深度学习的编程模式

Heading 1

参考：http://www.csdn.net/article/2015-10-11/2825883?reload=1
http://www.oschina.net/news/41789/imperative-vs-declarative

符号式编程 vs 命令式编程

在这一节，我们先来比较符号式程序（symbolic style programs）和命令式程序（imperative style programs）两种形式。如果你是一名 Python 或者 C++程序员，那你应该很熟悉命令式程序了。命令式程序按照我们的命令来执行运算过程。大多数 Python 代码都属于命令式，例如下面这段 numpy 的计算。

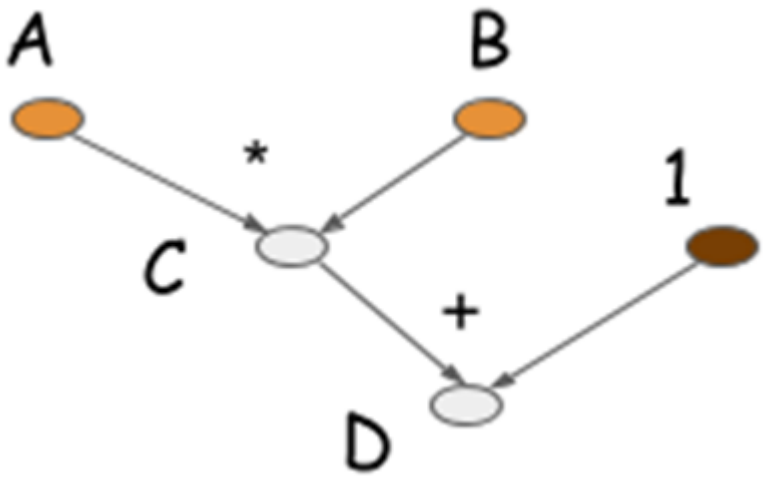
```
import numpy as np

a = np.ones(10)

b = np.ones(10) * 2

c = b * a

d = c + 1
```

当程序执行到 c = b * a 这一行时，机器确实做了一次乘法运算。符号式程序略有不同。下面这段代码属于符号式程序，它同样能够计算得到 d 的值。

```
A = Variable('A')

B = Variable('B')

C = B * A

D = C + Constant(1)

# compiles the function
```

```
f = compile(D)

d = f(A=np.ones(10), B=np.ones(10)*2)
```

符号式程序的不同之处在于，当执行 C = B * A 这一行代码时，程序并没有产生真正的计算，而是生成了一张计算图/符号图（computation graph/symbolic graph）来描述整个计算过程。下图就是计算得到 D 的计算图。



大多数符号式程序都会显式地或是隐式地包含编译步骤。这一步将计算图转换为能被调用的函数。在代码的最后一行才真正地进行了运算。符号式程序的最大特点就是清晰地将定义运算图的步骤与编译运算的步骤分割开来。

采用命令式编程的深度学习库包括 Torch，Chainer， Minerva。采用符号式编程的库有 Theano 和 CGT。一些使用配置文件的库，例如 cxxnet 和 Caffe，也都被视为是符号式编程。因为配置文件的内容定义了计算图。

现在你明白两种编程模型了吧，我们接着来比较它们！

命令式程序更加灵活

这并不能算是一种严格的表述，只能说大多数情况下命令式程序比符号式程序更灵活。如果你想用 Python 写一段命令式程序的代码，直接写就是了。但是，你若想写一段符号式程序的代码，则完全不同了。看下面这段命令式程序，想想你会怎样把它转化为符号式程序呢。

```
a = 2

b = a + 1
```

```
d = np.zeros(10)

for i in range(d):

    d += np.zeros(10)
```

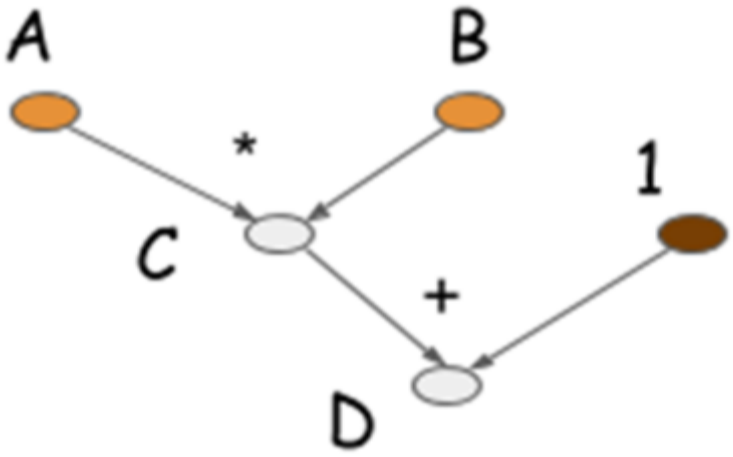
你会发现事实上并不容易，因为 Python 的 for 循环可能并不被符号式程序的 API 所支持。你若用 Python 来写符号式程序的代码，那绝对不是真的 Python 代码。实际上，你写的是符号式 API 定义的领域特定语言（DSL）。符号式 API 是 DSL 的加强版，能够生成计算图或是神经网络的配置。照此说法，输入配置文件的库都属于符号式的。

