PyTorch 2 internals

A not so short guide to recent PyTorch innovations

Christian S. Perone
(*christian.perone@gmail.com*)
http://blog.christianperone.com

London, UK, Dec 2023

Who Am I

- **Christian S. Perone**
- ML Research Engineer in London/UK
- Blog at
- blog.christianperone.com
- Open-source projects at
- https://github.com/perone
- Twitter @tarantulae

Disclaimer

PyTorch development pace is so fast that no man ever steps in PyTorch code twice, for it's not the same code and he's not the same man.

—Heraclitus, *500 BC*

Section I

Tensors

Tensors

Simply put, TENSORS are a generalization of vectors and matrices. In PyTorch, they are a multi-dimensional matrix containing elements of a **single data type**.

Tensors

Simply put, TENSORS are a generalization of vectors and matrices. In PyTorch, they are a multi-dimensional matrix containing elements of a **single data type**.

```
>>> import torch
>>> t = torch.tensor([[1., -1.], [1., -1.]])
>>> t
tensor([[ 1., -1.]
        [ 1., -1.]])
```

Tensors

Simply put, TENSORS are a generalization of vectors and matrices. In PyTorch, they are a multi-dimensional matrix containing elements of a **single data type**.

```
>>> import torch
>>> t = torch.tensor([[1., -1.], [1., -1.]])
>>> t
tensor([[ 1., -1.]
        [ 1., -1.]])
```

```
>>> t.dtype # They have a type
torch.float32
```

Tensors

Simply put, TENSORS are a generalization of vectors and matrices. In PyTorch, they are a multi-dimensional matrix containing elements of a **single data type**.

```
>>> import torch
>>> t = torch.tensor([[1., -1.], [1., -1.]])
>>> t
tensor([[ 1., -1.]
        [ 1., -1.]])
>>> t.dtype # They have a type
torch.float32
>>> t.shape # a shape
torch.Size([2, 2])
```

Tensors

Simply put, TENSORS are a generalization of vectors and matrices. In PyTorch, they are a multi-dimensional matrix containing elements of a **single data type**.

```
>>> import torch
>>> t = torch.tensor([[1., -1.], [1., -1.]])
>>> t
tensor([[ 1., -1.]
        [ 1., -1.]])
>>> t.dtype # They have a type
torch.float32
>>> t.shape # a shape
torch.Size([2, 2])
>>> t.device # and live in some device
device(type='cpu')
```

Tensors

- Although PyTorch has an elegant *python first* design, all PyTorch heavy work is actually implemented in C++.
- In Python, the integration of C++ code is (usually) done using what is called an **extension**;

Tensors

- Although PyTorch has an elegant *python first* design, all PyTorch heavy work is actually implemented in C++.
- In Python, the integration of C++ code is (usually) done using what is called an **extension**;
- PyTorch uses **ATen**, which is the foundational tensor operation library on which all else is built;

Tensors

- Although PyTorch has an elegant *python first* design, all PyTorch heavy work is actually implemented in C++.
- In Python, the integration of C++ code is (usually) done using what is called an **extension**;
- PyTorch uses **ATen**, which is the foundational tensor operation library on which all else is built;
- To do automatic differentiation, PyTorch uses **Autograd**, which is an augmentation on top of the **ATen** framework;

Tensors

- Although PyTorch has an elegant *python first* design, all PyTorch heavy work is actually implemented in C++.
- In Python, the integration of C++ code is (usually) done using what is called an **extension**;
- PyTorch uses **ATen**, which is the foundational tensor operation library on which all else is built;
- To do automatic differentiation, PyTorch uses **Autograd**, which is an augmentation on top of the **ATen** framework;

Quick recap Python objects

```c
typedef struct {
  PyObject_HEAD
  double ob_fval;
} PyFloatObject;
```

Quick recap Python objects

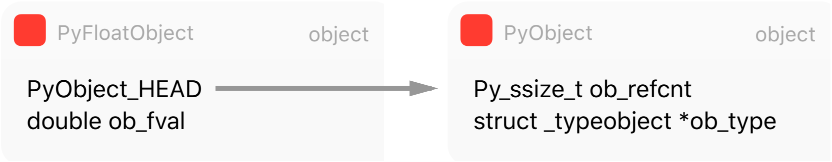
```
typedef struct {
  PyObject_HEAD
  double ob_fval;
} PyFloatObject;
```

```
typedef struct _object {
  Py_ssize_t ob_refcnt;
  struct _typeobject *ob_type;
} PyObject;
```

Quick recap Python objects

```c
typedef struct {
  PyObject_HEAD
  double ob_fval;
} PyFloatObject;
```

```c
typedef struct _object {
  Py_ssize_t ob_refcnt;
  struct _typeobject *ob_type;
} PyObject;
```



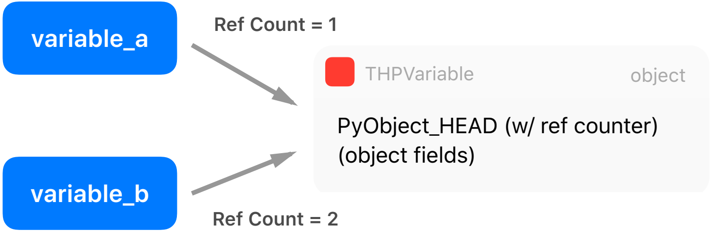
Quick recap Python objects

```
struct THPVariable {
        PyObject_HEAD;
        c10::MaybeOwned<at::Tensor> cdata;
        PyObject* backward_hooks = nullptr;
        PyObject* post_accumulate_grad_hooks = nullptr;
};
```

The **TH** prefix is from **T**orc**H**, and **P** means **P**ython.

Quick recap Python objects

```
struct THPVariable {
        PyObject_HEAD;
        c10::MaybeOwned<at::Tensor> cdata;
        PyObject* backward_hooks = nullptr;
        PyObject* post_accumulate_grad_hooks = nullptr;
};
```



The **TH** prefix is from Torc**H**, and **P** means **P**ython.

In Python, everything is an object

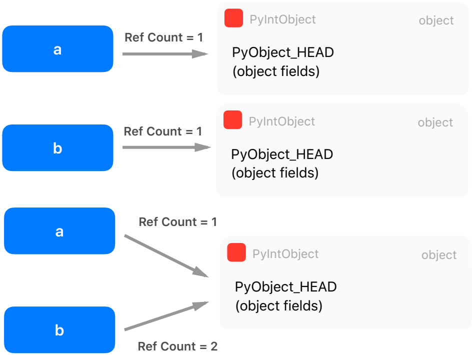
```
>>> a = 300
>>> b = 300
>>> a is b
False
```

In Python, everything is an object

```
>>> a = 300
>>> b = 300
>>> a is b
False
>>> a = 200
>>> b = 200
>>> a is b
True
```

In Python, everything is an object

```
>>> a = 300
>>> b = 300
>>> a is b
False
>>> a = 200
>>> b = 200
>>> a is b
True
```



A typical Python program spend much of its time allocating/deallocating integers. CPython then caches the small integers.

Zero-copying tensors

It is very common to load tensors in **numpy** and convert them to PyTorch, or vice-versa;

```
>>> np_array = np.ones((2,2))
>>> np_array
array([[1., 1.],
       [1., 1.]])
```

Underline after an operation means an in-place operation.

Zero-copying tensors

It is very common to load tensors in **numpy** and convert them to PyTorch, or vice-versa;

```
>>> np_array = np.ones((2,2))
>>> np_array
array([[1., 1.],
       [1., 1.]])
>>> torch_array = torch.tensor(np_array)
>>> torch_array
tensor([[1., 1.],
        [1., 1.]], dtype=torch.float64)
```

Underline after an operation means an in-place operation.

Zero-copying tensors

It is very common to load tensors in **numpy** and convert them to PyTorch, or vice-versa;

```
>>> np_array = np.ones((2,2))
>>> np_array
array([[1., 1.],
       [1., 1.]])
>>> torch_array = torch.tensor(np_array)
>>> torch_array
tensor([[1., 1.],
        [1., 1.]], dtype=torch.float64)
>>> torch_array.add_(1.0)
```

Underline after an operation means an in-place operation.

