Yes you should understand backprop
是的，你应该了解反向传播

Andrej Karpathy · Follow
7 min read · Dec 20, 2016

Listen Share

When we offered CS231n (Deep Learning class) at Stanford, we intentionally designed the programming assignments to include explicit calculations involved in backpropagation on the lowest level. The students had to implement the forward and the backward pass of each layer in raw numpy. Inevitably, some students complained on the class message boards:

当我们在斯坦福大学提供CS231n （深度学习课程）时，我们特意设计了编程作业，以包含最低级别的反向传播中涉及的显式计算。学生必须在原始 numpy 中实现每一层的前向和后向传递。难免有同学在班级留言板上抱怨：

> *"Why do we have to write the backward pass when frameworks in the real world, such as TensorFlow, compute them for you automatically?"*
>
> *"当现实世界中的框架（例如 TensorFlow）自动为您计算它们时，为什么我们必须编写反向传递？ "*

This is seemingly a perfectly sensible appeal - if you're never going to write backward passes once the class is over, why practice writing them? Are we just torturing the students for our own amusement? Some easy answers could make arguments along the lines of *"it's worth knowing what's under the hood as an intellectual curiosity"*, or perhaps *"you might want to improve on the core algorithm later"*, but there is a much stronger and practical argument, which I wanted to devote a whole post to:

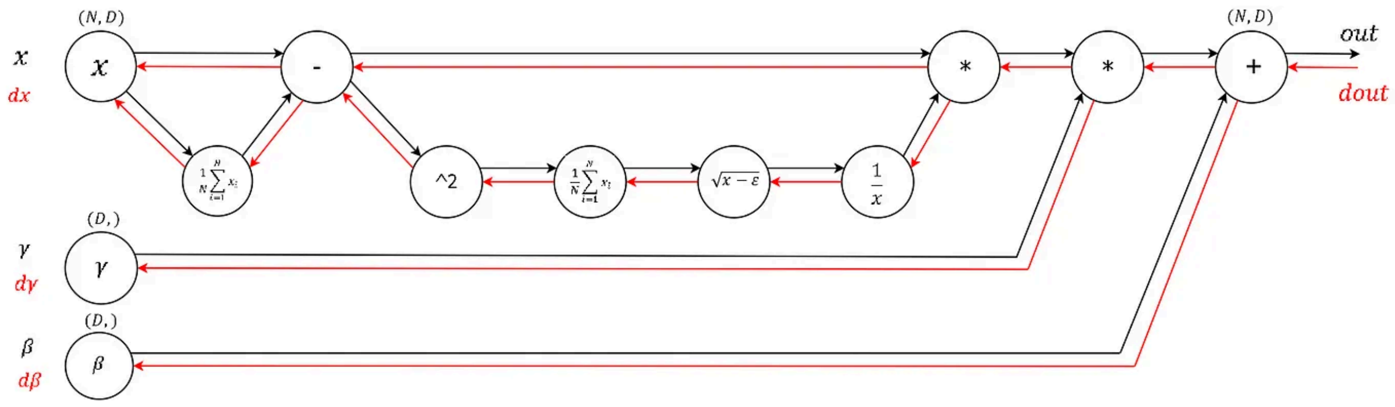
这似乎是一个完全明智的呼吁 - 如果你永远不会在课程结束后编写向后传递，为什么要练习编写它们呢？难道我们只是为了自己的消遣而折磨学生吗？一些简单的答案可能会提出这样的论点：*"作为一种求知欲，了解幕后的内容是值得的"*，或者*"你可能想稍后改进核心算法"*，但有一个更强大和实用的论点，我想用整篇文章来讨论：

> **The problem with Backpropagation is that it is a leaky abstraction.**

> 反向传播的问题在于它是一个有漏洞的抽象。

In other words, it is easy to fall into the trap of abstracting away the learning process — believing that you can simply stack arbitrary layers together and backprop will "magically make them work" on your data. So lets look at a few explicit examples where this is not the case in quite unintuitive ways.

