# 如何使用C++写出一个TensorFlow I~+C+++-Touth-Asvest-TraceFunt流

原文链接：  
原文链接：

Before we start, here’s the code:  
在我们开始之前，这里是代码：

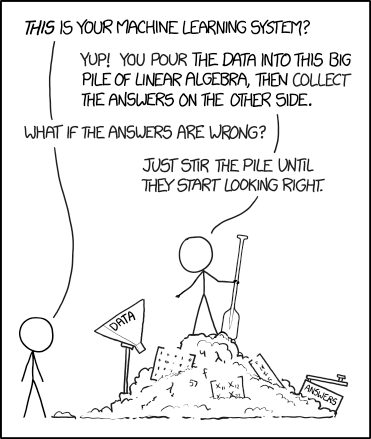
1. [Branch with Eigen backend](https://github.com/OneRaynyDay/autodiff/tree/eigen)
2. [Branch that only supports scalars](https://github.com/OneRaynyDay/autodiff/tree/master)

I worked on this project with .  
我和他一起做这个项目。

# Why? 为什么？

You’ve probably heard this phrase “Don’t roll your own \_\_\_” thousands of times if you’re a CS major. It can be filled with crypto, standard library, parser, etc. I think nowadays, it should also contain ML library.  
如果你是主修计算机专业的学生，你可能已经听过成千上万次这样的话“不要自己滚”。它可以填充密码、标准库、解析器等。我认为现在，它还应该包含ML库。

Regardless of this fact, it’s still an amazing lesson to learn from. People take tensorflow and similar libraries for granted nowadays; they treat it like a black box and let it run. There aren’t enough people who know what’s happening in the back. It’s really just a nonconvex optimization problem! Stop stirring the pile until it looks right.  
不管这个事实如何，这仍然是一个令人惊奇的教训。如今，人们把tensorflow和类似的库视为理所当然；他们把它当作一个黑匣子，任其运行。没有足够的人知道后面发生了什么。这真的只是一个非凸优化问题！别再搅动那堆了，直到它看起来合适为止。



xkcd

# Tensorflow

At tensorflow’s core, there is a big component that allows you to string together operations to form something called an **operator graph**. This operator graph is a directed graph G=(V,E)G=(V,E), where at some nodes u1,u2,…,un,v∈Vu1,u2,…,un,v∈V and e1,e2,…,en∈E,ei=(ui,v)e1,e2,…,en∈E,ei=(ui,v) we have that there exists some kind of operator that maps u1,…,unu1,…,un to vv.

For example, if we have x + y = z, then (x,z),(y,z)∈E(x,z),(y,z)∈E.

This is great for evaluating the arithmetic expression. We can get the result by finding the **sinks** of the operator graph. **Sinks** are vertices such that v∈V,∄e=(v,u)v∈V,∄e=(v,u). In other words, these are vertices that have no directed edges from it to anything else. Similarly, **sources** are v∈V,∄e=(u,v)v∈V,∄e=(u,v).

For us, we will always put in values at the sources, and the values will propagate to the sinks.  
对我们来说，我们总是在源代码处输入值，这些值将传播到汇。

# Reverse-mode Differentation 逆模差分

Here’s some if you think my explanation is bad.  
如果你认为我的解释不好，这里有一些。

Differentiation is a core requirement in many of the models required in tensorflow, because we need it to run gradient descent. Everyone who graduated from highschool\* knows what differentiation is; it’s just take derivatives of functions and then do chain rule if the function is a complicated composition of basic functions!  
微分是tensorflow中许多模型的核心要求，因为我们需要它来运行梯度下降。每个高中毕业的人都知道微分是什么，如果函数是一个复杂的基本函数的组合，它只是取函数的导数，然后做链式法则！

## Super brief overview 简要概述

If we had a function like:  
如果我们有这样一个函数：

f(x,y)=x∗yf(x,y)=x∗y  
f（x，y）=x∗yf（x，y）=x∗y

Then differentiation with respect to x will yield:  
那么关于x的微分将产生：

df(x,y)dx=ydf(x,y)dx=y  
df（x，y）dx=ydf（x，y）dx=y

Then differentiation with respect to y will yield:  
那么关于y的微分将产生：

df(x,y)dy=xdf(x,y)dy=x  
df（x，y）dy=xdf（x，y）dy=x

Here’s another example:  
下面是另一个例子：

f(x1,x2,…,xn)=f(x)=xTxf(x1,x2,…,xn)=f(x)=xTx  
f（x1，x2，…，xn）=f（x）=xTxf（x1，x2，…，xn）=f（x）=xTx

