**[译] 计算图上的微积分：Backpropagation**

字数2806 阅读1004 评论1 喜欢10

[原文](http://colah.github.io/posts/2015-08-Backprop/)

**引言**

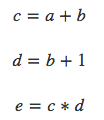
Backpropagation (BP) 是使得训练深度模型在计算上可行的关键算法。对现代神经网络，这个算法相较于无脑的实现可以使梯度下降的训练速度提升千万倍。而对于模型的训练来说，这其实是 7 天和 20 万年的天壤之别。  
除了在深度学习中的使用，BP 本身在其他的领域中也是一种强大的计算工具，例如从天气预报到分析数值的稳定性——只是同一种思想拥有不同的名称而已。实际上，BP 已经在不同领域中被重复发明了数十次了（参见 [Griewank (2010)](http://www.math.uiuc.edu/documenta/vol-ismp/52_griewank-andreas-b.pdf" \t "_blank)）。更加一般性且与应用场景独立的名称叫做 **反向微分** (reverse-mode differentiation)

反向微分，我自己翻译的，如果有人知道正确的翻译请告知。

**从本质上看，BP 是一种快速求导的技术，可以作为一种不单单用在深度学习中并且可以胜任大量数值计算场景的基本的工具。**

**计算图**

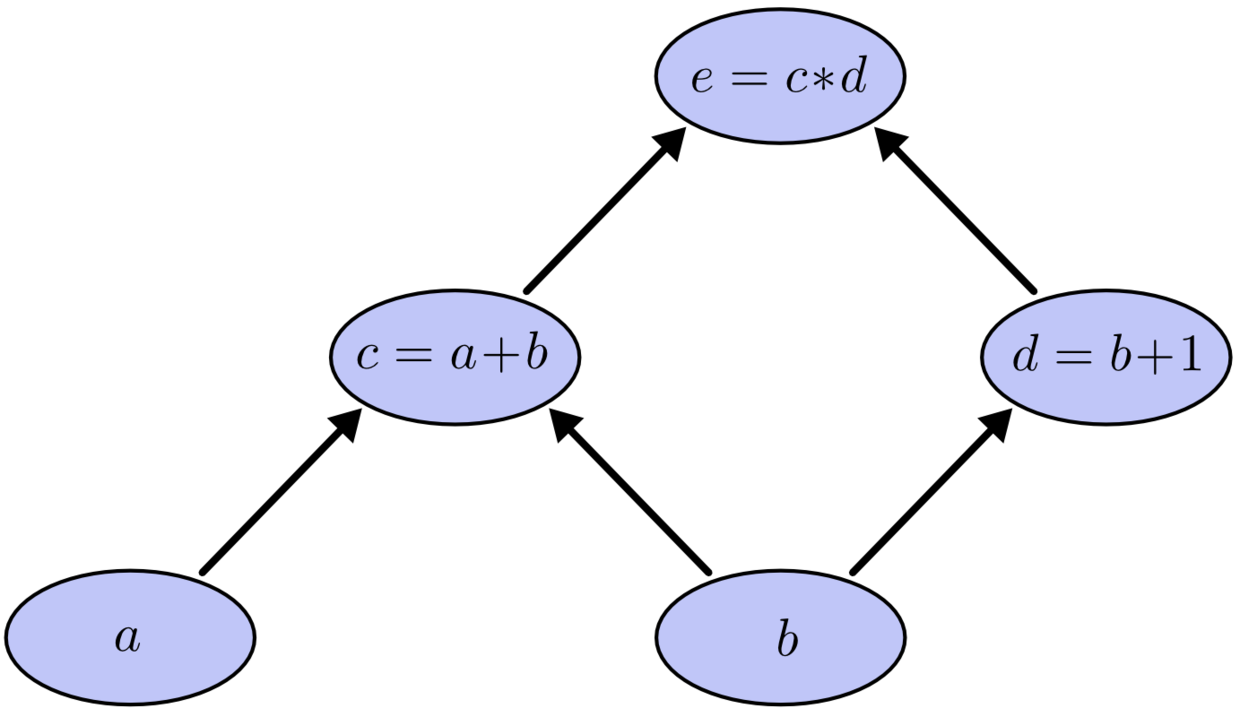
计算图是种很好的研究数学表达式的方式。例如，我们有这样一个表达式 e = (a + b) \* ( b + 1)。其包含三个操作：两个加法和一个乘法。为了更好的讲述，我们引入两个中间变量，c 和 d，这样每个函数的输出就有一个变量表示了。现在我们有：



