

PyTorch 内核机制

PyTorch 的构建者表明，PyTorch 的哲学是解决当务之急，也就是说即时构建和运行我们的计算图。这恰好适合 Python 的编程理念，一边定义就可以在 Jupyter Notebook 一边运行，因此，PyTorch 的工作流程非常接近于 Python 的科学计算库 NumPy。

Christian 表明 PyTorch 之所以这么方便，很多都是因为它的「基因」——内部运行机制决定的。这一篇报告并不会介绍如何使用 PyTorch 基础模块，或如何用 PyTorch 训练一个神经网络，Christian 关注的是如何以直观的形式介绍 PyTorch 的内核机制，即各个模块到底是怎么工作的。

Christian 在 Reddit 表示这一次报告由于录像问题并不能上传演讲视频，因此暂时只能分享演讲 PPT。不过 Christian 最近也会再做一次该主题的演讲，所以我们可以期待下次能有介绍 PyTorch 的视频。

- 演讲 PPT 地址: <https://speakerdeck.com/perone/pytorch-under-the-hood>
- 百度云地址: <https://pan.baidu.com/s/1aaE0l1geF7VwEnQRwmzBtA>

如下所示为这次演讲的主要议程，它主要从张量和 JIT 编译器出发介绍底层运行机制：

Agenda

```
TENSORS
  Tensors
  Python objects
  Zero-copy
  Tensor storage
  Memory allocators (CPU/GPU)
  The big picture
JIT
  Just-in-time compiler
  Tracing
  Scripting
  Why TorchScript?
  Building IR and JIT Phases
  Optimizations
  Serialization
  Using models in other languages
PRODUCTION
  Some tips
Q&A
```

在讨论 PyTorch 的各组件机制前，我们需要了解整体工作流。PyTorch 使用一种称之为 **imperative / eager** 的范式，即每一行代码都要求构建一个图以定义完整计算图的一个部分。即使完整的计算图还没有完成构建，我们也可以独立地执行这些作为组件的小计算图，这种动态计算图被称为「**define-by-run**」方法。

其实初学者了解到整体流程就可以学着使用了，但底层机制有助于对代码的理解和掌控。

张量

在概念上，张量就是向量和矩阵的推广，PyTorch 中的张量就是元素为同一数据类型多维矩阵。虽然 PyTorch 的接口是 Python，但底层主要都是用 C++实现的，而在 Python 中，集成 C++代码通常被称为「扩展」。

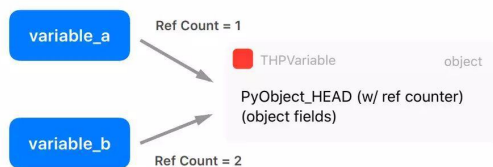
因为张量主要承载数据，并进行计算。PyTorch 的张量计算使用最底层和基本的张量运算库 ATen，它的自动微分使用 Autograd，该自动微分工具同样建立在 ATen 框架上。

Python 对象

为了定义 C/C++中一个新的 Python 对象类型，你需要定义如下 THPVariable 类似结构。其中第一个 PyObject_HEAD 宏旨在标准化 Python 对象，并扩展至另一个结构，该结构包含一个指向类型对象的指针，以及一个带有引用计数（ref count）的字段。

QUICK RECAP PYTHON OBJECTS

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

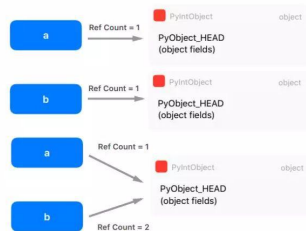


Python API 中有两个额外的宏，分别称为 Py_INCREF() 和 Py_DECREF(), 可用于增加和减少 Python 对象的引用计数。

在 Python 中，任何东西都是对象，例如变量、数据结构和函数等。

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 张量

由于 Numpy 数组的使用非常普遍，我们确实需要在 Numpy 和 PyTorch 张量之间做转换。因此 PyTorch 给出了 `from_numpy()` 和 `numpy()` 两个方法，从而在 NumPy 数组和 PyTorch 张量之间做转换。