This derivative is just:  
这个导数就是：

df(x)dxi=2xidf(x)dxi=2xi  
DF（x）DXI= 2XIDF（x）DXI＝2XI

So the gradient is just:  
所以梯度就是：

∇xf(x)=2x∇xf(x)=2x  
∇xf（x）=2x∇xf（x）=2x

The chain rule, for example applied to f(g(h(x)))f(g(h(x))):  
链规则，例如应用于f（g（h（x））f（g（h（x））））：

df(g(h(x)))dx=df(g(h(x)))dg(h(x))dg(h(x))dh(x)dh(x)xdf(g(h(x)))dx=df(g(h(x)))dg(h(x))dg(h(x))dh(x)dh(x)x  
df（g（h（x））dx=df（g（h（x））dg（h（x））dg（h（x））dh（x）dh（x）xdf（g（h（x））dx=df（g（h（x）））dg（h（x））dg（h（x））dh（x）dh（x）x

## Reverse Mode in 5 Minutes 5分钟后反转模式

So now keep in mind the DAG structure we have for the operator graph, and the chain rule on the last example. To evaluate, we can see something like:  
所以现在要记住我们对于算子图的DAG结构，以及最后一个例子中的链式规则。要进行评估，我们可以看到如下情况：

x -> h -> g -> f  
x->h->g->f

As a graph. It will give us the answer at f. However, we can go the reversed direction as well:  
作为一个图表。它将在f处给出答案。但是，我们也可以朝相反的方向走：

dx <- dh <- dg <- df  
dx<-dh<-dg<-df

And this will look like the chain rule! We’ll need to multiply the derivatives together on the path to get to our final result.  
这看起来就像链式规则！我们需要把这些导数相乘，才能得到最终的结果。

Here’s an example of an operator graph:  
下面是一个运算符图的示例：

So this basically decays into a graph traversal problem. Does anyone smell topological sort and DFS/BFS?  
所以这基本上会变成一个图遍历问题。有人闻到拓扑排序和DFS/BFS的味道吗？

Yup, so to support topological sort on both ways, we need to contain a set of parents and a set of children, and the sinks are sources for the other direction(vice versa).  
是的，所以要同时支持拓扑排序，我们需要包含一组父类和一组子类，汇是另一个方向的源（反之亦然）。

# Implementation 实施

Before school started, Minh Le and I started designing this project. We decided to use the Eigen library backend for linear algebra operations. They have a matrix class called MatrixXd. We are using that here.  
开学前，我和明乐开始设计这个项目。我们决定使用特征库后端进行线性代数运算。他们有一个矩阵类叫做MatrixXd。我们在用这个。

Each variable node is represented by the var class:  
每个变量节点都由var类表示：

class var {  
// Forward declaration  
struct impl;  
  
public:  
 // For initialization of new vars by ptr  
 var(std::shared\_ptr<impl>);  
  
 var(double);  
 var(const MatrixXd&);  
 var(op\_type, const std::vector<var>&);   
 ...  
   
 // Access/Modify the current node value  
 MatrixXd getValue() const;  
 void setValue(const MatrixXd&);  
 op\_type getOp() const;  
 void setOp(op\_type);  
   
 // Access internals (no modify)  
 std::vector<var>& getChildren() const;  
 std::vector<var> getParents() const;  
 ...  
private:   
 // PImpl idiom requires forward declaration of the class:  
 std::shared\_ptr<impl> pimpl;  
};  
  
struct var::impl{  
public:  
 impl(const MatrixXd&);  
 impl(op\_type, const std::vector<var>&);  
 MatrixXd val;  
 op\_type op;   
 std::vector<var> children;  
 std::vector<std::weak\_ptr<impl>> parents;  
};  
类var{//Forward declarationstruct impl；public:/，用于通过ptr var（std：：shared戋ptr<impl>）；var（double）；var（const matrixd&）；var（op戋type，const std：：vector<var>&）。。。//访问/修改当前节点值matrixd getValue（）const；void setValue（const matrixd&）；op\_type getOp（）const；void setOp（op\_type）；//Access internals（no Modify）std：：vector<var>&getChildren（）const；std：：vector<var>getParents（）const；…private://PImpl习惯用法需要类的前向声明：std：：shared嫒ptr<impl>PImpl；}；struct var：：impl{public:impl（const matrixd&）；impl（op嫒type，const std：：vector<var>&）；matrixd val；op嫒type op；std：：vector<var>children；std：：vector<std：：weak嫒ptr<impl>>parents；}；