*表达式*

下面可以创建计算图了，我们将每个表达式和输入的变量看做是节点。如果一个节点的值是另一个节点的输入，就画出一条从该节点到另一节点的边。

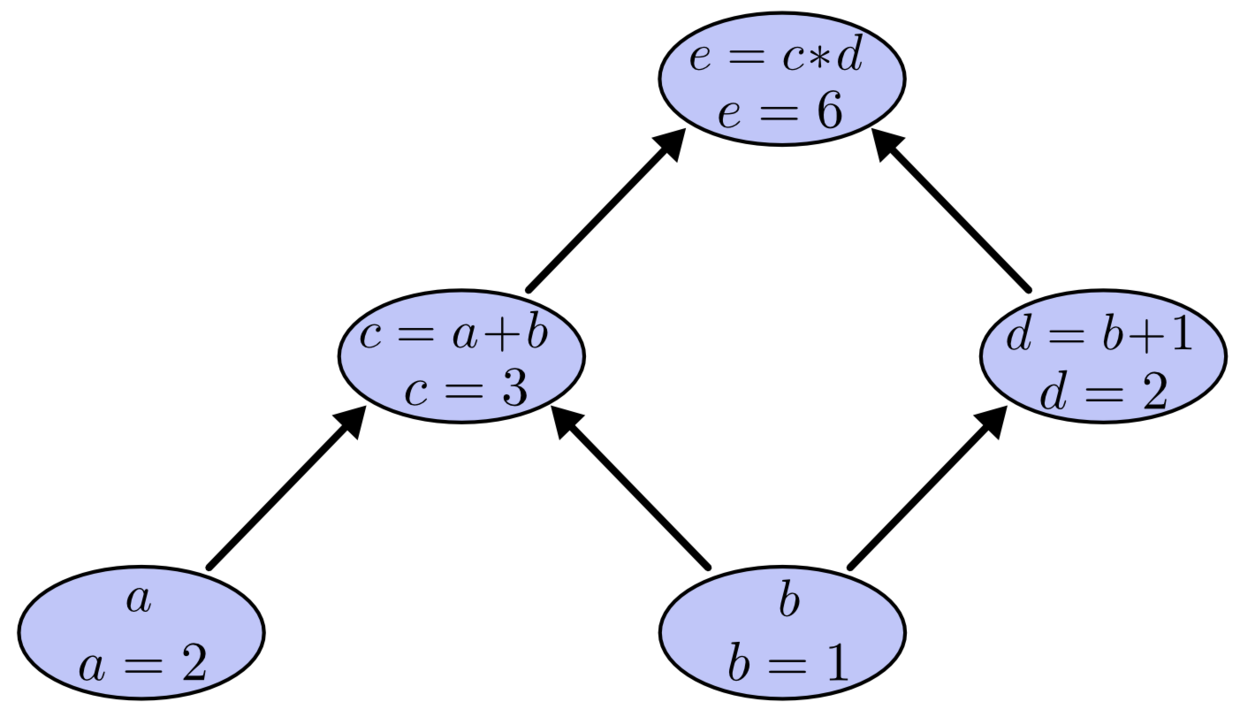
计算图是有向图



*计算图示例*

这种样式的图在计算机科学领域到处可见，特别是在函数式程序中。他们与依赖图（dependency graph）或者调用图（call graph）紧密相关。同样他们也是非常流行的深度学习框架 [Theano](http://deeplearning.net/software/theano/" \t "_blank) 背后的核心抽象。