Zero-copying tensors

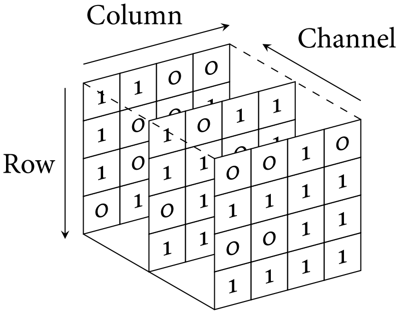
It is very common to load tensors in **numpy** and convert them to PyTorch, or vice-versa;

```
>>> np_array = np.ones((2,2))
>>> np_array
array([[1., 1.],
       [1., 1.]])
>>> torch_array = torch.tensor(np_array)
>>> torch_array
tensor([[1., 1.],
        [1., 1.]], dtype=torch.float64)
>>> torch_array.add_(1.0)
>>> np_array
array([[1., 1.], # array is intact, a copy was made
       [1., 1.]])
```

Underline after an operation means an in-place operation.

Zero-copying tensors

- Now imagine that you have a batch of 128 images, 3 channels each (RGB) and with size of 224x224;



- This will yield a size in memory of ~ **74MB**. We don't want to duplicate memory (except when copying them to discrete GPUs of course);

Zero-copying tensors

Let's see now a slightly different code using the function `torch.from_numpy()` this time:

```
>>> np_array
array([[1., 1.],
       [1., 1.]])
>>> torch_array = torch.from_numpy(np_array)
```

Zero-copying tensors

Let's see now a slightly different code using the function `torch.from_numpy()` this time:

```
>>> np_array
array([[1., 1.],
       [1., 1.]])
>>> torch_array = torch.from_numpy(np_array)
>>> torch_array.add_(1.0)
```

Zero-copying tensors

Let's see now a slightly different code using the function `torch.from_numpy()` this time:

```
>>> np_array
array([[1., 1.],
       [1., 1.]])
>>> torch_array = torch.from_numpy(np_array)
>>> torch_array.add_(1.0)
>>> np_array
array([[2., 2.],
       [2., 2.]])
```

Zero-copying tensors

Let's see now a slightly different code using the function `torch.from_numpy()` this time:

```
>>> np_array
array([[1., 1.],
       [1., 1.]])
>>> torch_array = torch.from_numpy(np_array)
>>> torch_array.add_(1.0)
>>> np_array
array([[2., 2.],
       [2., 2.]])
```

The original numpy array **was changed**, because it used a **zero-copy** operation.

Zero-copying tensors

Difference between **in-place** and **standard operations** might not be so clear in some cases:

```
>>> np_array
array([[1., 1.],
       [1., 1.]])
>>> torch_array = torch.from_numpy(np_array)
```

Zero-copying tensors

Difference between **in-place** and **standard operations** might not be so clear in some cases:

```
>>> np_array
array([[1., 1.],
       [1., 1.]])
>>> torch_array = torch.from_numpy(np_array)
>>> np_array = np_array + 1.0
```

Zero-copying tensors

Difference between **in-place** and **standard operations** might not be so clear in some cases:

```
>>> np_array
array([[1., 1.],
       [1., 1.]])
>>> torch_array = torch.from_numpy(np_array)
>>> np_array = np_array + 1.0
>>> torch_array
tensor([[1., 1.],
        [1., 1.]], dtype=torch.float64)
```

Zero-copying tensors

Difference between **in-place** and **standard operations** might not be so clear in some cases:

```
>>> np_array
array([[1., 1.],
       [1., 1.]])
>>> torch_array = torch.from_numpy(np_array)
>>> np_array = np_array + 1.0
>>> torch_array
tensor([[1., 1.],
        [1., 1.]], dtype=torch.float64)
```

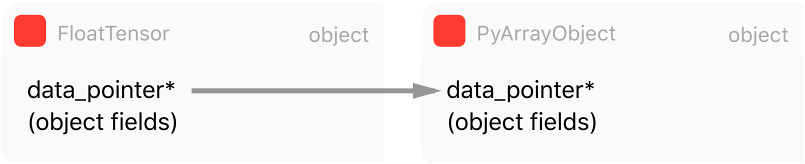
However, if you use `np_array += 1.0`, that is an in-place operation that will change `torch_array` memory.

Zero-copying tensors

```
at::Tensor tensor_from_numpy(PyObject* obj, (omitted)) {
        // some parts omitted for brevity
        auto array = (PyArrayObject*)obj;
        int ndim = PyArray_NDIM(array);
        auto sizes = to_aten_shape(ndim, PyArray_DIMS(array));
        auto strides = to_aten_shape(ndim, PyArray_STRIDES(array));
        void* data_ptr = PyArray_DATA(array);
        Py_INCREF(obj);
        return at::lift_fresh(at::from_blob(
        data_ptr, sizes, strides,
        [obj](void* data) {
                pybind11::gil_scoped_acquire gil;
                Py_DECREF(obj);
        },
        at::device(kCPU).dtype(numpy_dtype_to_aten(PyArray_TYPE(array))
}
```

Pay attention to the reference counting using `Py_INCREF()` and the call to `at::from_blob()` function.

Data pointers



The tensor `FloatTensor` did a copy of the numpy array **data pointer** and not of the contents. The reference is kept safe by the Python reference counting mechanism.

Tensor Storage

The abstraction responsible for holding the data isn't actually the `Tensor`, but the `Storage`.

Tensor Storage

The abstraction responsible for holding the data isn't actually the `Tensor`, but the `Storage`.

```
struct C10_API StorageImpl : public c10::intrusive_ptr_target {
// (...)
private:
    // (...)
    DataPtr data_ptr_;
    SymInt size_bytes_;
    Allocator* allocator_;
    // (...)
}
```

Tensor Storage

The abstraction responsible for holding the data isn't actually the `Tensor`, but the `Storage`.

```
struct C10_API StorageImpl : public c10::intrusive_ptr_target {
// (...)
private:
    // (...)
    DataPtr data_ptr_;
    SymInt size_bytes_;
    Allocator* allocator_;
    // (...)
}
```

- Holds a pointer to the raw data and contains information such as the size and allocator;
- Storage is a dumb abstraction, there is no metadata telling us how to interpret the data it holds;

Tensor Storage

- The `Storage` abstraction is very powerful because it decouples the raw data and how we can interpret it;

Tensor Storage

- The `Storage` abstraction is very powerful because it decouples the raw data and how we can interpret it;
- We can have multiple tensors sharing the **same storage**, but with different interpretations, also called **views**, but **without duplicating** memory:

Tensor Storage

- The `Storage` abstraction is very powerful because it decouples the raw data and how we can interpret it;
- We can have multiple tensors sharing the **same storage**, but with different interpretations, also called **views**, but **without duplicating** memory:

```
>>> x = torch.ones((2, 2))
>>> x_view = x.view(4)
>>> x_data = x.untyped_storage().data_ptr()
>>> x_view_data = x_view.untyped_storage().data_ptr()
>>> x_data == x_view_data
True
```

Tensor Storage

- The `Storage` abstraction is very powerful because it decouples the raw data and how we can interpret it;
- We can have multiple tensors sharing the **same storage**, but with different interpretations, also called **views**, but **without duplicating** memory:

```
>>> x = torch.ones((2, 2))
>>> x_view = x.view(4)
>>> x_data = x.untyped_storage().data_ptr()
>>> x_view_data = x_view.untyped_storage().data_ptr()
>>> x_data == x_view_data
True
```

- `x_view` is a different view (interpretation) of the same data present in the underlying storage that is shared between both tensors.

Memory allocators (CPU/GPU)

- The tensor storage can be allocated either in the CPU memory or GPU, therefore a mechanism is required to switch between these different allocations:

Memory allocators (CPU/GPU)

- The tensor storage can be allocated either in the CPU memory or GPU, therefore a mechanism is required to switch between these different allocations:

```
struct C10_API Allocator {
        virtual ~Allocator() = default;
        virtual DataPtr allocate(size_t n) const = 0;
        virtual DeleterFnPtr raw_deleter() const {...}
        void* raw_allocate(size_t n) {...}
        void raw_deallocate(void* ptr) {...}
};
```

- There are `Allocator`s that will use the GPU allocators such as `cudaMalloc()` when the storage should be used for the GPU or `posix_memalign()` POSIX functions for data in the CPU memory.

CUDA caching allocator

PyTorch uses a CUDA caching allocator that maintains a cache of allocations with the `Block` structure:

CUDA caching allocator

PyTorch uses a CUDA caching allocator that maintains a cache of allocations with the `Block` structure:

```
struct Block {
        int device; // gpu
        cudaStream_t stream; // allocation stream
        size_t size; // block size in bytes
        BlockPool* pool{nullptr}; // owning memory pool
        void* ptr{nullptr}; // memory address
    bool allocated{false}; // in-use flag
        Block* prev{nullptr}; // prev block if split from a
        Block* next{nullptr}; // next block if split from a
        // (...)
}
```

The `torch.cuda.empty_cache()` will release all unused blocks.

The big picture

- The `Tensor` has a `Storage` which in turn has a pointer to the raw data and to the `Allocator` to allocate memory according to the destination device.