由于命令式程序比符号式程序更本地化，因此更容易利用语言本身的特性并将它们穿插在计算流程中。例如打印输出计算过程的中间值，或者使用宿主语言的条件判断和循环属性。

符号式程序更高效

我们在上一节讨论中提到，命令式程序更灵活，对宿主语言的本地化也更好。那为何大部分深度学习函数库反而选择了符号式呢？主要原因还是内存使用和运算时间两方面的效率。我们再来回顾一下本文开头的小例子。

```
import numpy as np

a = np.ones(10)

b = np.ones(10) * 2

c = b * a

d = c + 1

...
```



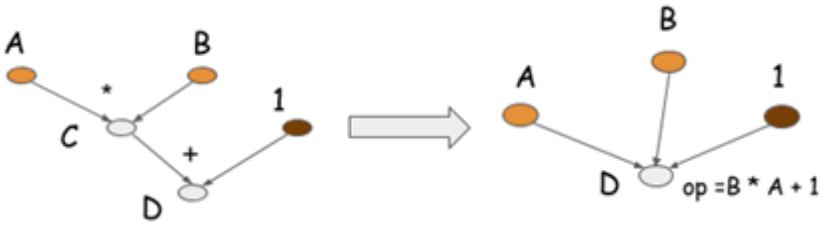
假设数组的每个单元占据 8 字节。如果我们在 Python 控制台执行上述程序需要消耗多少内存呢？我们一起来做些算术题，首先需要存放 4 个包含 10 个元素的数组，需要 4 * 10 * 8 = 320 个字节。但是，若是运行计算图，我们可以重复利用 C 和 D 的内存，只需要 3 * 10 * 8 = 240 字节的内存就够了。

符号式程序的限制更多。当用户对 D 进行编译时，用户告诉系统只需要得到 D 的值。计算的中间结果，也就是 C 的值，对用户是不可见的。这就允许符号式程序重复利用内存进行同址计算（in-place computation）。

然而，命令式程序属于未雨绸缪的类型。如果上述程序在 Python 控制台执行，任何一个变量之后都有可能被用到，系统因此就不能对这些变量共享内存区间了。

当然，这样断言有些理想化，因为命令式程序在变量超出作用域时会启动垃圾回收机制，内存将得以重新利用。但是，受限于“未雨绸缪”这一特点，我们的优化能力还是有限。常见于梯度计算等例子，我们将在在下一节讨论。

符号式程序的另一个优化点是运算折叠。上述代码中，乘法和加法运算可以被折叠为一次运算。如下图所示。这意味着如果使用 GPU 计算，只需用到一个 GPU 内核（而不是两个）。这也正是我们在 cxxnet 和 Caffe 这些优化库中手工调整运算的过程。这样做能提升计算效率。



在命令式程序里我们无法做到。因为中间结果可能在未来某处被引用。这种优化在符号式程序里可行是因为我们得到了完整的计算图，对需要和不需要的变量有一个明确的界线。而命令式程序只做局部运算，没有这条明确的界线。

Backprop 和 AutoDiff 的案例分析

在这一节，我们将基于自动微分或是反向传播的问题对比两种编程模式。梯度计算几乎是所有深度学习库所要解决的问题。使用命令式程序和符号式程序都能实现梯度计算。

我们先看命令式程序。下面这段代码实现自动微分运算，我们之前讨论过这个例子。

```
class array(object) :

    """Simple Array object that support autodiff."""

    def __init__(self, value, name=None):

        self.value = value

        if name:

            self.grad = lambda g : {name : g}



    def __add__(self, other):

        assert isinstance(other, int)

        ret = array(self.value + other)

        ret.grad = lambda g : self.grad(g)

        return ret



    def __mul__(self, other):

        assert isinstance(other, array)
```