对于上面用计算图表示的表达式，我们设置对应输入变量的值，通过这个图来计算每个节点的值。例如，假设 a = 2, b = 1：

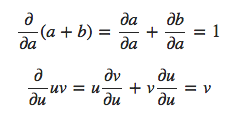


*计算在计算图上*

最终表达式的值就是 6。

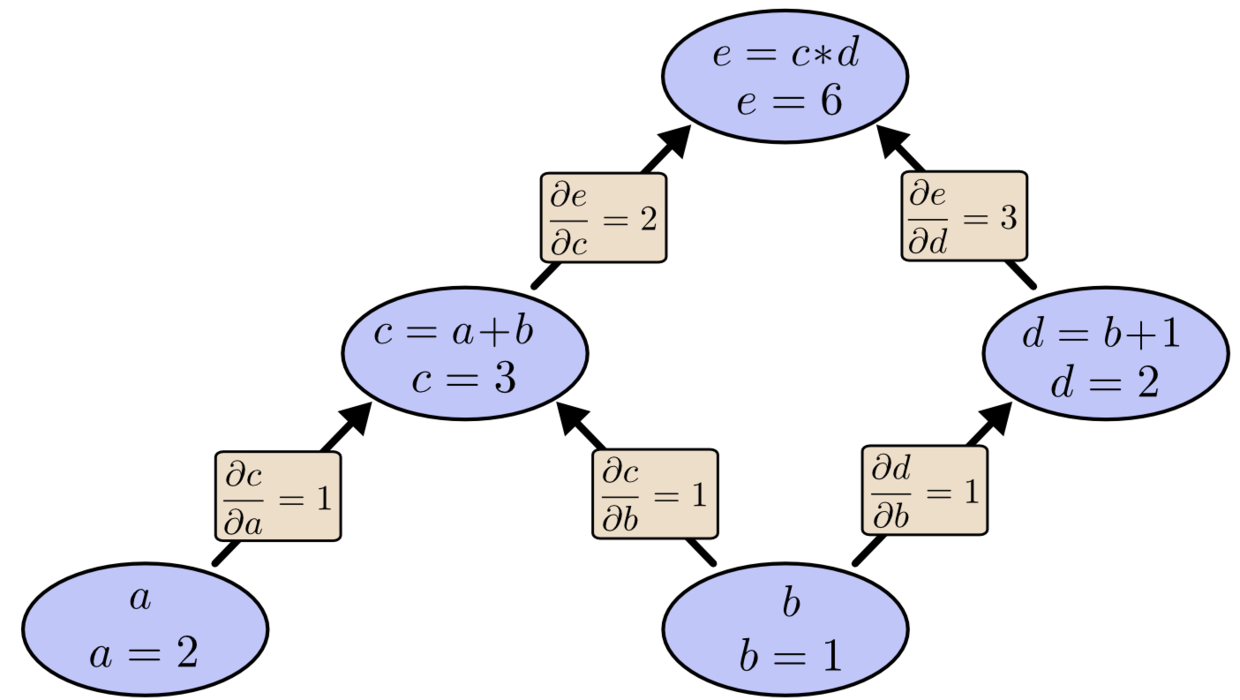
**计算图上的导数**

如果想要理解计算图上的导数，那么关键之处就是理解每条边上的导数。如果 a 直接影响 c，我们就想知道 a 如何影响了 c。如果 a 改变了一丢丢，c 会发生什么样的变化？这种东西我们称 c 关于 a 的偏导数。  
为了计算在这幅图中的偏导数，我们需要 和式法则（[sum rule](https://en.wikipedia.org/wiki/Sum_rule_in_differentiation" \t "_blank) ）和 乘式法则（[product rule](https://en.wikipedia.org/wiki/Product_rule)）：



*和式法则 和 乘式法则*

下面，在图中每条边上都有对应的导数了：



*计算偏导数*

那如果我们想知道哪些没有直接相连的节点之间的影响关系呢？假设就看看 e 如何被 a 影响的。如果我们以 1 的速度改变 a，那么 c 也是以 1 的速度在改变，导致 e 发生了 2 的速度在改变。因此 e 是以 1 \* 2 的关于 a 变化的速度在变化。  
而一般的规则就是对一个点到另一个点的所有的可能的路径进行求和，每条路径对应于该路径中的所有边的导数之积。因此，为了获得 e 关于 b 的导数：