换句话说，很容易陷入抽象学习过程的陷阱——相信你可以简单地将任意层堆叠在一起，并且反向传播将"神奇地让它们在你的数据上工作"。因此，让我们看一些明确的例子，但情况并非如此，以非常不直观的方式。



Some eye candy: a computational graph of a Batch Norm layer with a forward pass (black) and backward pass (red). (borrowed from this post)

一些养眼的东西：带有前向传递（黑色）和反向传递（红色）的 Batch Norm 层的计算图。（借自这篇文章）

Vanishing gradients on sigmoids

sigmoid 上的梯度消失

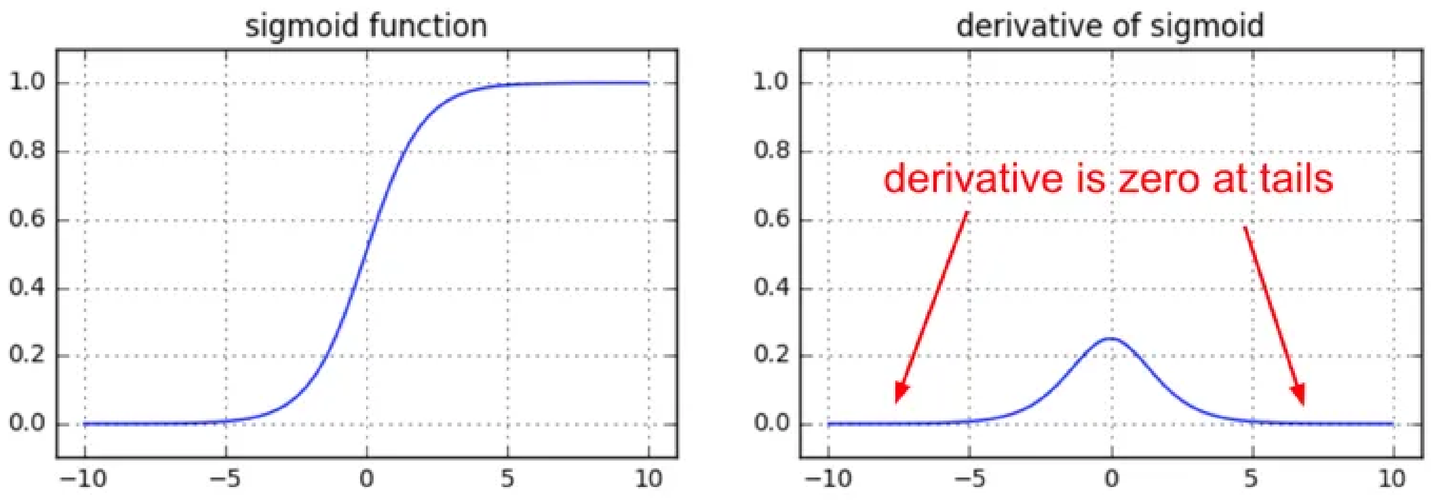
We're starting off easy here. At one point it was fashionable to use **sigmoid** (or **tanh**) non-linearities in the fully connected layers. The tricky part people might not realize until they think about the backward pass is that if you are sloppy with the weight initialization or data preprocessing these non-linearities can "saturate" and entirely stop learning — your training loss will be flat and refuse to go down. For example, a fully connected layer with sigmoid non-linearity computes (using raw numpy):

我们从这里开始很容易。在某一时刻，在全连接层中使用**sigmoid** （或**tanh** ）非线性很流行。人们在考虑反向传递之前可能不会意识到，如果你对权重初始化或数据预处理马虎，这些非线性可能会"饱和"并完全停止学习——你的训练损失将是平坦的并且拒绝继续下去向下。例如，具有 sigmoid 非线性的全连接层计算（使用原始 numpy）：

```
z = 1/(1 + np.exp(-np.dot(W, x))) # forward pass
dx = np.dot(W.T, z*(1-z)) # backward pass: local gradient for x
dW = np.outer(z*(1-z), x) # backward pass: local gradient for W
```

If your weight matrix **W** is initialized too large, the output of the matrix multiply could have a very large range (e.g. numbers between -400 and 400), which will make all outputs in the vector **z** almost binary: either 1 or 0. But if that is the case, **z*(1-z)**, which is local gradient of the sigmoid non-linearity, will in both cases become **zero** ("vanish"), making the gradient for both **x** and **W** be zero. The rest of the backward pass will come out all zero from this point on due to multiplication in the chain rule.

