PyTorch under the hood

A guide to understand PyTorch internals

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

PyData Montreal, Feb 2019

Agenda

Who Am I

- **Christian S. Perone**
- 14 years working with Machine Learning, Data Science and Software Engineering in industry R&D
- Blog at
- `blog.christianperone.com`
- Open-source projects at
- `https://github.com/perone`
- Twitter @tarantulae

DISCLAIMER

- PyTorch is a **moving target**, Deep Learning ecosystem moves fast and big changes happens every week;

Disclaimer

- PyTorch is a **moving target**, Deep Learning ecosystem moves fast and big changes happens every week;
- This is not a talk to teach you the basics of PyTorch or how to train your network, but to teach you how **PyTorch components works under the hood** in a intuitive way;

DISCLAIMER

- PyTorch is a **moving target**, Deep Learning ecosystem moves fast and big changes happens every week;
- This is not a talk to teach you the basics of PyTorch or how to train your network, but to teach you how **PyTorch components works under the hood** in a intuitive way;
- This talk is updated to the PyTorch v.1.0.1 version;

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;
- In the Python API, PyTorch previously had separate `Variable` and a `Tensor` types, after v.0.4.0 they were merged into `Tensor`.

QUICK RECAP PYTHON OBJECTS

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

QUICK RECAP PYTHON OBJECTS

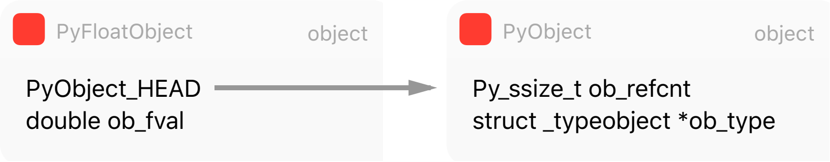
```c
typedef struct {
  PyObject_HEAD
  double ob_fval;
} PyFloatObject;
```

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

Quick recap Python objects

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

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



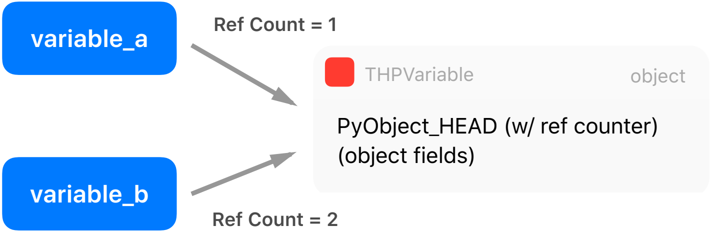
QUICK RECAP PYTHON OBJECTS

```
struct THPVariable {
        PyObject_HEAD
        torch::autograd::Variable cdata;
        PyObject* backward_hooks;
};
```

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

Quick recap Python objects

```
struct THPVariable {
        PyObject_HEAD
        torch::autograd::Variable cdata;
        PyObject* backward_hooks;
};
```



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

IN PYTHON, EVERYTHING IS AN OBJECT

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

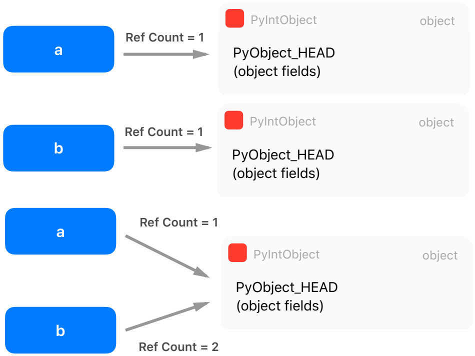
IN PYTHON, EVERYTHING IS AN OBJECT

```
>>> 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

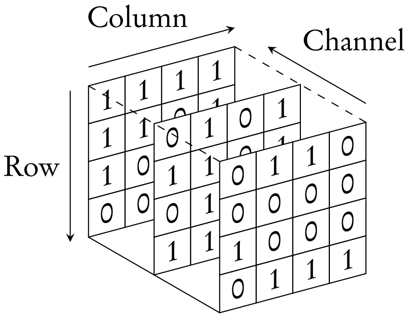
It is very common to load tensors in **numpy** and convert them to PyTorch, or vice-versa;

```
>>> 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 $\sim$ **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)
```

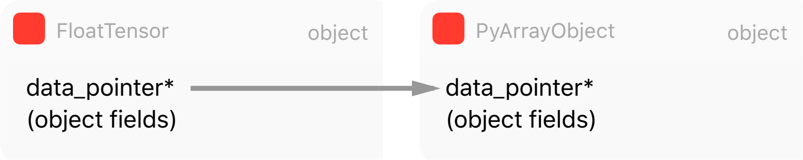
However, if you use `np_array += 1.0`, that is an in-place operation that will change `torch_array` memory.

Zero-copying tensors