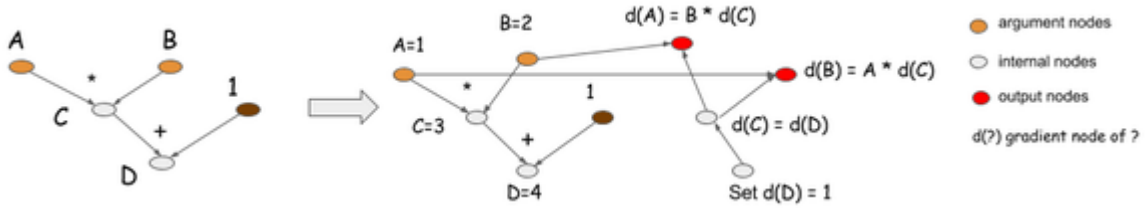
```
        ret = array(self.value * other.value)

        def grad(g):

            x = self.grad(g * other.value)

            x.update(other.grad(g * self.value))

            return x

        ret.grad = grad

        return ret



# some examples

a = array(1, 'a')

b = array(2, 'b')

c = b * a

d = c + 1

print d.value

print d.grad(1)

# Results

# 3

# {'a': 2, 'b': 1}
```

在上述程序里，每个数组对象都含有 grad 函数（事实上是闭包-closure）。当我们执行 d.grad 时，它递归地调用 grad 函数，把梯度值反向传播回来，返回每个输入值的梯度值。看起来似乎有些复杂。让我们思考一下符号式程序的梯度计算过程。下面这段代码是符号式的梯度计算过程。

```
A = Variable('A')
```

```
B = Variable('B')

C = B * A

D = C + Constant(1)

# get gradient node.

gA, gB = D.grad(wrt=[A, B])

# compiles the gradient function.

f = compile([gA, gB])

grad_a, grad_b = f(A=np.ones(10), B=np.ones(10)*2)
```

D 的 grad 函数生成一幅反向计算图，并且返回梯度节点 gA 和 gB。它们对应于下图的红点。



命令式程序做的事和符号式的完全一致。它隐式地在 grad 闭包里存储了一张反向计算图。当执行 d.grad 时，我们从 d(D)开始计算，按照图回溯计算梯度并存储结果。

因此我们发现无论符号式还是命令式程序，它们计算梯度的模式都一致。那么两者的差异又在何处？再回忆一下命令式程序“未雨绸缪”的要求。如果我们准备一个支持自动微分的数组库，需要保存计算过程中的 grad 闭包。这就意味着所有历史变量不能被垃圾回收，因为它们通过函数闭包被变量 d 所引用。那么，若我们只想计算 d 的值，而不想要梯度值该怎么办呢？

在符号式程序中，我们声明 f=compiled([D>)来替换。它也声明了计算的边界，告诉系统我只想计算正向通路的结果。那么，系统就能释放之前结果的存储空间，并且共享输入和输出的内存。

假设现在我们运行的不是简单的示例，而是一个 n 层的深度神经网络。如果我们只计算正向通路，而不用反向（梯度）通路，我们只需分配两份临时空间存放中间层的结果，而不是 n 份。由于命令式程序需要为今后可能用到的梯度值做准备，中间结果不得不保存，就需要用到 n 份临时空间。

正如我们所见，优化的程度取决于对用户行为的约束。符号式程序的思路就是让用户通过编译明确地指定计算的边界。而命令式程序为之后所有情况做准备。符号式程序更充分地了解用户需要什么和不想要什么，这是它的天然优势。

当然，我们也能对命令式程序施加约束条件。例如，上述问题的解决方案之一是引入一个上下文变量。我们可以引入一个没有梯度的上下文变量，来避免梯度值的计算。这给命令式程序带来了更多的约束条件，以换取性能上的改善。

```
with context.NoGradient():

    a = array(1, 'a')

    b = array(2, 'b')

    c = b * a

    d = c + 1
```

然而，上述的例子还是有许多可能的未来，也就是说不能在正向通路中做同址计算来重复利用内存（一种减少 GPU 内存的普遍方法）。这一节介绍的技术产生了显式的反向通路。在 Caffe 和 cxxnet 等工具包里，反向传播是在同一幅计算图内隐式完成的。这一节的讨论同样也适用于这些例子。