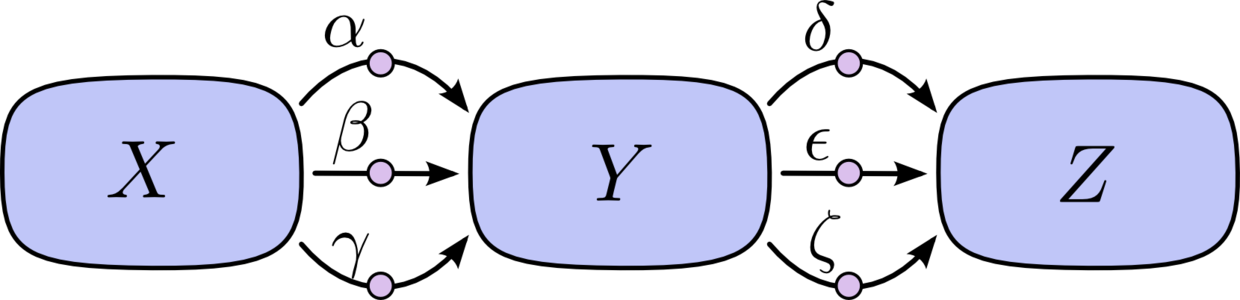
http://upload-images.jianshu.io/upload_images/42741-57c7494df45c6a87.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240

*e 关于 b 的导数*

这个值就代表着 b 改变的速度通过 c 和 d 影响到 e 的速度。  
**路径求和**的法则其实就是 **多元链式法则**（[multivariate chain rule](https://en.wikipedia.org/wiki/Chain_rule#Higher_dimensions)）的 另一种思考方式。

**分解路径**

路径求和可能路径数量很容易就会组合爆炸。



*路径数量*

在上面的图中，从 X 到 Y 有三条路径，从 Y 到 Z 也有三条。如果我们希望计算dZ/dX，那么就要对 3 \* 3 = 9 条路径进行求和了：

http://upload-images.jianshu.io/upload_images/42741-904d94e6e0aacd51.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240

*dZ/dX*

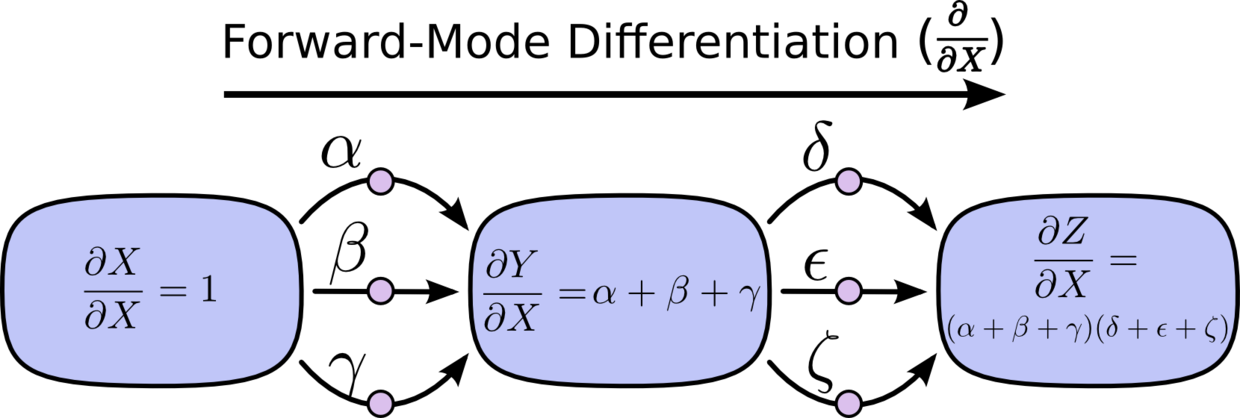
上面的图有 9 条路径，但是在图变得更加复杂的时候，这个数量会指数级地增长。  
相比于粗暴地对所有的路径进行求和，更好的方式是进行因式分解：

http://upload-images.jianshu.io/upload_images/42741-b34e4cb64c08e067.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240