如果你的权重矩阵**W**初始化得太大，矩阵乘法的输出可能有一个非常大的范围（例如-400到400之间的数字），这将使向量**z**中的所有输出几乎都是二进制的：要么1要么0。但是如果是这种情况，则 **z*(1-z)** （S 型非线性的局部梯度）在两种情况下都将变为零（"消失"），从而使**x**和**W**的梯度都为零。由于链式法则中的乘法，从此时起，向后传递的其余部分将全部为零。



Another non-obvious fun fact about sigmoid is that its local gradient (z*(1-z)) achieves a maximum at 0.25, when z = 0.5. That means that every time the gradient signal flows through a sigmoid gate, its magnitude always diminishes by one quarter (or more). If you're using basic SGD, this would make the lower layers of a network train much slower than the higher ones.

关于 sigmoid 的另一个不明显的有趣事实是，当 z = 0.5 时，其局部梯度 (z*(1-z)) 在 0.25 处达到最大值。这意味着每次梯度信号流过 sigmoid 门时，其幅度总是会减小四分之一（或更多）。如果您使用基本 SGD，这将使网络的较低层训练速度比较高层慢得多。

TLDR: if you're using **sigmoids** or **tanh** non-linearities in your network and you understand backpropagation you should always be nervous about making sure that the initialization doesn't cause them to be fully saturated. See a longer explanation in this CS231n lecture video.

总而言之：如果你正在使用**乙状结肠**或者**正值**如果您的网络中存在非线性并且您了解反向传播，那么您应该始终对确保初始化不会导致它们完全饱和感到紧张。请参阅此CS231n 讲座视频中的详细说明。

Dying ReLUs 垂死的 ReLU

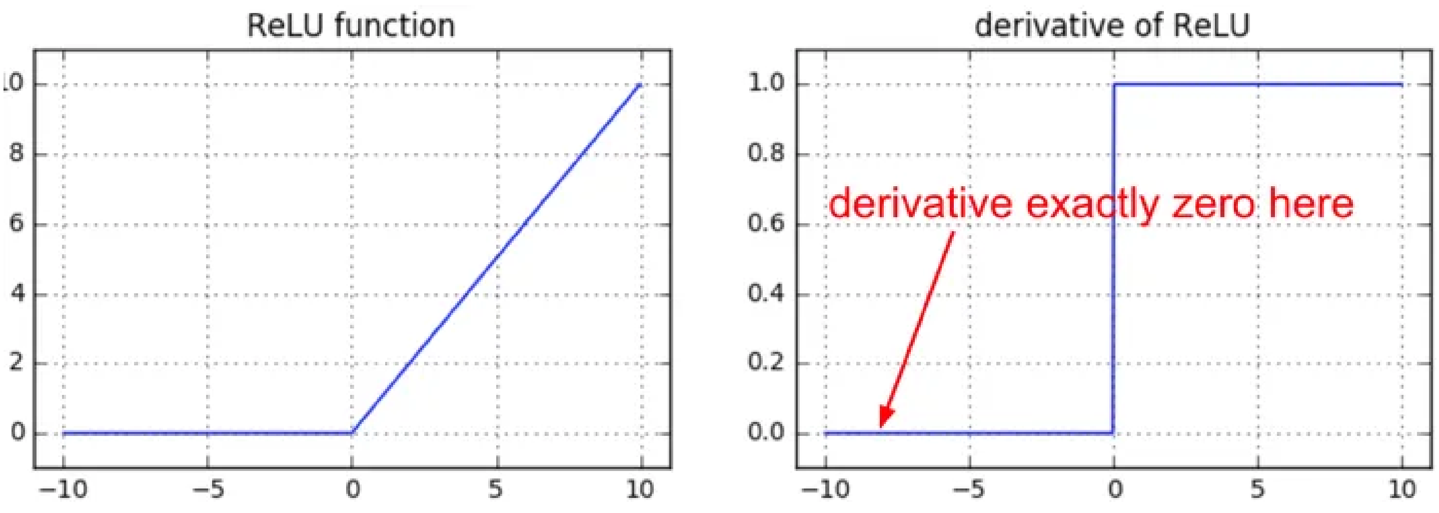
Another fun non-linearity is the ReLU, which thresholds neurons at zero from below. The forward and backward pass for a fully connected layer that uses ReLU would at the core include:

另一个有趣的非线性是 ReLU，它将神经元的阈值从下面设置为零。使用 ReLU 的全连接层的前向和后向传递的核心包括：

```
z = np.maximum(0, np.dot(W, x)) # forward pass
dW = np.outer(z > 0, x) # backward pass: local gradient for W
```

If you stare at this for a while you'll see that if a neuron gets clamped to zero in the forward pass (i.e. **z**=0, it doesn't "fire"), then its weights will get zero gradient. This can lead to what is called the "dead ReLU" problem, where if a ReLU neuron is unfortunately initialized such that it never fires, or if a neuron's weights ever get knocked off with a large update during training into this regime, then this neuron will remain permanently dead. It's like permanent, irrecoverable brain damage. Sometimes you can forward the entire training set through a trained network and find that a large fraction (e.g. 40%) of your neurons were zero the entire time.

如果你盯着这个看一会儿，你会发现如果一个神经元在前向传递中被钳位到零（即**z** = 0，它不会"触发"），那么它的权重将得到零梯度。这可能会导致所谓的"dead ReLU"问题，即如果 ReLU 神经元不幸被初始化而永远不会激发，或者如果在训练到该状态期间神经元的权重因大幅更新而被取消，则该神经元将永远死亡。这就像永久性的、不可恢复的脑损伤。有时，您可以通过经过训练的网络转发整个训练集，并发现很大一部分（例如 40%）神经元始终为零。



TLDR: If you understand backpropagation and your network has ReLUs, you're always nervous about dead ReLUs. These are neurons that never turn on for any example in your entire training set, and will remain permanently dead. Neurons can also die during training, usually as a symptom of aggressive learning rates. See a longer explanation in CS231n lecture video.

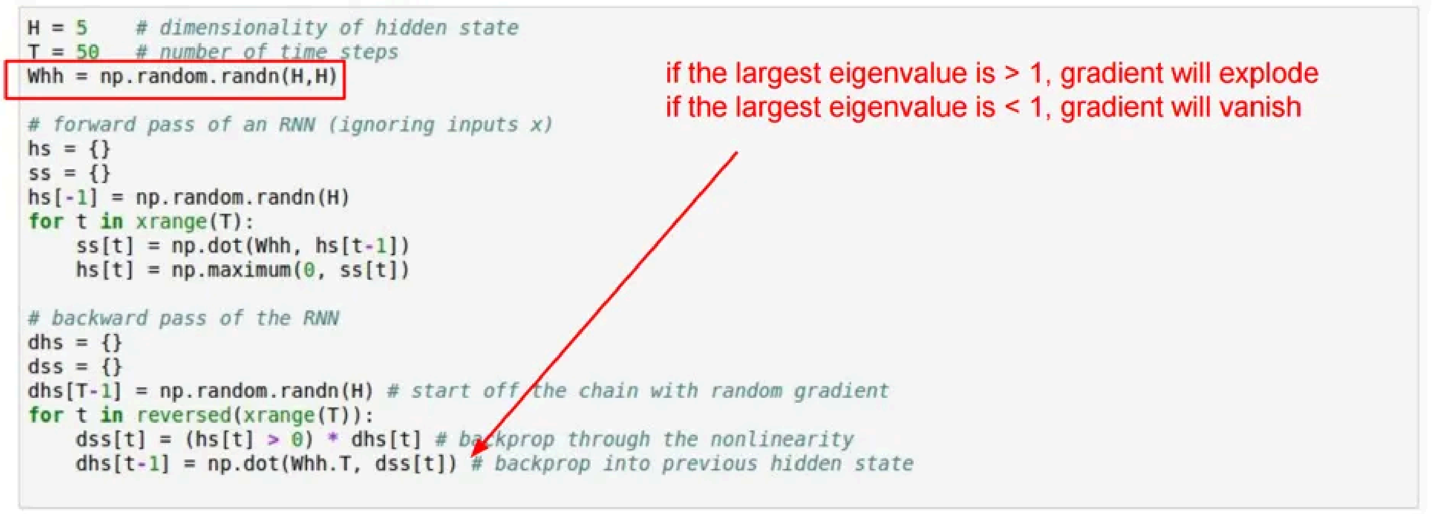
TLDR： 如果您了解反向传播并且您的网络具有 ReLU，那么您总是会对失效的 ReLU 感到紧张。这些神经元在整个训练集中的任何示例中都不会打开，并且将永久死亡。神经元也可能在训练过程中死亡，这通常是学习率过高的症状。请参阅CS231n 讲座视频中的详细说明。