```
at::Tensor tensor_from_numpy(PyObject* obj) {
    // (...) - omitted 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));
    // (...) - omitted for brevity
    void* data_ptr = PyArray_DATA(array);
    auto& type = CPU(dtype_to_aten(PyArray_TYPE(array)));
    Py_INCREF(obj);
    return type.tensorFromBlob(data_ptr, sizes, strides,
                               [obj](void* data) {
        AutoGIL gil;
        Py_DECREF(obj);
    });
}
```

Pay attention to the reference counting using `Py_INCREF()` and the call to `tensorFromBlob()` 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 final : (...) {
// (...)
private:
    // (...)
    caffe2::TypeMeta data_type_;
    DataPtr data_ptr_;
    int64_t numel_;
    Allocator* allocator_;
}
```

TENSOR STORAGE

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

```
struct C10_API StorageImpl final : (...) {
// (...)
private:
    // (...)
    caffe2::TypeMeta data_type_;
    DataPtr data_ptr_;
    int64_t numel_;
    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:

```
>>> tensor_a = torch.ones((2, 2))
>>> tensor_b = tensor_a.view(4)
>>> tensor_a_data = tensor_a.storage().data_ptr()
>>> tensor_b_data = tensor_b.storage().data_ptr()
>>> tensor_a_data == tensor_b_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:

```
>>> tensor_a = torch.ones((2, 2))
>>> tensor_b = tensor_a.view(4)
>>> tensor_a_data = tensor_a.storage().data_ptr()
>>> tensor_b_data = tensor_b.storage().data_ptr()
>>> tensor_a_data == tensor_b_data
True
```

- `tensor_b` 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 Allocator {
    virtual ~Allocator() {}
    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 `cudaMallocHost()` when the storage should be used for the GPU or `posix_memalign()` POSIX functions for data in the CPU memory.

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

```
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

```python
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)) {
%4 : Float() = prim::Constant[value={2}]()
%5 : Device = prim::Constant[value="cpu"]()
%6 : int = prim::Constant[value=6]()
%7 : bool = prim::Constant[value=0]()
%8 : bool = prim::Constant[value=0]()
%9 : Float() = aten::to(%4, %5, %6, %7, %8)
%10 : Float() = aten::detach(%9)
return (%10); }
```

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`:

```python
@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 : Tensor) {
%2 : float = prim::Constant[value=1]()
%5 : int = prim::Constant[value=1]()
%6 : int = prim::Constant[value=2]()
%1 : Tensor = aten::mean(%x)
%3 : Tensor = aten::gt(%1, %2)
%4 : bool = prim::Bool(%3)
%r : int = prim::If(%4)
  block0() {
    -> (%5)
  }
  block1() {
    -> (%6)
  }
  return (%r);
}
```

SCRIPTING

The `my_function()` is now a `ScriptModule`:

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

Scripting

The `my_function()` is now a `ScriptModule`:

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

When we check the results again:

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

SCRIPTING

The `my_function()` is now a `ScriptModule`:

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

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.

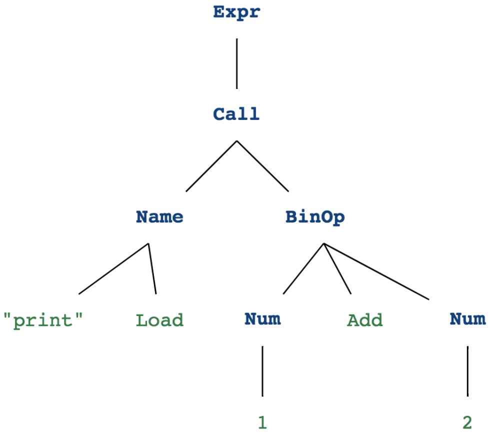
BUILDING THE IR

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.

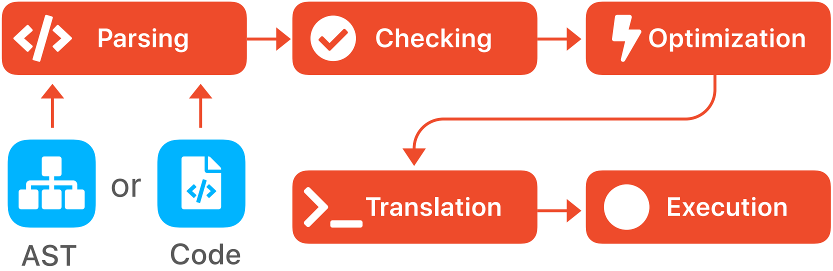
```
>>> ast_mod = ast.parse("print(1 + 2)")
>>> astpretty.pprint(ast_mod.body[0], show_offsets=False)