*因式分解*

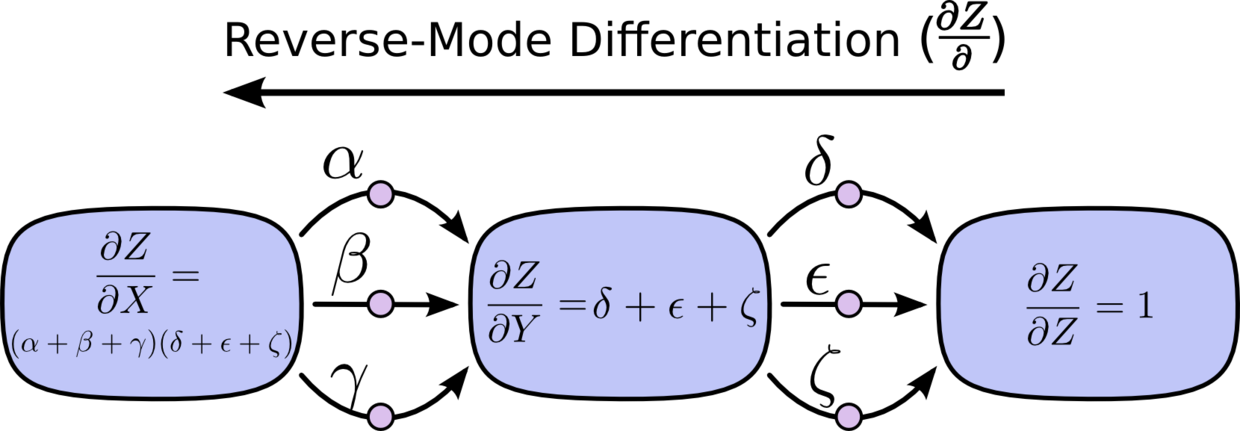
**注意了！这里就是 *前向微分* 和 *反向微分* 诞生的地方！** 这两个算法是通过因式分解来高效计算导数的。通过在每个几点上反向合并路径而非显式地对所有的路径求和来大幅提升计算的速度。实际上，两个算法对每条边的访问都只有**一次**！

前向微分从图的输入开始，一步一步到达终点。在每个节点处，对输入的路径进行求和。每个这样的路径都表示输入影响该节点的一个部分。通过将这些影响加起来，我们就得到了输入影响该节点的全部，也就是关于输入的导数。



*前向微分*

尽管你可能没有从图的结构来考虑这个问题，前向微分其实是在学习了微积分后我们的自然的思维方式。  
相对的，反向微分是从图的输出开始，反向一步一步抵达最开始输入处。在每个节点处，会合了所有源于该节点的路径。