In here, we employed the pImpl idiom, which means “pointer to implementation”. It’s great for many things, like decoupling implementation from interface, and allowing us to instantiate things on the heap when we have a local shell of interface on the stack. Some side-effects of pImpl are slightly slower runtime, but much shorter compile time. This allows us to keep our data structures persistent through multiple function calls/returns. A tree data structure like this should be persistent.  
在这里，我们使用了pImpl这个成语，意思是“指向实现的指针”。这对于很多事情都很好，比如将实现与接口分离，并且允许我们在堆栈上有一个本地的接口外壳时在堆上实例化一些东西。pImpl的一些副作用是运行时稍慢，但编译时间要短得多。这允许我们通过多个函数调用/返回来保持数据结构的持久性。像这样的树数据结构应该是持久的。

We have a couple enums which tells us which operations are currently being performed:  
我们有几个枚举，告诉我们当前正在执行哪些操作：

enum class op\_type {  
 plus,  
 minus,  
 multiply,  
 divide,  
 exponent,  
 log,  
 polynomial,  
 dot,  
 ...  
 none // no operators. leaf.  
};  
枚举类操作类型{加、减、乘、除、指数、对数、多项式、点。。。无//无运算符。叶。}；

The actual class that’s performing the evaluation of this tree is called expression:  
执行此树计算的实际类称为表达式：

class expression {  
public:  
 expression(var);  
 ...  
 // Recursively evaluates the tree.  
 double propagate();  
 ...  
 // Computes the derivative for the entire graph.  
 // Performs a top-down evaluation of the tree.  
 void backpropagate(std::unordered\_map<var, double>& leaves);  
 ...   
private:  
 var root;  
};  
类表达式{public:expression（var）。。。//递归计算树。双传播（）。。。//计算整个图的导数。//对树执行自上而下的求值。void backpropagate（std：：无序映射<var，double>&leaves）。。。private:var根；}；

Inside of backpropagate, we have code that does something similar to this:  
在backpropagate中，我们有一些代码执行类似的操作：

backpropagate(node, dprev):  
 derivative = differentiate(node)\*dprev  
 for child in node.children:  
 backpropagate(child, derivative)   
backpropagate（node，dprev）：导数=微分（node）\*dprev表示node中的子节点。children:backpropagate（子节点，导数）

This is pretty much doing a DFS; you see it?  
这几乎是在做一个DFS；你看到了吗？

## Why C++? 为什么是C++？

In fact, C++ is probably not the correct language to use for this. We could’ve spent much less time developing in a functional language like OCaml. Now I realize why Scala is being used in machine learning, mainly spark ;).  
事实上，C++可能不是正确的语言。我们本可以花更少的时间开发像OCaml这样的函数式语言。现在我明白了为什么Scala被用于机器学习，主要是spark；）。

However, there are obvious benefits to C++:  
但是，C++有明显的好处：

### Eigen 本征

For example, we can directly use tensorflow’s linear algebra library, called Eigen. It’s a template-abusing lazy-evaluation linear algebra library. Similar in flavour to our expression tree, we build up the expression, and it will only be evaluated when we really need to. However, for Eigen, they determine this during compile time,which is when templates are being used, meaning runtime is decreased. I have a lot of respect for the people who wrote Eigen, since looking at template errors make my eyes bleed.  
例如，我们可以直接使用tensorflow的线性代数库Eigen。它是一个滥用惰性求值线性代数库的模板。与表达式树的风格类似，我们构建表达式，并且只有在真正需要时才会对其求值。然而，对于Eigen，它们在编译时（即使用模板时）确定这一点，这意味着运行时减少。我很尊重写Eigen的人，因为看到模板错误会让我的眼睛流血。

Their code would look something like:  
他们的代码看起来像：

Matrix A(...), B(...);  
auto lazy\_multiply = A.dot(B);  
typeid(lazy\_multiply).name(); // the class name is something like Dot\_Matrix\_Matrix.  
Matrix(lazy\_multiply); // functional-style casting forces evaluation of this matrix.  
Matrix A（…）、B（…）；auto lazy\_multiply=A.dot（B）；typeid（lazy\_multiply）.name（）；///类名类似于dot\_Matrix\_Matrix.Matrix（lazy\_multiply）；///函数式强制计算此矩阵。

The Eigen library is very powerful, and that’s why it’s one of the main backends that tensorflow uses themselves. That means there are other optimizations other than this lazy evaluation technique.  
特征库非常强大，这就是为什么它是tensorflow自己使用的主要后端之一。这意味着除了这种懒惰的计算技术之外，还有其他优化。