大多数基于函数库（如 cxxnet 和 caffe）的配置文件，都是为了一两个通用需求而设计的。计算每一层的激活函数，或是计算所有权重的梯度。这些库也面临同样的问题，若一个库能支持的通用计算操作越多，我们能做的优化（内存共享）就越少，假设都是基于相同的数据结构。

因此经常能看到一些例子在约束性和灵活性之间取舍。

模型检查点

模型存储和重新加载的能力对大多数用户来说都很重要。有很多不同的方式来保存当前工作。通常保存一个神经网络，需要存储两样东西，神经网络结构的配置和各节点的权重值。

支持对配置文件设置检查点是符号式程序的加分项。因为符号式的模型构建阶段并不包含计算步骤，我们可以直接序列化计算图，之后再重新加载它，无需引入附加层就解决了保存配置文件的问题。

```
A = Variable('A')
```

```
B = Variable('B')

C = B * A

D = C + Constant(1)

D.save('mygraph')

...

D2 = load('mygraph')

f = compile([D2])

# more operations

...
```

因为命令式程序逐行执行计算。我们不得不把整块代码当做配置文件来存储，或是在命令式语言的顶部再添加额外的配置层。

参数更新

大多数符号式编程属于数据流（计算）图。数据流图能方便地描述计算过程。然而，它对参数更新的描述并不方便，因为参数的更新会引起变异（mutation），这不属于数据流的概念。大多数符号式编程的做法是引入一个特殊的更新语句来更新程序的某些持续状态。

用命令式风格写参数更新往往容易的多，尤其是当需要相互关联地更新时。对于符号式编程，更新语句也是被我们调用并执行。在某种意义上来讲，目前大部分符号式深度学习库也是退回命令式方法进行更新操作，用符号式方法计算梯度。

没有严格的边界

我们已经比较了两种编程风格。之前的一些说法未必完全准确，两种编程风格之间也没有明显的边界。例如，我们可以用 Python 的（JIT）编译器来编译命令式程序，使我们获得一些符号式编程对全局信息掌握的优势。但是，之前讨论中大部分说法还是正确的，并且当我们开发深度学习库时这些约束同样适用。

大操作 vs 小操作

我们穿越了符号式程序和命令式程序激烈交锋的战场。接下去来谈谈深度学习库所支持的一些操作。各种深度学习库通常都支持两类操作。

- 大的层操作，如 FullyConnected 和 BatchNormalize
- 小的操作，如逐元素的加法、乘法。cxxnet 和 Caffe 等库支持层级别的操作，而 Theano 和 Minerva 等库支持细粒度操作。

更小的操作更灵活

显而易见，因为我们总是可以组合细粒度的操作来实现更大的操作。例如，sigmoid 函数可以简单地拆分为除法和指数运算。

```
sigmoid(x)=1.0/(1.0+exp(-x))
```

如果我们用小运算作为模块，那就能表示大多数的问题了。对于更熟悉 cxxnet 和 Caffe 的读者来说，这些运算和层级别的运算别无二致，只是它们粒度更细而已。

```
SigmoidLayer(x) = EWiseDivisionLayer(1.0, AddScalarLayer(ExpLayer(-x), 1.0))
```

因此上述表达式变为三个层的组合，每层定义了它们的前向和反向（梯度）函数。这给我们搭建新的层提供了便利，因为我们只需把这些东西拼起来即可。

大操作更高效

如你所见，直接实现 sigmoid 层意味着需要用三个层级别的操作，而非一个。

```
SigmoidLayer(x)=EWiseDivisionLayer(1.0,AddScalarLayer(ExpLayer(-x),1.0))
```

这会增加计算和内存的开销（能够被优化）。

因此 cxxnet 和 Caffe 等库使用了另一种方法。为了直接支持更粗粒度的运算，如 BatchNormalization 和 SigmoidLayer，在每一层内人为设置计算内核，只启动一个或少数几个 CUDA 内核。这使得实现效率更高。

编译和优化

小操作能被优化吗？当然可以。这会涉及到编译引擎的系统优化部分。计算图有两种优化形式

- 内存分配优化，重复利用中间结果的内存。
- 计算融合，检测图中是否包含 sigmoid 之类的模式，将其融合为更大的计算核。内存分配优化事实上也不止局限于小运算操作，也能用于更大的计算图。

然而，这些优化对于 cxxnet 和 Caffe 之类的大运算库显得无所谓。因为你从未察觉到它们内部的编译步骤。事实上这些库都包含一个编译的步骤，把各层转化为固定的前向、后向执行计划，逐个执行。