Section II

JIT

JIT - Just-in-time compiler

- PyTorch is eager by design, which means that it is easily hackable to debug, inspect, etc;

JIT - Just-in-time compiler

- PyTorch is eager by design, which means that it is easily hackable to debug, inspect, etc;
- However, this poses problems for optimization and for decoupling it from Python (the model itself is Python code);

JIT - Just-in-time compiler

- PyTorch is eager by design, which means that it is easily hackable to debug, inspect, etc;
- However, this poses problems for optimization and for decoupling it from Python (the model itself is Python code);
- PyTorch 1.0 introduced `torch.jit`, which has two main methods to convert a PyTorch model to a serializable and optimizable format;
- **TorchScript** was also introduced as a statically-typed subset of Python;

JIT - Just-in-time compiler

Two very different worlds with their own requirements.

EAGER MODE

Prototype, debug, train, experiment

tracing

»

scripting

SCRIPT MODE

Optimization, other languages, deployment

Tracing

```python
def my_function(x):
    if x.mean() > 1.0:
        r = torch.tensor(1.0)
    else:
        r = torch.tensor(2.0)
    return r
```

Tracing

```
def my_function(x):
    if x.mean() > 1.0:
        r = torch.tensor(1.0)
    else:
        r = torch.tensor(2.0)
    return r

>>> ftrace = torch.jit.trace(my_function, (torch.ones(2, 2)))
```

Tracing

```
def my_function(x):
    if x.mean() > 1.0:
        r = torch.tensor(1.0)
    else:
        r = torch.tensor(2.0)
    return r
```

```
>>> ftrace = torch.jit.trace(my_function, (torch.ones(2, 2)))
```

```
>>> ftrace.graph
graph(%x : Float(2, 2, strides=[2, 1], requires_grad=0, device=cpu)):
%5 : Float(requires_grad=0, device=cpu) = prim::Constant[value={2}]()
%6 : Device = prim::Constant[value="cpu"]()
%7 : int = prim::Constant[value=6]()
%8 : bool = prim::Constant[value=0]()
%9 : bool = prim::Constant[value=0]()
%10 : NoneType = prim::Constant()
%11 : Float(requires_grad=0, device=cpu) = aten::to(%5, %6, %7, %8, %9,
%12 : Float(requires_grad=0, device=cpu) = aten::detach(%11)
return (%12)
```

Tracing

To call the JIT'ed function, just call the `forward()` method:

```
>>> x = torch.ones(2, 2)
>>> ftrace.forward(x)
tensor(2.)
```

Tracing

To call the JIT'ed function, just call the `forward()` method:

```
>>> x = torch.ones(2, 2)
>>> ftrace.forward(x)
tensor(2.)
```

However, tracing will not record any control-flow like if statements or loops, it executes the code with the given context and creates the graph. You can see this limitation below:

```
>>> x = torch.ones(2, 2).add_(1.0)
>>> ftrace.forward(x)
tensor(2.)
```

According to `my_function()`, result should have been 1.0. Tracing also checks for differences between traced and Python function, but what about **Dropout** ?

Scripting

Another alternative is to use **scripting**, where you can use decorators such as `@torch.jit.script`:

```
@torch.jit.script
def my_function(x):
    if bool(x.mean() > 1.0):
        r = 1
    else:
        r = 2
    return r
```

Scripting

```
>>> my_function.graph
graph(%x.1 : Tensor):
%2 : NoneType = prim::Constant()
%4 : float = prim::Constant[value=1.]()
%9 : int = prim::Constant[value=1]()
%10 : int = prim::Constant[value=2]()
%3 : Tensor = aten::mean(%x.1, %2)
%5 : Tensor = aten::gt(%3, %4)
%7 : bool = aten::Bool(%5)
%r : int = prim::If(%7)
block0():
-> (%9)
block1():
-> (%10)
return (%r)
```

Scripting

The `my_function()` is now a `torch.jit.ScriptFunction`:

```
>>> type(my_function)
torch.jit.ScriptFunction
```

Scripting

The `my_function()` is now a `torch.jit.ScriptFunction`:

```
>>> type(my_function)
torch.jit.ScriptFunction
```

When we check the results again:

```
>>> x = torch.ones(2, 2)
>>> my_function(x)
2
```

Scripting

The `my_function()` is now a `torch.jit.ScriptFunction`:

```
>>> type(my_function)
torch.jit.ScriptFunction
```

When we check the results again:

```
>>> x = torch.ones(2, 2)
>>> my_function(x)
2

>>> x = torch.ones(2, 2).add_(1.0)
>>> my_function(x)
1
```

Control-flow logic was preserved !

Why TorchScript ?

- The concept of having a well-defined **Intermediate Representation** (IR) is very powerful, it's the main concept behind LLVM platform as well;

Why TorchScript ?

- The concept of having a well-defined **Intermediate Representation** (IR) is very powerful, it's the main concept behind LLVM platform as well;
- This opens the door to:
  - Decouple the model (computationl graph) from Python runtime;

Why TorchScript ?

- The concept of having a well-defined **Intermediate Representation** (IR) is very powerful, it's the main concept behind LLVM platform as well;
- This opens the door to:
  - Decouple the model (computationl graph) from Python runtime;
  - Use it in production with C++ (no GIL) or other languages;

Why TorchScript ?

- The concept of having a well-defined **Intermediate Representation** (IR) is very powerful, it's the main concept behind LLVM platform as well;
- This opens the door to:
  - Decouple the model (computationl graph) from Python runtime;
  - Use it in production with C++ (no GIL) or other languages;
  - Capitalize on optimizations (whole program);

Why TorchScript ?

- The concept of having a well-defined **Intermediate Representation** (IR) is very powerful, it's the main concept behind LLVM platform as well;
- This opens the door to:
  - Decouple the model (computationl graph) from Python runtime;
  - Use it in production with C++ (no GIL) or other languages;
  - Capitalize on optimizations (whole program);
  - Split the development world of hackable and easy to debug from the world of putting these models in production and optimize them.

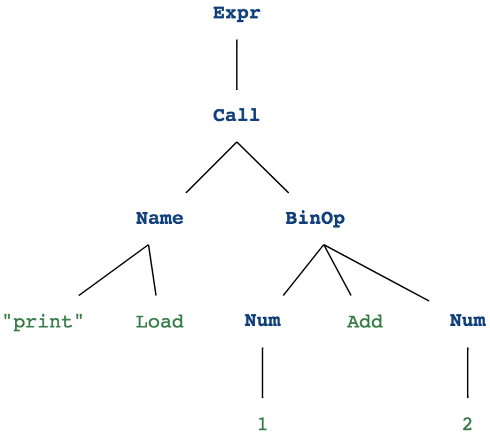
Building the IR

To build the IR, PyTorch takes leverage of the Python **Abstract Syntax Tree** (AST) which is a tree representation of the syntactic structure of the source code.

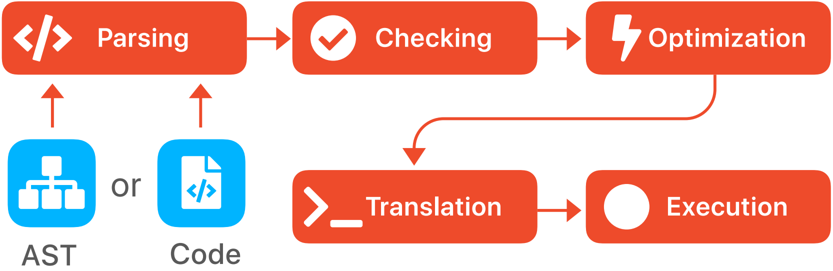
```
>>> ast_mod = ast.parse("print(1 + 2)")
>>> astpretty.pprint(ast_mod.body[0], show_offsets=False)

Expr(
    value=Call(
        func=Name(id='print', ctx=Load()),
        args=[
            BinOp(
                left=Num(n=1),
                op=Add(),
                right=Num(n=2),
            ),
        ],
        keywords=[],
    ),
)
```

Building the IR

```python
print(1 + 2)
```



PyTorch JIT Phases



Optimizations

Many optimizations can be used on the computational graph of the model, such as **Loop Unrolling**:

```
for i.. i+= 1
  for j..
    code(i, j)
```

```
for i.. i+= 4
  for j..
    code(i, j)
    code(i+1, j)
    code(i+2, j)
    code(i+3, j)
remainder loop
```

Optimizations

Also **Peephole optimizations** such as:

```
x.t().t() = x
```

Optimizations

Also **Peephole optimizations** such as:

```
x.t().t() = x
```

Example:

```
def dumb_function(x):
    return x.t().t()
>>> traced_fn = torch.jit.trace(dumb_function,
...                             torch.ones(2,2))
>>> traced_fn.graph_for(torch.ones(2,2))
graph(%x : Tensor):
return (%x)
```

Optimizations

Also **Peephole optimizations** such as:

```
x.t().t() = x
```

Example:

```
def dumb_function(x):
    return x.t().t()
>>> traced_fn = torch.jit.trace(dumb_function,
...                                torch.ones(2,2))
>>> traced_fn.graph_for(torch.ones(2,2))
graph(%x : Tensor):
return (%x)
```

Other optimizations include **Constant Propagation**, **Dead Code Elimination** (DCE), **fusion**, **inlining**, etc.