### Operator Overload 操作员过载

Developing this library in Java would’ve been nice - no shared\_ptrs, unique\_ptrs, weak\_ptrs; we get an actual, capable, GC. This saves development time by a lot, not to mention probably faster in execution speed as well. However, Java doesn’t allow operator overloads, and consequently they can’t have this:  
用Java开发这个库会很好——没有共享的、唯一的、弱的；我们得到了一个实际的、有能力的GC。这节省了大量的开发时间，更不用说可能更快的执行速度了。但是，Java不允许运算符重载，因此它们不能这样做：

// These 3 lines code up an entire neural network!  
var sigm1 = 1 / (1 + exp(-1 \* dot(X, w1)));  
var sigm2 = 1 / (1 + exp(-1 \* dot(sigm1, w2)));  
var loss = sum(-1 \* (y \* log(sigm2) + (1-y) \* log(1-sigm2)));  
//这三行代码组成了一个完整的神经网络！var sigm1=1/（1+exp（-1\*dot（X，w1））；var sigm2=1/（1+exp（-1\*dot（sigm1，w2））；var loss=sum（-1\*（y\*log（sigm2））+（1-y）\*log（1-sigm2））；

The above is actual , by the way. Isn’t this extremely pretty? I would argue that this is even prettier than the python wrapper for tensorflow. And just to let you know, these are matrices, as well.  
顺便说一句，以上是事实。这不是很漂亮吗？我认为这甚至比tensorflow的python包装器还要漂亮。为了让你知道，这些也是矩阵。

In Java, this would’ve been extremely ugly, with a bunch of add(), divide()… and et cetera. More importantly, the users would be implicitly forcing PEMDAS, which C++’s operators already exhibit very well.  
在Java中，如果有一堆add（）、divide（）…等等，这将是非常难看的。更重要的是，用户会隐式地强迫PMEDAS，C++的操作员已经表现得很好。

## Features, Not Bugs 功能，而不是错误

There are some things that you can actually specify in this library that tensorflow doesn’t have clear API for, or not that I know of. For example, if we wanted to train only a specific subset of the weights, we can actually only backpropagate to the specific sources we’re interested in. This is really useful for things like transfer learning for convolutional neural nets, since many times a large net, like VGG19, is beheaded and then appended with a few extra layers of which the weights are trained according to the new domain samples.  
你可以在这个库中指定一些tensorflow没有清晰的API，或者我不知道的东西。例如，如果我们只想训练一个特定的权重子集，我们实际上只能反向传播到我们感兴趣的特定源。这对于卷积神经网络的传递学习非常有用，因为很多时候，一个大的网络，比如VGG19，被斩首，然后附加一些额外的层，其中的权重是根据新的域样本训练的。

## Benchmarks 基准

On Python’s Tensorflow library, training for 10000 epochs on the Iris dataset for classification, with the same hyperparameters, we have:  
在Python的Tensorflow库中，使用相同的超参数，在Iris数据集上训练10000个epoch进行分类，我们得到：

1. Tensorflow’s neural net: 23812.5 ms  
   张量流神经网络：23812.5ms
2. Scikit’s neural net library: 22412.2 ms  
   Scikit神经网络库：22412.2ms
3. Autodiff’s neural net, with iterative, optimized: 25397.2 ms  
   Autodiff神经网络，迭代优化：25397.2ms
4. Autodiff’s neural net, with iterative, no optimize: 29052.4 ms  
   Autodiff神经网络，迭代，无优化：29052.4ms
5. Autodiff’s neural net, with recursive, no optimize: 28121.5 ms  
   Autodiff神经网络，递归，无优化：28121.5ms

So it seems, surprisingly, Scikit runs the fastest out of all of these. It may be because we’re not doing huge matrix multiplications. It may be that tensorflown had to take an extra compilation step, with variable initializers and what not. Or, it’s perhaps we had to run loops inside of python rather than in C(python loops are really bad!). I’m not sure myself.  
因此，令人惊讶的是，Scikit似乎是所有这些中跑得最快的。这可能是因为我们没有做巨大的矩阵乘法运算。可能是tensorflown需要额外的编译步骤，使用变量初始值设定项等等。或者，我们可能不得不在python中运行循环，而不是在C中运行（python循环真的很糟糕！）我自己也不确定。

I am fully aware that this is by no means a comprehensive benchmarking test, as it only is applied to a single data point, and in a specific situation. However, the performance of this library is not meant to be state of the art, since **we don’t ever want to roll our own tensorflow**.