Exploding gradients in RNNs

RNN 中的梯度爆炸

Vanilla RNNs feature another good example of unintuitive effects of backpropagation. I'll copy paste a slide from CS231n that has a simplified RNN that does not take any input **x**, and only computes the recurrence on the hidden state (equivalently, the input **x** could always be zero):

Vanilla RNN 是反向传播的非直观效应的另一个很好的例子。我将复制粘贴 CS231n 中的一张幻灯片，该幻灯片具有简化的 RNN，不接受任何输入**x**，并且仅计算隐藏状态上的递归（等效地，输入**x**可以始终为零）：



```
H = 5    # dimensionality of hidden state
T = 50   # number of time steps
Whh = np.random.randn(H,H)

# forward pass of an RNN (ignoring inputs x)
hs = {}
ss = {}
hs[-1] = np.random.randn(H)
for t in xrange(T):
    ss[t] = np.dot(Whh, hs[t-1])
    hs[t] = np.maximum(0, ss[t])

# backward pass of the RNN
dhs = {}
dss = {}
dhs[T-1] = np.random.randn(H) # start off the chain with random gradient
for t in reversed(xrange(T)):
    dss[t] = (hs[t] > 0) * dhs[t] # backprop through the nonlinearity
    dhs[t-1] = np.dot(Whh.T, dss[t]) # backprop into previous hidden state
```

if the largest eigenvalue is > 1, gradient will explode
if the largest eigenvalue is < 1, gradient will vanish

This RNN is unrolled for **T** time steps. When you stare at what the backward pass is doing, you'll see that the gradient signal going backwards in time through all the hidden states is always being multiplied by the same matrix (the recurrence matrix **Whh**), interspersed with non-linearity backprop.

该 RNN 展开**T**个时间步长。当您盯着向后传递正在执行的操作时，您会发现通过所有隐藏状态及时向后移动的梯度信号始终乘以相同的矩阵（递归矩阵**Whh** ），并散布着非线性反向传播。

What happens when you take one number **a** and start multiplying it by some other number **b** (i.e. a*b*b*b*b*b*b...)? This sequence either goes to zero if |**b**| < 1, or explodes to infinity when |**b**|>1. The same thing happens in the backward pass of an RNN, except **b** is a matrix and not just a number, so we have to reason about its largest eigenvalue instead.

当您将一个数字**a**开始与其他数字**b**相乘（即 a*b*b*b*b*b...）时会发生什么？如果|**b**|，则该序列要么为零< 1，或者当|**b**|>1时爆炸到无穷大。同样的事情也发生在 RNN 的后向传播中，只不过**b**是一个矩阵而不仅仅是一个数字，所以我们必须推断它的最大特征值。

TLDR: If you understand backpropagation and you're using RNNs you are nervous about having to do gradient clipping, or you prefer to use an LSTM. See a longer explanation in this CS231n lecture video.