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

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



PYTORCH JIT PHASES



EXECUTING

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

```
bool runImpl(Stack& stack) {
    auto& instructions = function->instructions;
    size_t last = instructions.size();

    while (pc < last) {
      auto& inst = instructions[pc];
      try {
        loadTensorsFromRegisters(inst.inputs, stack);
        size_t new_pc = pc + 1 + inst.callback(stack);
        for (int i = inst.outputs.size - 1; i >= 0; --i) {
          int reg = get(inst.outputs, i);
          registers[reg] = pop(stack);
        }
        pc = new_pc;

        // (...) omitted
```

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:

```python
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 : Float(*, *)) {
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 : Float(*, *)) {
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/resnet.py
extracting: resnet/model.json
extracting: resnet/tensors/0
(...)
```

Serialization

code/resnet.py

```
op_version_set = 0
def forward(self, input_1: Tensor) -> Tensor:
    input_2 = torch._convolution(input_1, self.conv1.weight, ...)
    # (...)
    input_3 = torch.batch_norm(input_2, self.bn1.weight, self.bn1.bias,
        self.bn1.running_mean, self.bn1.running_var, ...)
    # (...)
```

SERIALIZATION

code/resnet.py

```
op_version_set = 0
def forward(self, input_1: Tensor) -> Tensor:
    input_2 = torch._convolution(input_1, self.conv1.weight, ...)
    # (...)
    input_3 = torch.batch_norm(input_2, self.bn1.weight, self.bn1.bias,
        self.bn1.running_mean, self.bn1.running_var, ...)
    # (...)
```

model.json

```
{"parameters":
[{ "isBuffer": false,
"tensorId": "1",
"name": "weight" }],
"name": "conv1",
"optimize": true}
```

SERIALIZATION

code/resnet.py

```
op_version_set = 0
def forward(self, input_1: Tensor) -> Tensor:
    input_2 = torch._convolution(input_1, self.conv1.weight, ...)
    # (...)
    input_3 = torch.batch_norm(input_2, self.bn1.weight, self.bn1.bias,
        self.bn1.running_mean, self.bn1.running_var, ...)
    # (...)
```

model.json

```
{"parameters":
[{ "isBuffer": false,
"tensorId": "1",
"name": "weight" }],
"name": "conv1",
"optimize": true}
```

model.json

```
[{"isBuffer": true,
"tensorId": "4",
"name": "running_mean"},
{"isBuffer": true,
"tensorId": "5",
"name": "running_var"}],
"name": "bn1",
"optimize": true}
```

USING THE MODEL IN C++

PyTorch also has a C++ API that you can use to load/train models in C++. This is good for production, mobile, embedded devices, etc.

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();
}
```

Using the model in NodeJS

```
> var torchjs = require("torchjs");
> var script_module = new torchjs.ScriptModule("resnet18_trace.pt");
> var data = torchjs.ones([1, 3, 224, 224], false);
> console.log(data);
Tensor[Type=Variable[CPUFloatType], Size=[1, 3, 224, 224]