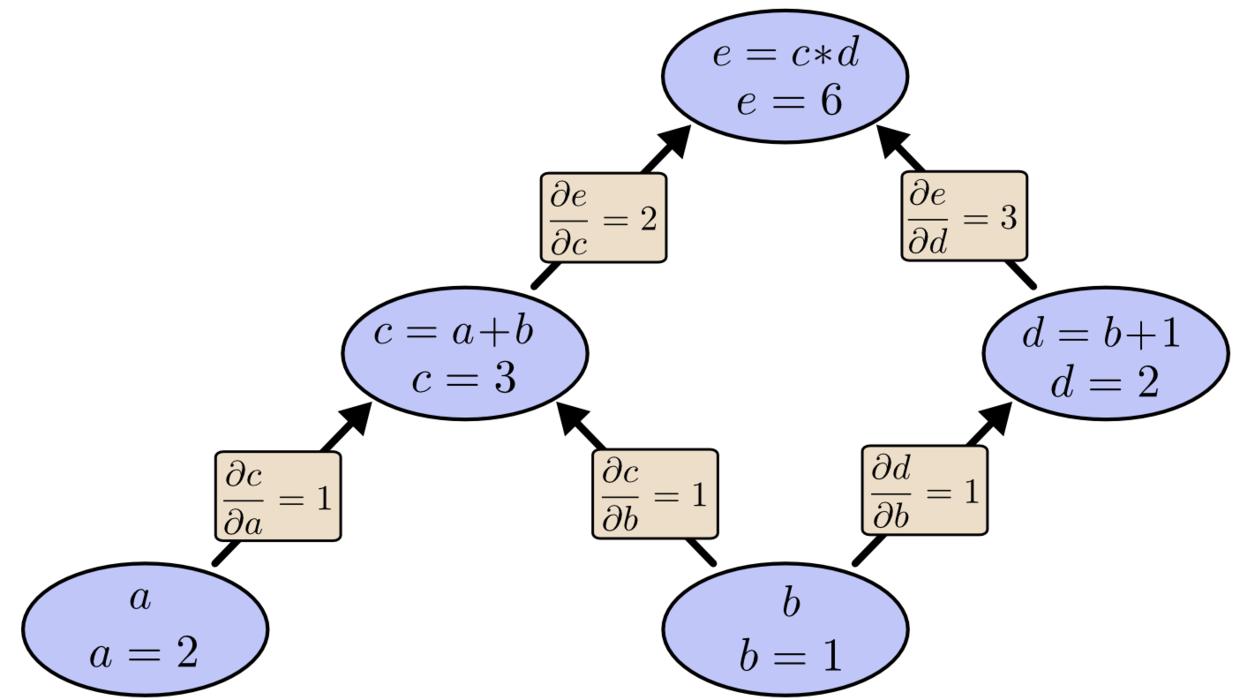
*反向微分*

前向微分 跟踪了输入如何改变每个节点的情况。反向微分 则跟踪了每个节点如何影响输出的情况。也就是说，前向微分应用操作 d/dX 到每个节点，而反向微分应用操作 dZ/d到每个节点。

这其实可以看做是动态规划（[dynamic programming](https://en.wikipedia.org/wiki/Dynamic_programming)）