Serialization

```
>>> resnet = torch.jit.trace(models.resnet18(),
...                          torch.rand(1, 3, 224, 224))
>>> resnet.save("resnet.pt")
```

Serialization

```
>>> resnet = torch.jit.trace(models.resnet18(),
...                          torch.rand(1, 3, 224, 224))
>>> resnet.save("resnet.pt")
```

```
$ file resnet.pt
resnet.pt: Zip archive data
```

Serialization

```
>>> resnet = torch.jit.trace(models.resnet18(),
...                           torch.rand(1, 3, 224, 224))
>>> resnet.save("resnet.pt")
```

```
$ file resnet.pt
resnet.pt: Zip archive data
```

```
$ unzip resnet.pt
Archive:  resnet.pt
extracting: resnet/version
extracting: resnet/code/__torch__/torchvision/models/resnet
extracting: resnet/data/0
(...)
```

Serialization

code/resnet.py

```
def forward(self: (...) resnet.ResNet,
x: Tensor) -> Tensor:
        # (...)
        _0 = (bn1).forward((conv1).forward(x, ), )
        _1 = (maxpool).forward((relu).forward(_0, ), )
        _2 = (layer2).forward((layer1).forward(_1, ), )
        _3 = (layer4).forward((layer3).forward(_2, ), )
        input = torch.flatten((avgpool).forward(_3, ), 1)
        return (fc).forward(input, )
```

Using the model in C++

In the example below we load the exported TorchScript model and run the `forward()` using Torch's C++ API.

Example of loading a traced model in PyTorch C++ API:

```
#include <torch/script.h>
int main(int argc, const char* argv[])
{
  auto module = torch::jit::load("resnet.pt");
  std::vector<torch::jit::IValue> inputs;
  inputs.push_back(torch::ones({1, 3, 224, 224}));
  at::Tensor output = module->forward(inputs).toTensor();
}
```

Executing

Just like Python interpreter executes your code, PyTorch has an interpreter that executes the IR instructions:

```
bool runImpl(Stack& stack) {
    // (...) omitted
    try {
      while (true) {
        Frame& frame = frames.back();
        Instruction inst = INST_FETCH(0);
        switch (inst.op) {
          case INST(ENTER): {
            INST_GUARD;
            const auto& obj = peek(stack, 0, 1);
            TORCH_INTERNAL_ASSERT(obj.isObject());
            entered_objects.push_back(obj);
          }
            INST_NEXT;
    // (...) omitted
```

Section III

Dynamo

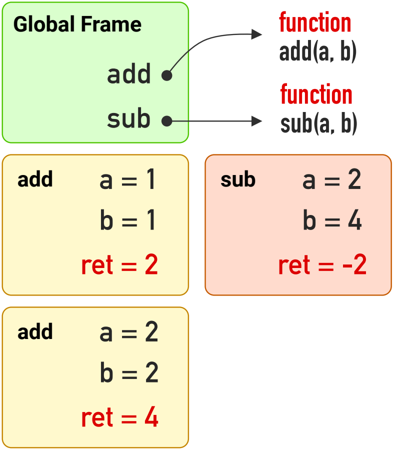
Python Stack Frames

Conceptually, an interpreter executes instructions within a context, which we refer to as **frames**.

Python Stack Frames

Conceptually, an interpreter executes instructions within a context, which we refer to as **frames**.

A function call generates a new frame, which is cleared when the function returns. This process is facilitated by a stack, with the frames being placed in order, thus giving rise to the term **stack frames**.



CPython Frame Evaluation

Frame evaluation in CPython happens in `_PyEval_EvalFrameDefault` function. This is where the core of Python execution is, all bytecode gets executed here and this function is heavily optimized:

```
for (;;) {
        opcode = next_uop->opcode;
        oparg = next_uop->oparg;
        // (...)
        case UNARY_NOT: {
                PyObject *value;
                PyObject *res;
                value = stack_pointer[-1];
                assert(PyBool_Check(value));
                res = Py_IsFalse(value) ? Py_True : Py_False;
                stack_pointer[-1] = res;
                break;
        }
}
```

TorchDynamo

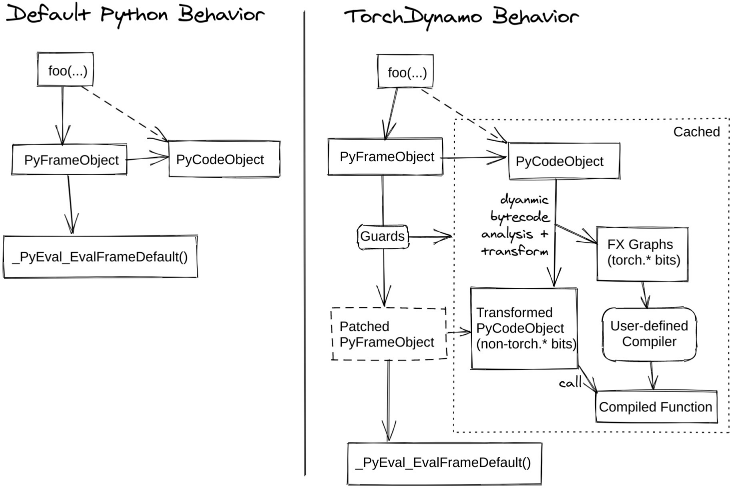
- TorchScript can be limiting in some situations. **TorchDynamo** can overcome some of the limitations while still allowing *unmodified Python code* to be compiled;

- TorchDynamo was introduced as a way to acquire graphs, it uses a feature introduced in CPython 3.6 (PEP 523) where the frame evaluation API was exposed to allow specification of a per-interpreter function pointer to **handle the evaluation of frames**;

TorchDynamo

```
void enable_eval_frame_shim(PyThreadState* tstate) {
  #if PY_VERSION_HEX >= 0x03090000
  if (_PyInterpreterState_GetEvalFrameFunc(tstate->interp) !=
  &custom_eval_frame_shim) {
    DEBUG_CHECK(previous_eval_frame == NULL);
    previous_eval_frame = \
      _PyInterpreterState_GetEvalFrameFunc(tstate->interp);
    _PyInterpreterState_SetEvalFrameFunc(tstate->interp,
      &custom_eval_frame_shim);
  }
  #else
  if (tstate->interp->eval_frame != &custom_eval_frame_shim) {
    // First call
    tstate->interp->eval_frame = &custom_eval_frame_shim;
  }
  #endif
}
```

TorchDynamo



TorchDynamo behavior. Credit of the diagram to Jason Ansel.

TorchDynamo

- TorchDynamo can switch back to the default Python frame evaluation when it is not able to capture the graph, creating what is called a **graph break**;
- The graph break can be created due to a lot of reasons such as: calling external libs such as numpy, converting tensors to Python types (e.g. `Tensor.tolist()`, `Tensor.item()`, etc);
- You can get the **reason** for each graph break and each graph break has obviously a performance penalty of switching back and forth between compiled code and Python code;
- TorchDynamo is used by `torch.compile()` but it is also exposed in the `torch_dynamo` module.

TorchDynamo

```
def my_fn(x):
    x = x * 2
    x = x.tolist()
    x += [1, 2]
    return x

def custom_backend(gm: torch.fx.GraphModule,
    example_inputs: List[torch.Tensor]):
    gm.graph.print_tabular()
    return gm.forward

opt_my_fn = torch.compile(my_fn, backend=custom_backend)
ret = opt_my_fn(torch.tensor([1., 2.]))
```

Note that we are explicitly calling the `Tensor.tolist()` where Torch will have to convert tensors into a Python `list` object.