对于包含小操作的计算图，这些优化是至关重要的。因为每次操作都很小，很多子图模式能被匹配。而且，因为最终生成的操作可能无法完全枚举，需要内核显式地重新编译，与大操作库固定的预编译核正好相反。这就是符号式库支持小操作的开销原因。编译优化的需求也会增加只支持小操作库的工程开销。

正如符号式与命令式的例子，大操作库要求用户提供约束条件（对公共层）来“作弊”，因此用户才是真正完成子图匹配的人。这样人脑就把编译时的附加开销给省了，通常也不算太糟糕。

表达式模板和静态类型语言

我们经常需要写几个小操作，然后把它们合在一起。Caffe 等库使用人工设置的内核来组装这些更大模块，否则用户不得不在 Python 端完成这些组装了。

实际上我们还有第三种选择，而且很好用。它被称为表达式模板。基本思想就是在编译时用模板编程从表达式树（expression tree）生成通用内核。更多的细节请移步表达式模板教程。cxxnet 是一个广泛使用表达式模板的库，它使得代码更简洁、更易读，性能和人工设置的内核不相上下。

表达式模板与 Python 内核生成的区别在于表达式模板是在 c++编译时完成，有现成的类型，所以没有运行期的额外开销。理论上其它支持模板的静态类型语言都有该属性，然而目前为止我们只在 C++中见到过。

表达式模板库在 Python 操作和人工设置内核之间开辟了一块中间地带，使得 C++用户可以组合小操作成为一个高效的大操作。这是一个值得考虑的优化选项。

混合各种风格

我们已经比较了各种编程模型，接下去的问题就是该如何选择。在讨论之前，我们必须强调本文所做的比较结果可能并不会对你面临的问题有多少影响，主要还是取决于你的问题。

记得 Amdahl 定律吗，你若是花费时间来优化无关紧要的部分，整体性能是不可能有大幅度提升的。

我们发现通常在效率、灵活性和工程复杂度之间有一个取舍关系。往往不同的编程模式适用于问题的不同部分。例如，命令式程序对参数更新更合适，符号式编程则是梯度计算。

本文提倡的是混合多种风格。回想 Amdahl 定律，有时候我们希望灵活的这部分对性能要求可能并不高，那么简陋一些以支持更灵活的接口也未尝不可。在机器学习中，集成多个模型的效果往往好于单个模型。

如果各个编程模型能以正确的方式被混合，我们取得的效果也很好于单个模型。我们在此列一些可能的讨论。

符号式和命令式程序

有两种方法可以混合符号式和命令式的程序。

- 把命令式程序作为符号式程序调用的一部分。
- 把符号式程序作为命令式程序的一部分。

我们观察到通常以命令式的方法写参数更新更方便，而梯度计算使用符号式程序更有效率。

目前的符号式库里也能发现混合模式的程序，因为 Python 自身是命令式的。例如，下面这段代码把符号式程序融入到 numpy（命令式的）中。

```
A = Variable('A')

B = Variable('B')

C = B * A

D = C + Constant(1)

# compiles the function

f = compile(D)

d = f(A=np.ones(10), B=np.ones(10)*2)

d = d + 1.0
```

它的思想是将符号式图编译为一个可以命令式执行的函数，内部对用户而言是个黑盒。这就像我们常做的，写一段 c++程序并将其嵌入 Python 之中。

然而，把 numpy 当做命令式部分使用并不理想，因为参数的内存是放在 GPU 里。更好的方式是用支持 GPU 的命令式库和编译过的符号式函数交互，或是在符号式程序中加入一小部分代码帮助实现参数更新功能。

小操作和大操作

组合小操作和大操作也能实现，而且我们有一个很好的理由支持这样做。设想这样一个应用，如更换损失函数或是在现有结构中加入用户自定义的层，我们通常的做法是用大操作组合现有的部件，用小操作添加新的部分。

回想 Amdahl 定律，通常这些新部件不太会是计算瓶颈。由于性能的关键部分我们在大操作中已经做了优化，这些新的小操作一点不做优化也能接受，或是做一些内存的优化，而不是进行操作融合的优化。

选择你自己的风格

我们已经比较了深度学习编程的几种风格。本文的目的在于罗列这些选择并比较他们的优劣势。并没有一劳永逸的方法，这并不妨碍保持你自己的风格，或是组合你喜欢的几种风格，创造更多有趣的、智慧的深度学习库。