> var output = script_module.forward(data);
> console.log(output);
Tensor[Type=Variable[CPUFloatType], Size=[1, 1000]
```

Complete tutorial at https://goo.gl/7wMJuS.

Section III

Production

ISSUES WITH TUTORIALS

- Be careful with online tutorials using Flask, etc. They are simple, but they often fail on good practices:
  - They often use JSON and base64 to serialize images. This adds $\sim$ 33% overhead **per call** (uncompressed);

Issues with tutorials

- Be careful with online tutorials using Flask, etc. They are simple, but they often fail on good practices:
  - They often use JSON and base64 to serialize images. This adds $\sim$ 33% overhead **per call** (uncompressed);
  - They don't pay attention to zero-copy practices, so they often transform, reshape, convert to numpy, convert to PyTorch, etc;

Issues with tutorials

- Be careful with online tutorials using Flask, etc. They are simple, but they often fail on good practices:
  - They often use JSON and base64 to serialize images. This adds $\sim$ 33% overhead **per call** (uncompressed);
  - They don't pay attention to zero-copy practices, so they often transform, reshape, convert to numpy, convert to PyTorch, etc;
  - They often use HTTP/1;

ISSUES WITH TUTORIALS

- Be careful with online tutorials using Flask, etc. They are simple, but they often fail on good practices:
  - They often use JSON and base64 to serialize images. This adds ~ 33% overhead **per call** (uncompressed);
  - They don't pay attention to zero-copy practices, so they often transform, reshape, convert to numpy, convert to PyTorch, etc;
  - They often use HTTP/1;
  - They seldom do batching (important for GPUs);

ISSUES WITH TUTORIALS

- Be careful with online tutorials using Flask, etc. They are simple, but they often fail on good practices:
  - They often use JSON and base64 to serialize images. This adds $\sim$ 33% overhead **per call** (uncompressed);
  - They don't pay attention to zero-copy practices, so they often transform, reshape, convert to numpy, convert to PyTorch, etc;
  - They often use HTTP/1;
  - They seldom do batching (important for GPUs);
  - They never put that "production" code in production.

Prefer binary serialization formats

Prefer using good **binary serialization** methods such as Protobuf that offers a **schema** and a schema evolution mechanism.

Example from EuclidesDB RPC message:

```
message AddImageRequest {
  int32 image_id = 1;
  bytes image_data = 2;
  // This field can encode JSON data
  bytes image_metadata = 3;
  repeated string models = 4;
}
```

* http://euclidesdb.readthedocs.io

Avoid extra copies

- Be careful to avoid extra copies of your tensors, especially during pre-processing;

Avoid extra copies

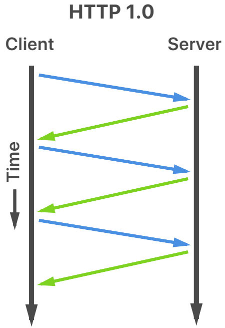
- Be careful to avoid extra copies of your tensors, especially during pre-processing;
- You can use in-place operations. It is a functional anti-pattern because it introduces side-effects, but it's a fair price to pay for performance;

Avoid extra copies

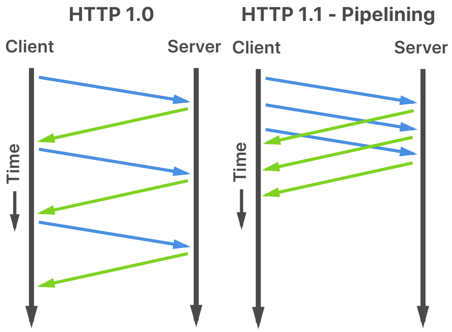
- Be careful to avoid extra copies of your tensors, especially during pre-processing;
- You can use in-place operations. It is a functional anti-pattern because it introduces side-effects, but it's a fair price to pay for performance;
- **Caveat**: in-place operations doesn't make much sense when you need gradients. PyTorch uses tensor versioning to catch that:

```
>>> a = torch.tensor(1.0, requires_grad=True)
>>> y = a.tanh()
>>> y.add_(2.0)
>>> y.backward() # error !
>>> a._version
0
>>> y._version
1
```

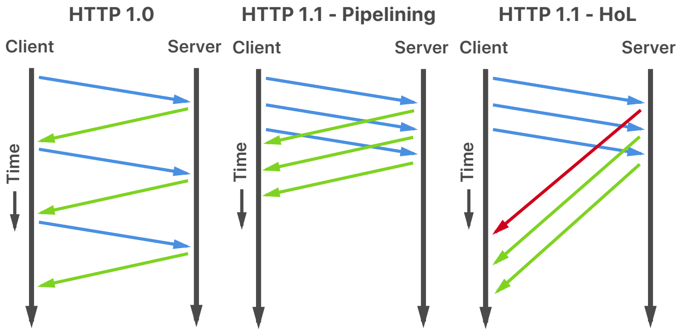
A tale of two HTTPs



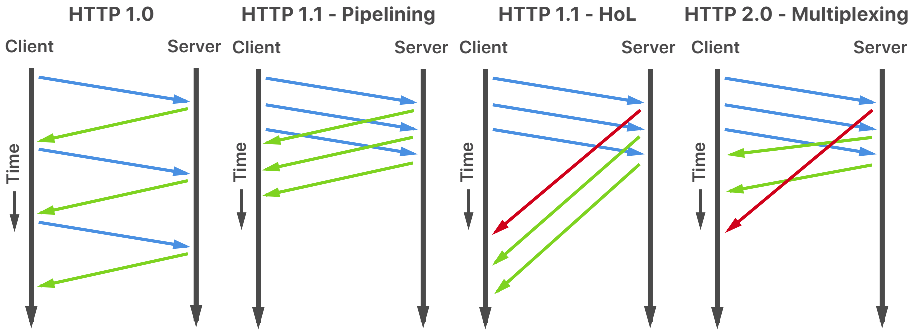
A tale of two HTTPs



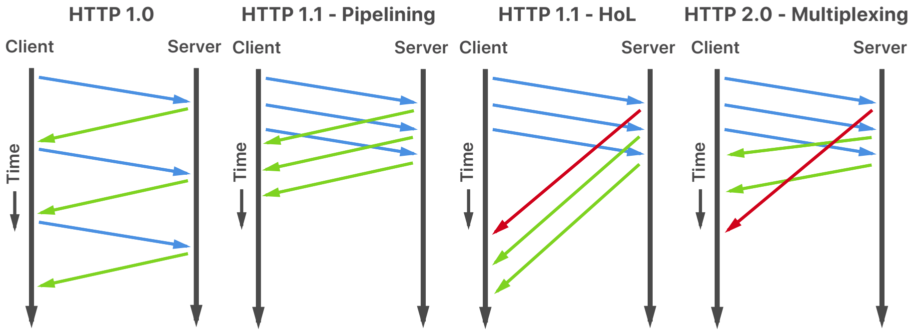
A tale of two HTTPs



A tale of two HTTPs

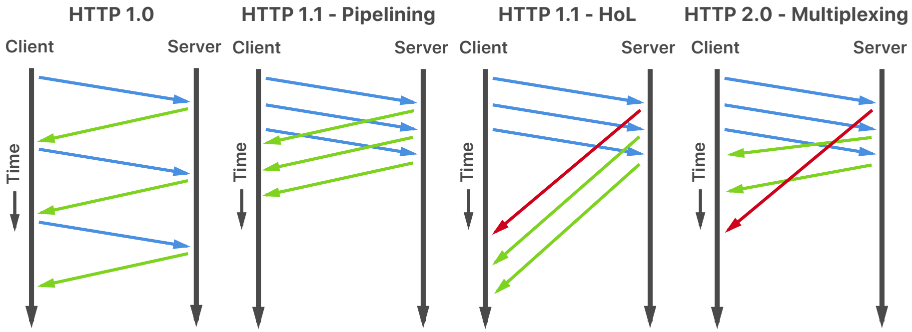


A tale of two HTTPs



- Use HTTP 2.0 if possible, and avoid the *head-of-line blocking*;

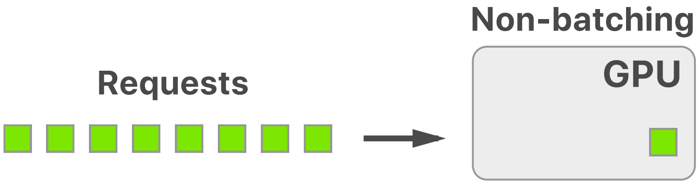
A tale of two HTTPs



- Use HTTP 2.0 if possible, and avoid the *head-of-line blocking*;
- Even better, you can use frameworks such as gRPC that uses HTTP/2.0 and Protobuf.

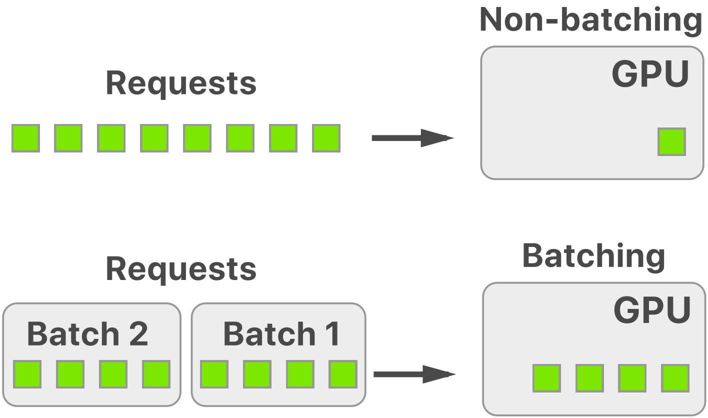
Batching

Batching data is a way to amortize the performance bottleneck.



BATCHING

Batching data is a way to amortize the performance bottleneck.



Section IV

Q&A

Q&A

Thanks !