TorchDynamo

Our `custom_backend` was called just once with the following captured graph:

```
opcode         name    target                   args        kwargs
-------------  ------  -----------------------  ----------  --------
placeholder    l_x_    L_x_                     ()          {}
call_function  mul     <built-in function mul>  (l_x_, 2)   {}
output         output  output                   ((mul,),)   {}
```

TorchDynamo

Our `custom_backend` was called just once with the following captured graph:

```
opcode         name    target                     args         kwargs
-------------  ------  -------------------------  -----------  --------
placeholder    l_x_    L_x_                        ()           {}
call_function  mul     <built-in function mul>    (l_x_, 2)    {}
output         output  output                     ((mul,),)    {}
```

This graph captures only the `x = x * 2` part of the code, because of the *graph break* introduced due to the `Tensor.tolist()` operation. TorchDynamo then delegates the execution of `x += [1, 2]` back to Python's default frame evaluation.

TorchDynamo

What happens if we modify our `my_fn` function to go back to a torch tensor and do a torch operation again ?

TorchDynamo

What happens if we modify our `my_fn` function to go back to a torch tensor and do a torch operation again ?

```
def my_fn(x):
    x = x * 2
    # To Python list
    x = x.tolist()
    x += [1, 2]
    # To torch tensor
    x = torch.tensor(x)
    x = x**2
    return x
```

TorchDynamo

```
opcode         name    target                    args
-------------  ------  ------------------------  ---------
placeholder    l_x_    L_x_                       ()
call_function  mul     <built-in function mul>   (l_x_, 2)
output         output  output                    ((mul,),)

opcode         name    target                     args
-------------  ------  -------------------------  -------------------
call_function  tensor  <built-in method tensor>   ([2.0, 4.0, 1, 2],)
call_function  pow_1   <built-in function pow>    (tensor, 2)
output         output  output                     ((pow_1,),)
```

Note that our `custom_backend` was called twice with different graphs representing the first part of computation and the second part of the computation, without the pure-Python operations on the Python `list`.

TorchDynamo

- So far, we haven't actually compiled any of the graphs that our `custom_backend` backend received. We have been focusing only in the graph acquisition problem.
- To get performance improvements, we need to equip `torch.compile()` with a compiler that will convert the acquired graphs into efficient native code for different target hardware such as NVIDIA GPUs, Arm CPUs, RISC-V CPUs, TPUs, exotic edge devices such as your smart toaster, among others.

TorchDynamo

- So far, we haven't actually compiled any of the graphs that our `custom_backend` backend received. We have been focusing only in the graph acquisition problem.
- To get performance improvements, we need to equip `torch.compile()` with a compiler that will convert the acquired graphs into efficient native code for different target hardware such as NVIDIA GPUs, Arm CPUs, RISC-V CPUs, TPUs, exotic edge devices such as your smart toaster, among others.

That's where **TorchInductor** comes into play.

Section IV

Inductor

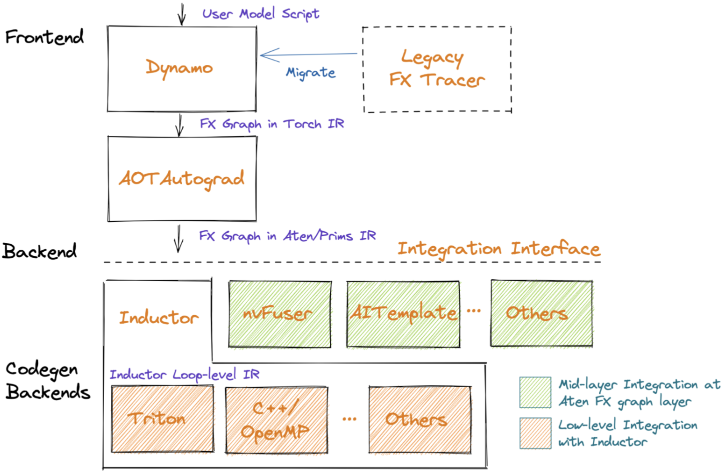
AOTAutograd

- TorchDynamo generates **Torch IR**, which is a high-level representation that is not suitable to many different compiler backends;
- If we want to speed-up training as well, we need to capture the backward pass as well, hence the need for the **AOTAutograd**, where AOT stands for ahead-of-time;
- The AOTAutograd will generate **ATen/Prims IR** from tracing the forward and backward graph ahead of time;

AOTAutograd

- TorchDynamo generates **Torch IR**, which is a high-level representation that is not suitable to many different compiler backends;
- If we want to speed-up training as well, we need to capture the backward pass as well, hence the need for the **AOTAutograd**, where AOT stands for ahead-of-time;
- The AOTAutograd will generate **ATen/Prims IR** from tracing the forward and backward graph ahead of time;
- IRs in PyTorch are a complex subject with many levels and many decompositions available;
- We will see an example of the difference between the graph generated by TorchDynamo vs the graph generated by AOTAutograd.

The big picture



Slide from "Deep Dive into TorchInductor and PT2 Backend Integration". Sherlock Huang et al.

Dynamo Torch IR

Let's take a look on the IR generated by TorchDynamo for the following model:

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(8, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = torch.nn.functional.softmax(x, -1)
        return x
```

Dynamo Torch IR

Let's use the `print_readable()` method to show the graph this time:

```
def custom_backend(gm: torch.fx.GraphModule,
                   example_inputs: list[torch.Tensor]):
    gm.print_readable()
    return gm.forward

model = MLP()
my_fn_opt = torch.compile(model,
                          backend=custom_backend)
input_tensor = torch.randn(10, 8)
ret = my_fn_opt(input_tensor)
```

Dynamo Torch IR

This will yield the following IR:

```
class GraphModule(torch.nn.Module):
  def forward(self, L_x_ : torch.Tensor):
    l_x_ = L_x_

    # code: x = self.fc1(x)
    l__self___fc1 = self.L__self___fc1(l_x_);
    l_x_ = None

    # code: x = torch.nn.functional.softmax(x, -1)
    softmax = torch.nn.functional.softmax(l__self___fc1, -1);
    l__self___fc1 = None
    return (softmax,)
```

AOTAutograd ATen IR

Let's now change the backend a bit to use AOTAutograd:

```
from torch._functorch.aot_autograd import \
    aot_module_simplified

def custom_backend(gm: torch.fx.GraphModule,
                   example_inputs: list[torch.Tensor]):
    def my_compiler(gm, example_inputs):
        gm.print_readable()
        return gm.forward

    return aot_module_simplified(
        gm,
        example_inputs,
        fw_compiler=my_compiler
    )
```

AOTAutograd ATen IR

And here we are with the AOTAutograd generated IR (with `= None`'s and some comments removed for brevity):

```
class GraphModule(torch.nn.Module):
  def forward(self, primals_1: f32[10, 8],
              primals_2: f32[10], primals_3: f32[10, 8]):
    # code: x = self.fc1(x)
    t: f32[8, 10] = torch.ops.aten.t.default(primals_1)
    addmm: f32[10, 10] = \
      torch.ops.aten.addmm.default(primals_2, primals_3, t)

    # code: x = torch.nn.functional.softmax(x, -1)
    _softmax: f32[10, 10] = \
      torch.ops.aten._softmax.default(addmm, -1, False)
    return [_softmax, primals_3, _softmax]
```

TorchInductor

Inductor takes the graph produced by AOTAutograd (consisting of ATen/Prim IR) and perform further graph decompositions:

```
def forward(self, arg0_1: f32[10, 8], arg1_1: f32[10],
            arg2_1: f32[10, 8]):
  # code: x = self.fc1(x)
  permute: f32[8, 10] = torch.ops.aten.permute.default(arg0_1, [1, 0])
  addmm: f32[1024, 10] = \
    torch.ops.aten.addmm.default(arg1_1, arg2_1, permute);

  # code: x = torch.nn.functional.softmax(x, -1)
  amax: f32[10, 1] = torch.ops.aten.amax.default(addmm, [-1], True)
  sub: f32[10, 10] = torch.ops.aten.sub.Tensor(addmm, amax)
  exp: f32[10, 10] = torch.ops.aten.exp.default(sub)
  sum_1: f32[10, 1] = torch.ops.aten.sum.dim_IntList(exp, [-1], True)
  div: f32[10, 10] = torch.ops.aten.div.Tensor(exp, sum_1)
  return (div,)
```

TorchInductor

- After that, the graph goes to the scheduling phase where fusion can happen and then to the appropriate TorchInductor backend;
- TorchInductor can generate C++/OpenMP code or Triton. The generated kernels are then called by a generated wrapper;
- Industry is collaborating with backend optimizations (e.g. Intel speedups for CPU `bfloat16` in some recent processors);

TorchInductor

- After that, the graph goes to the scheduling phase where fusion can happen and then to the appropriate TorchInductor backend;
- TorchInductor can generate C++/OpenMP code or Triton. The generated kernels are then called by a generated wrapper;
- Industry is collaborating with backend optimizations (e.g. Intel speedups for CPU `bfloat16` in some recent processors);

- We will see now a part of a C++ kernel generated by TorchInductor for the fused softmax with CPU tensors (*in MacOS as an example*).

TorchInductor

