, so we haven't spent time on it.

Operator registration

PyTorch JIT supports open registration of new operators, so they can be freely added at runtime e.g. via `dlopen`. The syntax is as follows:

```
RegisterOperators reg({
  Operator(
    // Specify a function signature
    "my_namespace::magic(Tensor a, Tensor b, int c) -> (Tensor, Tensor)",
    // An std::function that should be called to retrieve a callable implementing
    // this operator.
    [](Node *node) -> Operation {
      // Retrieve the multplier attribute from the node
      double multiplier = node->d(attr::multiplier);
      return [multiplier](Stack& stack) -> int {
        torch::Tensor a, b;
        int c;
        torch::jit::pop(stack, a, b, c);
        std::pair<torch::Tensor, torch::Tensor> result = magic_impl(a, b, c);
        torch::jit::push(stack, result.first, result.second);
        return 0; // Always return 0 here.
      }
    })
});
```

Graph specialization

Certain optimization require certain knowledge about the data types and devices of tensors appearing in user programs. To support this, we have a `GraphExecutor`, which is like a wrapper around an interpreter, that additionally checks what kind of inputs were given, and caches execution plans for `Graph`s specialized to their details. For example `Tensor` inputs to `Graph`s get assigned `TensorType`s (dtype, ndim, device, gradient status), and we later attempt to propagate that statically (using `torch/csrc/jit/passes/shape_analysis.cpp`).

This has the drawback that every call to a JITed function has to go through this matching of arguments to specialized graphs, which e.g. causes a 0.5% slowdown for CNNs (which don't even get any optimization benefits at the moment). In the future we might consider ditching the specialization in favor of more JIT-like techniques (gathering statistics about run time values like tensor sizes, and making optimizations in later stages).

Important files

This section contains a list of relatively important files and a brief description of their contents. All paths are relative to `torch/csrc/jit`.

- `ir.h` - implementation of `Graph`, `Block`, `Node`, `Value`

- `type.h` - implementation of `Type`
- `interpreter.cpp` - JIT interpreter (`Code`, `InterpreterImpl`)
- `ivalue.h` - implementation of `IValue`
- `stack.h` - implementation of `Stack`
- `graph_executor.cpp` - a runner for graphs that will specialize them to different argument configurations
- `tracer.h` - tracer for PyTorch code (generates straight line `Graph`s from any code)
- `operator.cpp` - infrastructure for overload resolution and custom operator registration
- `script/` - compiler from TorchScript (think Python AST) to `Graph`s
- `passes/*.cpp` - optimization passes
- `fusers/**/*` - CUDA and CPU codegens for pointwise subgraphs
- `autodiff.cpp` - symbolic AD for `Graph`s
- `symbolic_variable.h` - a helper to make `Graph` building easier

IR construction

There are three main ways of building up the IR.

Tracing This means that you run arbitrary Python/C++ code using PyTorch operators, and we record a straight line trace (control flow gets unrolled and inlined). Good for simple models, bad if you really have data dependent control flow (and it's not only used for metaprogramming). The relevant entry point for this is `torch.jit.trace`.

TorchScript This method implements a simple Python-like language (it's in fact a subset of Python that conforms to its semantics) and a compiler from it to the IR. Great if you need to retain control flow, but a bit annoying if you need more advanced language features.

Manual construction This doesn't really happen anywhere outside of the optimization passes, and is probably not recommended. `SymbolicVariable` is a helper that overloads many `Tensor` operators and makes them insert `Node`s into its `Graph` instead of doing actual compute.

Graph manipulation

As mentioend previously, the IR is really optimized to be easy to manipulate and change. TO help with that there are numerous methods on `Graph`s, `Node`s and `Value`s, and we maintain a lot of extra metadata that allows to quickly check certain conditions (e.g. looking up all use sites of a single `Value` takes constant time, because we have this information cached). Here's a list of the most relevant methods you can find (think of `ArrayRef` as of an `std::vector`, `Symbol` is an interned string):

`Graph`

- `ArrayRef<Value*> inputs()`
- `ArrayRef<Value*> outputs()`
- `graph_node_list nodes()`
- `Value* addInput()`
- `Value* insertInput(size_t offset)`
- `Value* eraseInput(size_t offset)`
- `size_t registerOutput(Value *output);`
- `void eraseOutput(size_t offset)`
- `Value* insert(Symbol opname, ArrayRef<NamedValue> args, ArrayRef<NamedValue> kwargs = {});`
  - This is the most convenient method of adding more nodes to the `Graph`. An example call looks like this: `graph->insert(aten::add, {some_value, 3})` (note how C++ values will get inserted into the `Graph` as constants automatically).
- `Block* block()` (returns the top-level block)
- `void lint()` (throws if the graph violates invariants like having the node list being a valid topological order of data dependencies)
- `void dump()` (prints the graph to `stdout` – useful for debugging)

`Value`

- `const TypePtr& type()`
- `Node* node()` (producer of this `Value`)
- `size_t offset()` (offset into the output list of the `node()`)
- `void replaceAllUsesWith(Value* other)`
- `const use_list& uses()` (`use_list` is `std::vector<Use>`, where `Use` is a struct containing a `Node*` and offset into its input list)
- `Graph* owningGraph()`