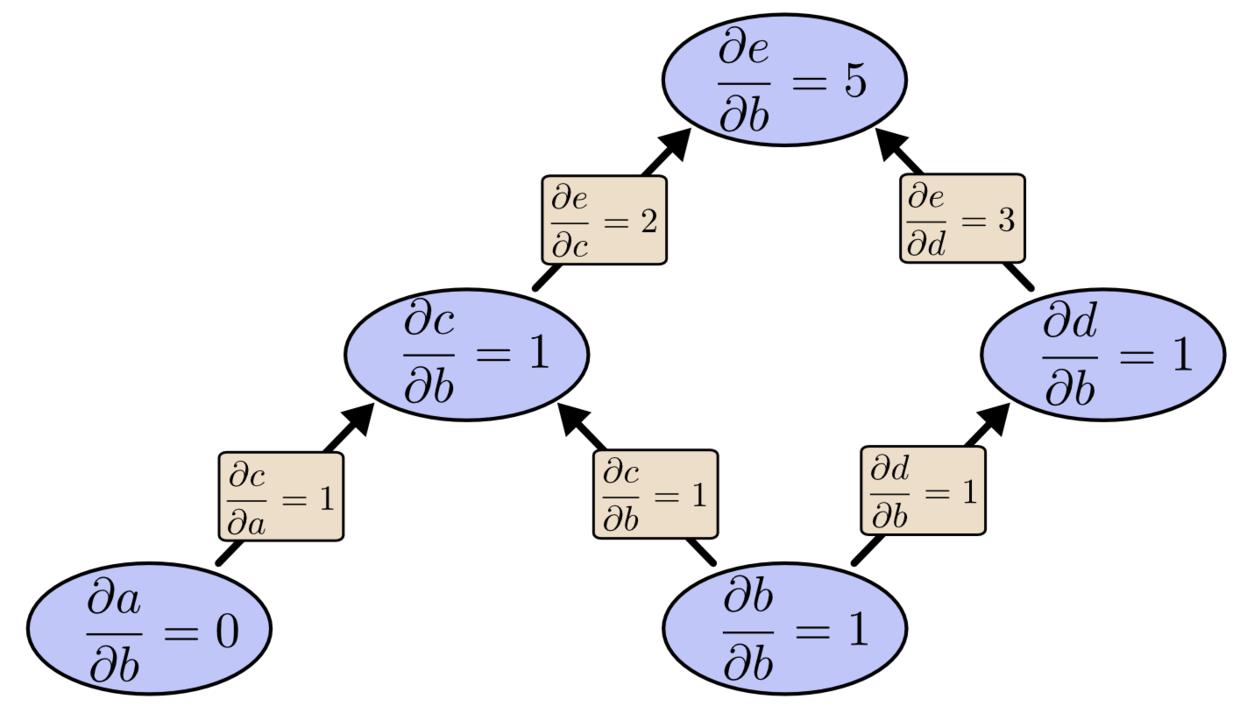
**计算上的胜利**

现在，你可能想知道为何人人都关心 反向微分 了。因为它本身看起来像是用一种奇怪的方式和前向微分做了同样的事情。这里有什么优点？  
让我们重新看看刚开始的例子：



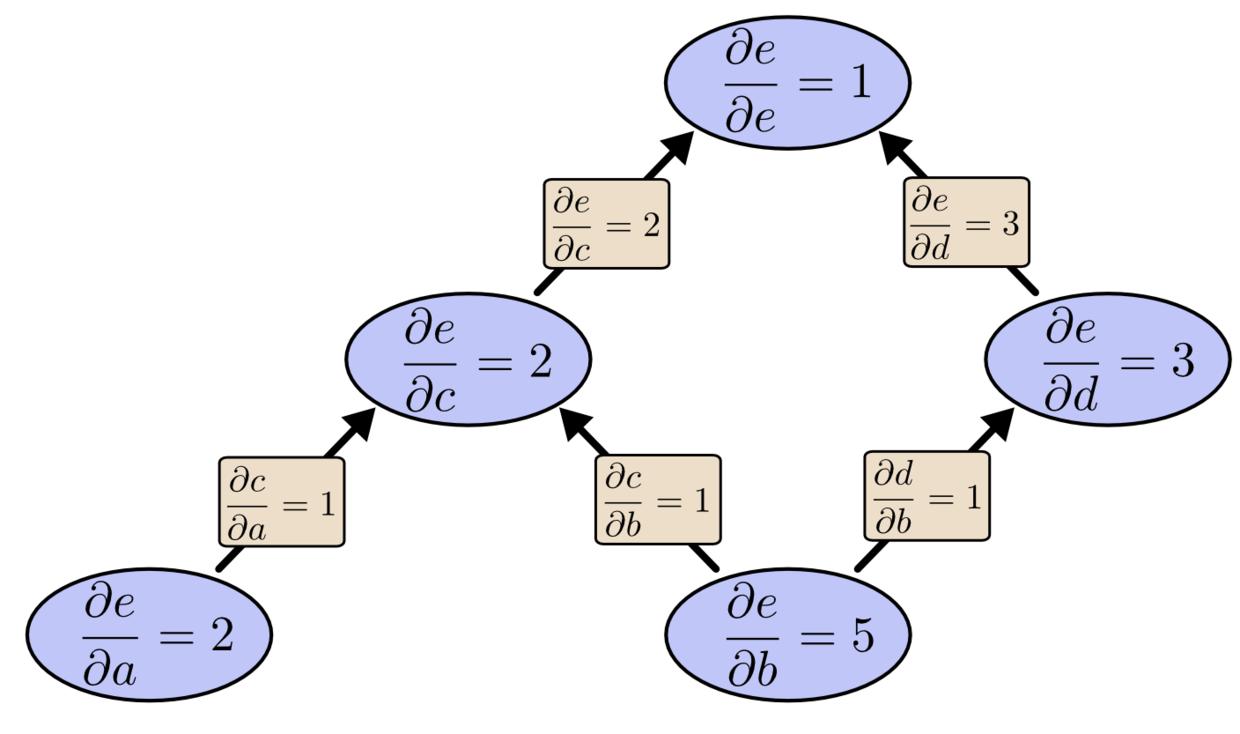
*计算图示例*

我们可以从 b 往上使用前向微分。这样获得了每个节点关于 b 的导数。



*关于 b 求导数*

我们已经计算得到了 de/db，输出关于一个输入 b 的导数。  
如果我们从 e 往下计算反向微分呢？这会得到 e 关于每个节点的导数：



*反向微分*

我们说到反向微分给出了 e 关于每个节点的导数，这里的确是每·一·个节点。我们得到了 de/da 和 de/db，e 关于输入 a 和 b 的导数。前向微分给了我们输出关于某一个输入的导数，而反向微分则给出了所有的导数。  
这幅图中，仅仅是两个因子在影响，但是你想象一个拥有百万个输入和一个输出的函数。前向微分需要百万次遍历计算图才能得到最终的导数，而反向微分仅仅需要一次就能得到所有的导数！百万级的速度提升多么美妙！  
训练神经网络时，我们将衡量神经网络表现的代价函数看做是那些决定网络行为的参数的函数。我们希望计算出代价函数关于所有参数的偏导数，从而进行梯度下降（[gradient descent](https://en.wikipedia.org/wiki/Gradient_descent)）。现在，常常会遇到百万甚至千万级的参数的神经网络。所以，反向微分，也就是 BP，在神经网络中发挥了关键作用！  
（有人要问，有使用前向微分更加合理的场景么？当然！因为反向微分得到一个输出关于所有输入的导数，前向微分得到了所有输出关于一个输出的导数。如果遇到了一个有多个输出的函数，前向微分肯定更加快速）

**这难道不是 Trivial 的嘛！？**

刚刚理解 BP 本质时，我的反应是：“Oh，这不就是链式法则么！？为什么人们花了这么久才能够发现！？” 我也并不是唯一有这种反应的。如果你问问“是不是还有更巧妙的计算前馈神经网络的导数的方法？”，这个答案并不是很难。  
但是我觉得，发明 BP 要比其本身看起来更加困难。你看，在BP被发明的那段时间里，人们并不非常关注前馈神经网络。并且使用导数来训练网络并不是很明显。在人们发现可以快速计算导数时，这种方法才会进入人们的视野。这里存在着循环依赖的关系。  
更糟糕的是，在日常思维中很容易忽略这种循环依赖关系。使用导数来训练神经网络？肯定你会困在局部最优解中。更明显的是，计算这些导数的代价非常大。仅仅因为我们知道这个观点可行，我们并没有立即开始研究那些不可能的原因究竟是什么。  
这也许就是事后诸葛亮的好处。一旦你已经构建出问题本身，最困难的工作便搞定了。