```
extern "C" void kernel(float* in_out_ptr0,
                       float* out_ptr0, float* out_ptr1)
{
  auto in_ptr0 = in_out_ptr0;
  {
    #pragma GCC ivdep
    for(long i0=static_cast<long>(0L); i0<static_cast<long>(10L);
        i0+=static_cast<long>(1L))
    {
      float tmp_acc0 = -std::numeric_limits<float>::infinity();
      for(long i1=static_cast<long>(0L); i1<static_cast<long>(10L);
          i1+=static_cast<long>(1L))
      {
        auto tmp0 = in_ptr0[static_cast<long>(i1 + (10L*i0))];
        tmp_acc0 = max_propagate_nan(tmp_acc0, tmp0);
      }
      out_ptr0[static_cast<long>(i0)] = tmp_acc0;
    }
  }
```

TorchInductor

Now, if we run the same code with **CUDA tensors**, what we will get is the Triton kernel below:

```
@triton.jit
def triton_(in_ptr0, out_ptr2, xnumel, rnumel, XBLOCK : tl.constexpr):
  # ... (omitted for brevity)
  tmp0 = tl.load(in_ptr0 + (r1 + (10*x0)), rmask & xmask, other=0)
  tmp1 = tl.broadcast_to(tmp0, [XBLOCK, RBLOCK])
  tmp3 = tl.where(rmask & xmask, tmp1, float("-inf"))
  tmp4 = triton_helpers.max2(tmp3, 1)[:, None]
  tmp5 = tmp0 - tmp4
  tmp6 = tl.exp(tmp5)
  tmp7 = tl.broadcast_to(tmp6, [XBLOCK, RBLOCK])
  tmp9 = tl.where(rmask & xmask, tmp7, 0)
  tmp10 = tl.sum(tmp9, 1)[:, None]
  tmp11 = tmp6 / tmp10
  tl.store(out_ptr2 + (r1 + (10*x0)), tmp11, rmask & xmask)
```

Section V

Torch Export

Torch Export Path

- Torch Export (`torch.export`) was created to do **whole-graph capture**;
- As we discussed earlier, TorchDynamo can create graph breaks and do this back-and-forth with the Python interpreter;
- This cooperative dynamic with Python makes it difficult to be able to embed it in environments without the Python runtime;

Torch Export Path

- Torch Export (`torch.export`) was created to do **whole-graph capture**;
- As we discussed earlier, TorchDynamo can create graph breaks and do this back-and-forth with the Python interpreter;
- This cooperative dynamic with Python makes it difficult to be able to embed it in environments without the Python runtime;
- `torch.export` relies on the `torch.compile` stack, but with important differences: it doesn't fallback to Python interpreter, so captured graph cannot have graph breaks and code changes can be required;
- The main goal of `torch.export` is to provide normalized IR using Core ATen IR opset that can be loaded and executed in different languages/environments.

Dynamo Torch IR

Let's use the same code we used earlier with TorchDynamo and export it with `torch.export`:

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(8, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = torch.nn.functional.softmax(x, -1)
        return x
```

Torch Export

```
>>> import torch.export as export
>>> model = MLP()
>>> sample = torch.randn(10, 8)
>>> exp = export.export(model, (sample,))
>>> exp
<torch.export.ExportedProgram object at 0x163c8ad10>
>>> print(exp)
```

Torch Export

```
>>> import torch.export as export
>>> model = MLP()
>>> sample = torch.randn(10, 8)
>>> exp = export.export(model, (sample,))
>>> exp
<torch.export.ExportedProgram object at 0x163c8ad10>
>>> print(exp)
```

```
class GraphModule(torch.nn.Module):
def forward(self, arg0_1: f32[10, 8], arg1_1: f32[10], arg2_1: f32[10, 8]):
    permute: f32[8, 10] = \
        torch.ops.aten.permute.default(arg0_1, [1, 0])
    addmm: f32[10, 10] = \
        torch.ops.aten.addmm.default(arg1_1, arg2_1, permute)
    _softmax: f32[10, 10] = \
        torch.ops.aten._softmax.default(addmm, -1, False)
    return (_softmax,)
(...)
```

Torch Export

Let's serialize the exported graph:

```
>>> export.save(exp, "serialized_graph.pt2")
```

Torch Export

Let's serialize the exported graph:

```
>>> export.save(exp, "serialized_graph.pt2")
```

We can see that the format is a zip archive:

```
$ file serialized_graph.pt2
serialized_graph.pt2: Zip archive data
```

Torch Export

Let's serialize the exported graph:

```
>>> export.save(exp, "serialized_graph.pt2")
```

We can see that the format is a zip archive:

```
$ file serialized_graph.pt2
serialized_graph.pt2: Zip archive data
```

... and we can extract to inspect:

```
$ unzip serialized_graph.pt2
extracting: serialized_exported_program.json
extracting: serialized_state_dict.json
extracting: version
```

Torch Export

There is a `version` file:

```
$ cat version
1
```

Torch Export

There is a `version` file:

```
$ cat version
1
```

A `serialized_exported_program.json`:

```
$ file serialized_exported_program.json
serialized_exported_program.json: JSON data
```

Torch Export

There is a `version` file:

```
$ cat version
1
```

A `serialized_exported_program.json`:

```
$ file serialized_exported_program.json
serialized_exported_program.json: JSON data
```

And the `serialized_state_dict.json`:

```
$ file serialized_state_dict.json
serialized_state_dict.json: Zip archive data
```

Not sure why PyTorch uses a `json` extension for a **Zip archive**.

Torch Export

```
$ jq "keys" serialized_exported_program.json
["equality_constraints",
"example_inputs",
"graph_module",
"opset_version",
"range_constraints",
"schema_version"]
```

Torch Export

```
$ jq "keys" serialized_exported_program.json
["equality_constraints",
"example_inputs",
"graph_module",
"opset_version",
"range_constraints",
"schema_version"]
```

The graph is in the `graph_module` and there is a `opset_version` with the used ATen IR opset version:

```
$ jq .opset_version serialized_exported_program.json
{
    "aten": 10
}
```

Torch Export

Let's see the nodes from the graph:

```
$ jq ".graph_module.graph.nodes[].target" (...)
"torch.ops.aten.permute.default"
"torch.ops.aten.addmm.default"
"torch.ops.aten._softmax.default"
```

Torch Export

Let's see the nodes from the graph:

```
$ jq ".graph_module.graph.nodes[].target" (...)
"torch.ops.aten.permute.default"
"torch.ops.aten.addmm.default"
"torch.ops.aten._softmax.default"
```

Let's see the outputs of the graph:

```
$ jq .graph_module.graph.outputs (...)
[{
    "as_none": null,
    "as_tensor": {
        "name": "_softmax"
    },
    "as_tensors": null,
    "as_int": null,
    "as_ints": null,
    "..."
}]
```

Torch Export

- You might need to rewrite your code if you use `torch.export`, especially if you have graph breaks and data/shape-dependent control flow as well;
- `torch.export` is, nevertheless, a very nice direction towards standardization of the IR. If vendors adopt it, you can skip intermediate representations (e.g. ONNX) and many nightmares;
- APIs, IRs opsets are very recent and subject to changes, so keep an eye on its development;

Torch Export

- You might need to rewrite your code if you use `torch.export`, especially if you have graph breaks and data/shape-dependent control flow as well;
- `torch.export` is, nevertheless, a very nice direction towards standardization of the IR. If vendors adopt it, you can skip intermediate representations (e.g. ONNX) and many nightmares;
- APIs, IRs opsets are very recent and subject to changes, so keep an eye on its development;

- We have now a serialized graph, let's now find out how we can actually execute it outside of Python. That's where **ExecuTorch** joins the party !

Section VI

ExecuTorch

ExecuTorch

- ExecuTorch (ET) leverages PyTorch 2 compiler and export path to enable **on-device** execution of PyTorch models;
- Portable runtime, low memory footprint and doesn't use TorchScript (as in PyTorch mobile);
- Still a lot of on-going development, this talk is aligned with the **v0.1.0** branch of ExecuTorch, *a preview release* for testing and evaluation;
- Multiple backends (arm, qualcomm, xnnpack, apple, etc) where ExecuTorch can delegate to DSPs, NPUs, CPUs, etc, being developed;
- Hope to see more industry collaboration.

ExecuTorch

Executorch has two main phases:

AOT (Ahead of Time)

This is the program preparation (before the execution). ExecuTorch leverages TorchDynamo and PyTorch export to convert the model into an IR. Optionally, backends can plug-in in this phase as well in what is called backend delegation for AOT.