`Node`

- `Symbol kind()`
- `ArrayRef<Value*> inputs()`
- `ArrayRef<Value*> outputs()`
- `Value* namedInput(Symbol name)` (lets you look up inputs by their names instead of depdending on the positional order)
- `bool is_constant(Symbol name)` (return true if input `name` is a constant)
- `optional<IValue> get(Symbol name)` (if `is_constant(name)`, returns an `IValue` containing its value)
- `optional<T> get(Symbol name)` (same as above but returns `T` instead of `IValue`)
- `Value* addInput(Value* value)`
- `Value* insertInput(size_t offset, Value* value)`
- `Value* replaceInput(size_t offset, Value* newValue)`
- `Value* replaceInputWith(Value* from, Value* to)`

- `Value* addOutput()`
- `Value* insertOutput(size_t offset)`
- `void eraseOutput(size_t offset)`
- `ArrayRef<Block*> blocks()`
- `Block* addBlock()`
- `void eraseBlock(size_t offset)`
- `void destroy()` (This is dangerous! All references to `Values` produced by this node, and to the node itself become invalid!)
- `void dump()` (Debug print to `stdout`)
- `Block* owningBlock()`
- `Graph* owningGraph()`

A larger example (simple RNN loop)

Building up on everything that I covered so far, here's a Python code that shows you how to inspect example translations into the IR (and shows a simple single-layer RNN). Note that the Python 3 type annotations are supported as well, but this is more portable.

```
import torch

@torch.jit.script
def lstm_cell(input, hidden, w_ih, w_hh, b_ih, b_hh):
    # type: (Tensor, Tuple[Tensor, Tensor], Tensor, Tensor, Tensor, Tensor) -> Tuple[Tensor
    hx, cx = hidden
    gates = torch.mm(input, w_ih.t()) + torch.mm(hx, w_hh.t()) + b_ih + b_hh

    ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)

    ingate = torch.sigmoid(ingate)
    forgetgate = torch.sigmoid(forgetgate)
    cellgate = torch.tanh(cellgate)
    outgate = torch.sigmoid(outgate)

    cy = (forgetgate * cx) + (ingate * cellgate)
    hy = outgate * torch.tanh(cy)

    return hy, cy

@torch.jit.script
def simple_lstm(input, hidden, wih, whh, bih, bhh):
    # type: (Tensor, Tuple[Tensor, Tensor], Tensor, Tensor, Tensor, Tensor) -> Tuple[Tensor
    outputs = []
    inputs = input.unbind(0)
    for seq_idx in range(len(inputs)):
        hidden = lstm_cell(inputs[seq_idx], hidden, wih, whh, bih, bhh)
        hy, _ = hidden
        outputs.append(hy)
    return hidden

print(simple_lstm.graph)
```

```
graph(%input : Dynamic
      %hidden.1 : Tuple
      %wih : Dynamic
      %whh : Dynamic
      %bih : Dynamic
      %bhh : Dynamic) {
  %20 : int = prim::Constant[value=1]()
  %19 : int = prim::Constant[value=4]()
  %10 : bool = prim::Constant[value=1]()
  %7 : int = prim::Constant[value=0]()
  %54 : World = prim::LoadWorld()
  %outputs : Dynamic[] = prim::ListConstruct()
  %inputs : Dynamic[] = aten::unbind(%input, %7)
  %9 : int = aten::len(%inputs)
  %hidden : Tuple, %56 : World = prim::Loop(%9, %10, %hidden.1, %54)
    block0(%seq_idx : int, %14 : Tuple, %55 : World) {
      %13 : Dynamic = aten::select(%inputs, %seq_idx)
      %hx : Dynamic, %cx : Dynamic = prim::TupleUnpack(%14)
      %23 : Dynamic = aten::t(%wih)
      %24 : Dynamic = aten::mm(%13, %23)
      %25 : Dynamic = aten::t(%whh)
      %26 : Dynamic = aten::mm(%hx, %25)
      %27 : Dynamic = aten::add(%24, %26, %20)
      %28 : Dynamic = aten::add(%27, %bih, %20)
      %gates : Dynamic = aten::add(%28, %bhh, %20)
      %30 : Dynamic[] = aten::chunk(%gates, %19, %20)
      %ingate.1 : Dynamic, %forgetgate.1 : Dynamic, %cellgate.1 : Dynamic, %outgate.1 : Dyna
      %ingate : Dynamic = aten::sigmoid(%ingate.1)
      %forgetgate : Dynamic = aten::sigmoid(%forgetgate.1)
      %cellgate : Dynamic = aten::tanh(%cellgate.1)
      %outgate : Dynamic = aten::sigmoid(%outgate.1)
      %39 : Dynamic = aten::mul(%forgetgate, %cx)
      %40 : Dynamic = aten::mul(%ingate, %cellgate)
      %_ : Dynamic = aten::add(%39, %40, %20)
      %42 : Dynamic = aten::tanh(%_)
      %hy : Dynamic = aten::mul(%outgate, %42)
      %hidden.2 : Tuple = prim::TupleConstruct(%hy, %_)
      %49 : World = aten::append(%55, %outputs, %hy)
      -> (%10, %hidden.2, %49)
    }
  %52 : Dynamic, %53 : Dynamic = prim::TupleUnpack(%hidden)
   = prim::StoreWorld(%56)
  return (%52, %53);
}
```