**结论**

计算导数远比你想象的要简单。这就是这篇文章告诉你的主要观点。实际上，这些方法是反直觉地简单，我们人类还是会傻傻地重新发现。在深度学习中，计算导数是相当重要的一件事，同样在其他领域中也是非常有用的知识。只不过还没成为一种众人皆知的事物。  
还有其他可以学到的东西么？肯定有。  
BP 也是一种理解导数在模型中如何流动的工具。在推断为何某些模型优化非常困难的过程中，BP 也是特别重要的。典型的例子就是在 Recurrent Neural Network 中理解 vanishing gradient 的原因。  
最后，我还要补充的是，这些技术中还有很多算法上的经验可以借鉴。BP 和 前向微分使用了一对技巧（线性化和动态规划）来更有效地计算导数。如果你真正理解了这些技术，你就可以有效地计算其他有趣包含导数的表达式。后面的博客也会继续做介绍。  
本文给出了关于 BP 的相对抽象的描述。强烈建议大家阅读 Michael Nielsen 关于 BP 的讲述（[chapter 2](http://neuralnetworksanddeeplearning.com/chap2.html" \t "_blank)），更加贴合神经网络本身。

**致谢**

Thank you to [Greg Corrado](http://research.google.com/pubs/GregCorrado.html), [Jon Shlens](https://shlens.wordpress.com/), [Samy Bengio](http://bengio.abracadoudou.com/) and [Anelia Angelova](http://www.vision.caltech.edu/anelia/) for taking the time to proofread this post.  
Thanks also to [Dario Amodei](https://www.linkedin.com/pub/dario-amodei/4/493/393), [Michael Nielsen](http://michaelnielsen.org/) and [Yoshua Bengio](http://www.iro.umontreal.ca/~bengioy/yoshua_en/index.html" \t "_blank) for discussion of approaches to explaining backpropagation. Also thanks to all those who tolerated me practicing explaining backpropagation in talks and seminar series!