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.]])
```

因为张量储存的成本比较大，如果我们在上述转换的过程中复制一遍数据，那么内存的占用会非常大。PyTorch 张量的一个优势是它会保留一个指向内部 NumPy 数组的指针，而不是直接复制它。这意味着 PyTorch 将拥有这一数据，并与 NumPy 数组对象共享同一内存区域。

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 的形式确实能省很多内存，但是如上所示在位（in-place）和标准运算之间的区别会有点模糊。如果用 `np_array = np_array + 1.0`, `torch_array` 的内存不会改变，但是如果用 `np_array += 1.0`, `torch_array` 的内存却又会改变。

CPU/GPU 内存分配

张量的实际原始数据并不是立即保存在张量结构中，而是保存在我们称之为「存储（Storage）」的地方，它是张量结构的一部分。一般张量存储可以通过 `Allocator` 选择是储存在计算机内存（CPU）还是显存（GPU）。

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:

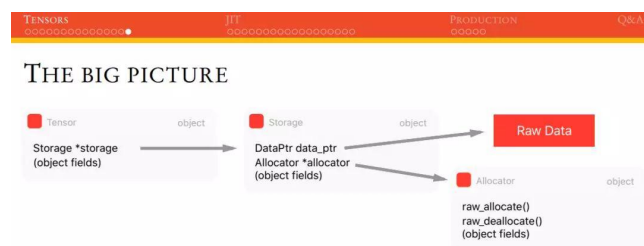
```
struct Allocator {  
    virtual ~Allocator() {}  
    virtual DataPtr allocate(size_t n) const = 0;  
    virtual DeleterFnPtr raw_deleter() const {...}  
    void* raw_allocate(size_t n) {...}  
    void raw_deallocate(void* ptr) {...}  
};
```

- 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.

PyTorch under the hood - Christian S. Ponne (2019)

THE BIG PICTURE

最后，PyTorch 主张量 `THTensor` 结构可以展示为下图。`THTensor` 的主要结构为张量数据，它保留了 `size/strides/dimensions/offsets/`等信息，同时还有存储 `THStorage`。



- 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.

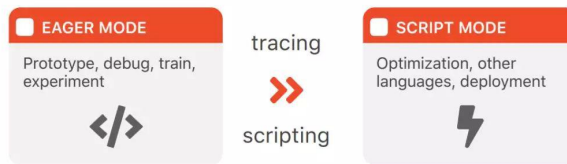
JIT

因为 PyTorch 是即时运行模式，这表明它很容易 `Debug` 或检查代码等。在 PyTorch 1.0 中，其首次引进了 `torch.jit`，它是一组编译工具，且主要目标是弥补研究与产品部署的差距。JIT 包含一种名为 `Torch Script` 的语言，这种语言是 Python 的子语言。使用 `Torch Script` 的代码可以实现非常大的优化，并且可以序列化以供在后续的 `C++API` 中使用。

如下所示为常见使用 Python 运行的 `Eager` 模式，也可以运行 `Script` 模式。`Eager` 模式适合块做原型与实验，而 `Script` 模式适合做优化与部署。

JIT - JUST-IN-TIME COMPILER

Two very different worlds with their own requirements.



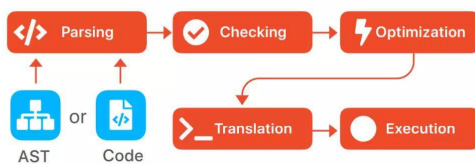
那么为什么要用 TORCHSCRIPT 呢？Christian 给出了以下理由：

- ▶ 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 (computational 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.

PyTorch JIT 主要过程

如下所示 JIT 主要会输入代码或 Python 的抽象句法树（AST），其中 AST 会用树结构表征 Python 源代码的句法结构。解析可能是解析句法结构和计算图，然后语法检测紧接着代码优化过程，最后只要编译并执行就可以了。

PYTORCH JIT PHASES



其中优化可以用于模型计算图，例如展开循环等。在如下所示的 **Peephole** 优化中，编译器仅在一个或多个基本块中针对已生成的代码，结合 CPU 指令的特点和一些转换规则提升性能。**Peephole** 优化也可以通过整体分析和指令转换提升代码性能。

如下所示矩阵的两次装置等于矩阵本身，这应该是需要优化的。

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.

PyTorch under the hood - Christian S. Ponce (EPFL)

执行

和 Python 解释器可以执行代码一样，PyTorch 在 JIT 过程中也有一个解释器执行中间表征指令：

```
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
    }
}
```

最后，Christian 还介绍了很多内部运行机制，不过因为它们都很难，而且暂时没有提供视频讲解，读者大牛们可以看看具体 PPT 内容。