ExecuTorch

Executorch has two main phases:

AOT (Ahead of Time)

This is the program preparation (before the execution). ExecuTorch leverages TorchDynamo and PyTorch export to convert the model into an IR. Optionally, backends can plug-in in this phase as well in what is called backend delegation for AOT.

Runtime

ExecuTorch runtime executes models on the edge devices (which can be a high-end or very constrained edge device). It will initialize, execute and release resources. It will also initialize delegates and (surprise) delegate execution of the program (or parts of it) to them as well.

ExecuTorch Concept Overview

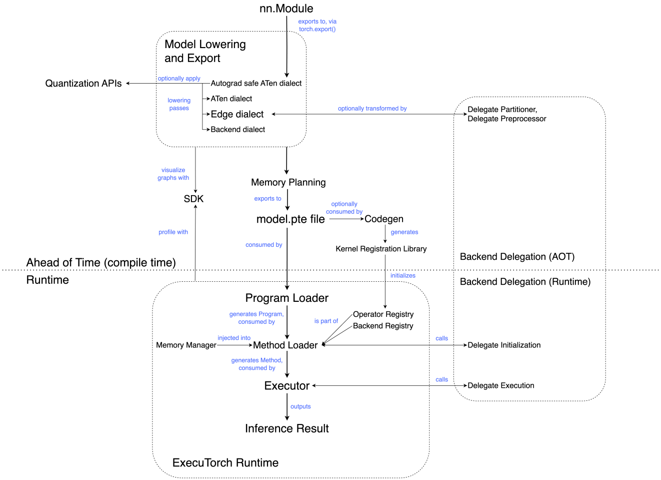


Image from ExecuTorch documentation, December 2023.

ExecuTorch Lowering

ExecuTorch performs progressive lowering of the graph or parts of the graph to different IRs, so the operations get progressively closer to the hardware:

- **Edge dialect**: all operators from predefined operator set and inputs/outputs must be tensor

ExecuTorch Lowering

ExecuTorch performs progressive lowering of the graph or parts of the graph to different IRs, so the operations get progressively closer to the hardware:

- **Edge dialect**: all operators from predefined operator set and inputs/outputs must be tensor
- **Backend dialect**: immediate result of exporting Edge dialect to a particular backend. Allows the introduction of target-specific operators (that are aware of the hardware they will run later)

ExecuTorch Memory Planning

Before serializing the program (`.pte` file), ExecuTorch performs **memory planning**. It uses size and lifespan of mutable tensors to plan their location (offset) in fixed size memory arenas:

ExecuTorch Memory Planning

Before serializing the program (`.pte` file), ExecuTorch performs **memory planning**. It uses size and lifespan of mutable tensors to plan their location (offset) in fixed size memory arenas:

Naive algorithm

Concatenates all the tensors together in a linear memory without considering any memory re-use.

ExecuTorch Memory Planning

Before serializing the program (`.pte` file), ExecuTorch performs **memory planning**. It uses size and lifespan of mutable tensors to plan their location (offset) in fixed size memory arenas:

Naive algorithm

Concatenates all the tensors together in a linear memory without considering any memory re-use.

Greedy algorithm

Tries to re-use the already allocated memory and choose based on the best-fit criteria.

ExecuTorch Memory Planning

Before serializing the program (`.pte` file), ExecuTorch performs **memory planning**. It uses size and lifespan of mutable tensors to plan their location (offset) in fixed size memory arenas:

Naive algorithm

Concatenates all the tensors together in a linear memory without considering any memory re-use.

Greedy algorithm

Tries to re-use the already allocated memory and choose based on the best-fit criteria.

```
program = edge_program.to_executorch( # Example
    exir.ExecutorchBackendConfig(
        memory_planning_pass=MemoryPlanningPass(
            memory_planning_algo="greedy",
            # (...)
)))
```

ExecuTorch Export

Let's export the same model that we had before:

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(8, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = torch.nn.functional.softmax(x, -1)
        return x
```

ExecuTorch Export

```
from torch._export import capture_pre_autograd_graph
from executorch.exir import to_edge

model = MLP()
model = model.eval()
inputs = (torch.randn(10, 8),)
pre_atgrad_aten_ir = capture_pre_autograd_graph(model,
                                                 inputs)
aten_ir = export.export(pre_atgrad_aten_ir, inputs)
edge_ir = to_edge(aten_ir)
program = edge_ir.to_executorch()

with open("model.pte", "wb") as fhandle:
    fhandle.write(program.buffer)
```

ExecuTorch Serialization

The serialization of the program uses the same memory efficient format used in TensorFlow Lite: **FlatBuffers**. The `Program` schema is defined in the `schema/program.fbs` file:

```
// (...) omitted for brevity

table Program {
    // Schema version.
    version:uint;

    // List of ExecutionPlans that make up the program.
    // Each ExecutionPlan corresponds with a different
    // entry point into the model.
    execution_plan:[ExecutionPlan];

    // (...) omitted for brevity
}
```

ExecuTorch Serialization

Let's see how our exported program looks like by converting the binary flatbuffer to json:

```
$ flatc --strict-json --raw-binary \
    -t executorch/schema/program.fbs -- ./model.pte
```

ExecuTorch Serialization

Let’s see how our exported program looks like by converting the binary flatbuffer to json:

```
$ flatc --strict-json --raw-binary \
    -t executorch/schema/program.fbs -- ./model.pte

$ jq ".execution_plan[0].name" model.json
"forward"
```

ExecuTorch Serialization

Let’s see how our exported program looks like by converting the binary flatbuffer to json:

```
$ flatc --strict-json --raw-binary \
    -t executorch/schema/program.fbs -- ./model.pte
```

```
$ jq ".execution_plan[0].name" model.json
"forward"
```

```
$ jq ".execution_plan[0].operators[].name" model.json
"aten::permute_copy"
"aten::addmm"
"aten::_softmax"
```

Memory Planning in Action

Let's see how one tensor looks like in the `Program`:

```
// (...)
"val_type": "Tensor",
"val": {
    "scalar_type": "FLOAT",
    "sizes": [10, 8],
    "dim_order": [0, 1],
    "allocation_info": {
        "memory_id": 1,
        "memory_offset": 800
    }
}
// (...)
```

Memory Planning in Action

Constant tensors (e.g. weights in a `Linear` layer) are handled differently than mutable tensors:

```
Result<void*> getTensorDataPtr(...) {
  if (s_tensor->constant_buffer_idx() > 0) {
    auto data =
        program->get_constant_buffer_data(
          s_tensor->constant_buffer_idx());
    return const_cast<void*>(data.get());
  }

  const executorch_flatbuffer::AllocationDetails* allocation_info =
      s_tensor->allocation_info();
  if (allocation_info != nullptr) {
    const uint32_t memory_id = allocation_info->memory_id() - 1;
    return allocator->get_offset_address(
        memory_id, allocation_info->memory_offset(), nbytes);
  }
  // (...)
}
```

ExecuTorch Concept Overview

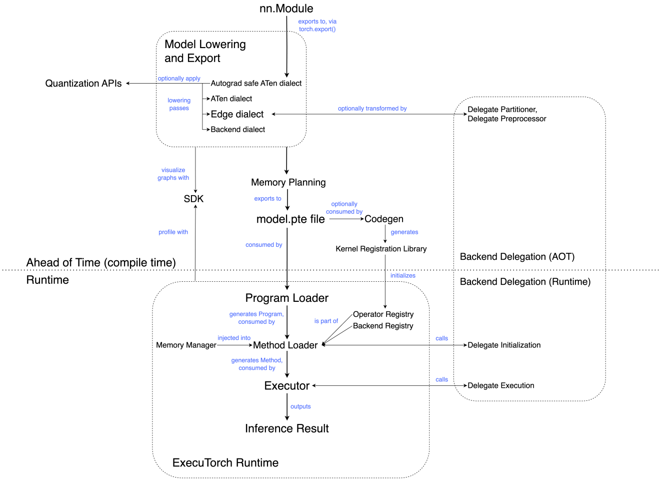


Image from ExecuTorch documentation, December 2023.

ExecuTorch Runtime

ExecuTorch runtime is a portable runtime:

- C++11 compatible, no exceptions or RTTI
- They provide cmake and buck2 build support
- Memory allocation mechanism is provided by the user, the core runtime doesn't do memory allocations (although backend kernels might, but disencouraged to do so)
- Can have different memory regions for mutable tensors (e.g. SRAM/DRAM placement)
- Without kernels or backend, runtime is 50kb

ExecuTorch Runtime

We have now the exported `Program` and want to load the `model.pte` and execute it on the edge.

- At this point, your next steps **will depend on the edge device** you want the runtime to run;
- There are many examples in ExecuTorch on how to deploy using XNNPACK, or targeting ARM (e.g. Ethos-U NPU), Qualcomm Hexagon NPU, DSPs, building Android/iOS apps, etc;

ExecuTorch Runtime

We have now the exported `Program` and want to load the `model.pte` and execute it on the edge.

- At this point, your next steps **will depend on the edge device** you want the runtime to run;
- There are many examples in ExecuTorch on how to deploy using XNNPACK, or targeting ARM (e.g. Ethos-U NPU), Qualcomm Hexagon NPU, DSPs, building Android/iOS apps, etc;
- For this tutorial, I will target a Pixel Watch 2 device (with a Cortex A53) and use the portable CPU kernels.

Loading the Program

Let's start looking at how we can use the runtime in C++ by first loading the serialized `Program`:

```
Result<FileDataLoader> loader = FileDataLoader::from(model_path);
Result<Program> program = Program::load(&loader.get());
Result<MethodMeta> method_meta = program->method_meta("forward");
```

Loading the Program

Let's start looking at how we can use the runtime in C++ by first loading the serialized `Program`:

```
Result<FileDataLoader> loader = FileDataLoader::from(model_path);
Result<Program> program = Program::load(&loader.get());
Result<MethodMeta> method_meta = program->method_meta("forward");
```

- The `.pte` file is opened
- File header is parsed
- Flatbuffer is created with serialized data

Memory Affair

Let's now create an allocator `method_allocator` for the method structure:

```
static uint8_t method_allocator_pool[4 * 1024U * 1024U];
MemoryAllocator method_allocator{
  MemoryAllocator(sizeof(method_allocator_pool),
                  method_allocator_pool)};
```

Memory Affair

Let's now create an allocator `method_allocator` for the method structure:

```
static uint8_t method_allocator_pool[4 * 1024U * 1024U];
MemoryAllocator method_allocator{
  MemoryAllocator(sizeof(method_allocator_pool),
                  method_allocator_pool)};
```

Most of this code is from `executor_runner.cpp` in ExecuTorch. Don't get too attached to the idiosyncrasies, but to what it is actually doing.

Memory Affair

Let's allocate now the planned buffers for the mutable tensors:

```
std::vector<std::unique_ptr<uint8_t[]>> buffers;
std::vector<Span<uint8_t>> spans;

size_t n_planned_buffers = \
  method_meta->num_memory_planned_buffers();

for (size_t id = 0; id < n_planned_buffers; ++id) {
  size_t buffer_size = \
    method_meta->memory_planned_buffer_size(id).get();
  buffers.push_back(std::make_unique<uint8_t[]>(buffer_size));
  spans.push_back({buffers.back().get(),
                   buffer_size});
}

HierarchicalAllocator planned_memory({buffers.data(), spans.size()});
MemoryManager memory_manager(&method_allocator, &planned_memory);
```

Memory Affair

We can now finally execute the method:

```
Result<Method> method = \
  program->load_method("forward", &memory_manager);
method.set_input(...) // set the method inputs
Error status = method->execute();

// Get the outputs into "outputs"
std::vector<EValue> outputs(method->outputs_size());
status = method->get_outputs(outputs.data(), outputs.size());
```

Our victim today

- Google Pixel Watch 2
- Qualcomm SW5100, 4x Cortex A53 cores
- 2GB of RAM
- Android Wear OS 4

Our victim today

- Google Pixel Watch 2
- Qualcomm SW5100, 4x Cortex A53 cores
- 2GB of RAM
- Android Wear OS 4

- *I'm not affiliated with Google, this happened to be the first small device in front of me. I'm planning to experiment with a more constrained RP2040 (Raspberry Pi Pico, Cortex-M0+) next time.*

Which CPU is that

Pixel Watch 2 runs Android, let's see the architecture:

```
$ uname -a
Linux localhost 5.15.104-android13-(...) armv8l Toybox
```

Interestingly this SoC supports armv8 64-bits, but it is running on 32-bits with the kernel compiled for armv8l (32-bits, little ending).

Which CPU is that

Pixel Watch 2 runs Android, let's see the architecture:

```
$ uname -a
Linux localhost 5.15.104-android13-(...) armv8l Toybox
```

Interestingly this SoC supports armv8 64-bits, but it is running on 32-bits with the kernel compiled for armv8l (32-bits, little ending).

```
$ cat /proc/cpuinfo
processor : 0
model name : ARMv8 Processor rev 4 (v8l)
Features : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4
           idiva idivt lpae evtstrm aes pmull sha1 sha2 crc32
CPU implementer : 0x51
CPU architecture: 8
CPU variant : 0xa
CPU part : 0x801
CPU revision : 4
(...)
```

Toolchains Everywhere

Let's prepare to use the Android toolchain for cross-compilation:

Download the Android NDK and set its path:

```
$ export ANDROID_NDK=/opt/android-ndk-r26b
```

Toolchains Everywhere

Let's prepare to use the Android toolchain for cross-compilation:

Download the Android NDK and set its path:

```
$ export ANDROID_NDK=/opt/android-ndk-r26b
```

Then we just add some variables into `CMakeLists.txt` in ExecuTorch:

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_SYSTEM_VERSION 24)
set(CMAKE_ANDROID_ARCH_ABI armeabi-v7a)
```

I only found the compatible `armeabi-v7a` architecture in Android NDK, since armv8l is backwards compatible with ARMv7, I'm using this one.

Selective Build

There are many ways of building our application and linking to ExecuTorch, what we will use is the **selective build**, which will select only a few kernels to be compiled and we will use **MobileNetV2**.

Selective Build

There are many ways of building our application and linking to ExecuTorch, what we will use is the **selective build**, which will select only a few kernels to be compiled and we will use **MobileNetV2**.

Luckily, ExecuTorch has some scripts to help with exporting the model and compiling. Let's export MobileNetV2 (`mv2`):

```
$ python3 -m examples.portable.scripts.export --model_name="mv2"
```

This will create the serialized program `mv2.pte`.

Selective Build

There are many ways of building our application and linking to ExecuTorch, what we will use is the **selective build**, which will select only a few kernels to be compiled and we will use **MobileNetV2**.

Luckily, ExecuTorch has some scripts to help with exporting the model and compiling. Let's export MobileNetV2 (`mv2`):

```
$ python3 -m examples.portable.scripts.export --model_name="mv2"
```

This will create the serialized program `mv2.pte`.

Now we can compile it with `cmake`:

```
$ examples/selective_build/test_selective_build.sh cmake
```

Selective Build

You can look at the `test_selective_build.sh` but the important bit here is the selected ops list we are building in our application:

```
$ cmake (...) -DEXECUTORCH_SELECT_OPS_LIST="aten::convolution.out,\
  (...) aten::mean.out,aten::view_copy.out,aten::permute_copy.out,\
  aten::addmm.out,aten,aten::clone.out"
```

Instead of building all kernels, we are selecting only a few of them. This is very important for more constrained devices.

Selective Build

You can look at the `test_selective_build.sh` but the important bit here is the selected ops list we are building in our application:

```
$ cmake (...) -DEXECUTORCH_SELECT_OPS_LIST="aten::convolution.out,\
  (...) aten::mean.out,aten::view_copy.out,aten::permute_copy.out,\
  aten::addmm.out,aten,aten::clone.out"
```

Instead of building all kernels, we are selecting only a few of them. This is very important for more constrained devices.

We just copy our binary `model_app` and the exported model `mv2.pte` to the Pixel Watch 2 using Android `adb` tool and then run the model:

```
$ model_app --model_path="mv2.pte"
```

Selective Build

The output of executing the example app in the Pixel Watch 2 will be something like this:

```
Output 0: tensor(sizes=[1, 1000], [
  -0.50986, 0.300638, 0.0953863, 0.147721, 0.231201, 0.338555,
  0.20689, -0.0575741, -0.389267, -0.0606858, -0.0213996,
  -0.121034, -0.288955, 0.134052, -0.171977, -0.060362,
  0.0203591, -0.0585306, 0.337859, -0.0718654, 0.490758,
  0.524143, 0.197859, 0.122067, -0.35913, 0.10946, 0.347745,
  0.478512, 0.226557, 0.0363519,
(...)
```

Showing the 1000 class logits for the input (all 1's in our case).

Thanks !

I hope you enjoyed this presentation ! This was an overview of the internals of some of the projects in the PyTorch ecosystem that came out recently. I skipped some other important aspects such as distributed training, but hopefully it will come soon in the next iteration of this presentation.

Huge thanks to all PyTorch contributors !

Section VII

~ Q&A ~

Q&A

Thanks !