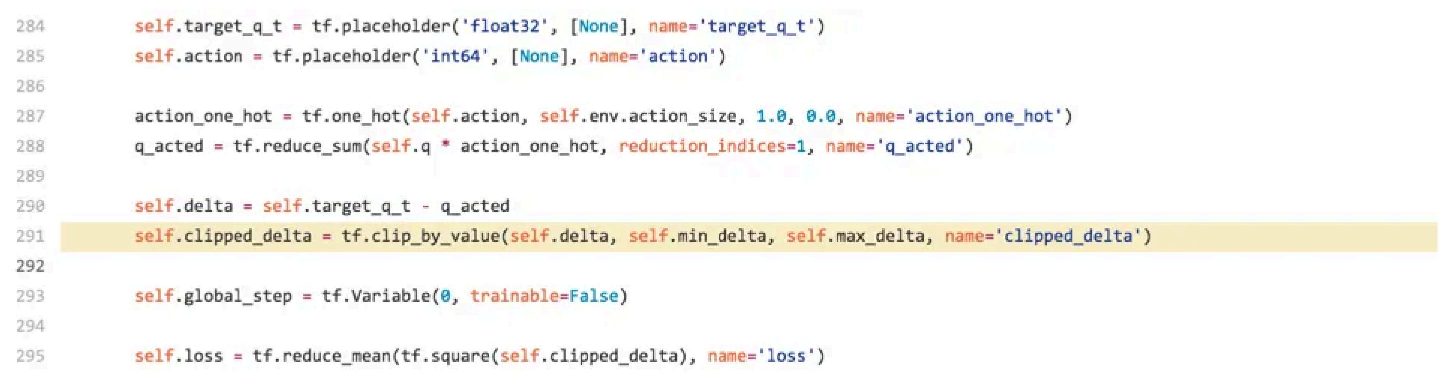
TLDR： 如果您了解反向传播并且正在使用 RNN，那么您会担心必须进行梯度裁剪，或者您更喜欢使用 LSTM。请参阅此CS231n 讲座视频中的详细说明。

Spotted in the Wild: DQN Clipping

野外发现：DQN 剪辑

Lets look at one more — the one that actually inspired this post. Yesterday I was browsing for a Deep Q Learning implementation in TensorFlow (to see how others deal with computing the numpy equivalent of **Q[:, a]**, where **a** is an integer vector — turns out this trivial operation is not supported in TF). Anyway, I searched *"dqn tensorflow"*, clicked the first link, and found the core code. Here is an excerpt:

让我们再看一个——真正激发这篇文章灵感的那个。昨天，我在 TensorFlow 中浏览深度 Q 学习实现（看看其他人如何处理计算**Q[:, a]**的 numpy 等价物，其中**a**是整数向量 - 结果发现 TF 不支持这个简单的操作）。不管怎样，我搜索了*"dqn tensorflow"*，点击第一个链接，找到了核心代码。以下是摘录：



```
284        self.target_q_t = tf.placeholder('float32', [None], name='target_q_t')
285        self.action = tf.placeholder('int64', [None], name='action')
286
287        action_one_hot = tf.one_hot(self.action, self.env.action_size, 1.0, 0.0, name='action_one_hot')
288        q_acted = tf.reduce_sum(self.q * action_one_hot, reduction_indices=1, name='q_acted')
289
290        self.delta = self.target_q_t - q_acted
291        self.clipped_delta = tf.clip_by_value(self.delta, self.min_delta, self.max_delta, name='clipped_delta')
292
293        self.global_step = tf.Variable(0, trainable=False)
294
295        self.loss = tf.reduce_mean(tf.square(self.clipped_delta), name='loss')
```

If you're familiar with DQN, you can see that there is the **target_q_t,** which is just **[reward * \gamma \argmax_a Q(s',a)]**, and then there is **q_acted**, which is **Q(s,a)** of the action that was taken. The authors here subtract the two into variable **delta,** which they then want to minimize on line 295 with the L2 loss with **tf.reduce_mean(tf.square()).** So far so good.

如果您熟悉 DQN，您可以看到有**target_q_t**，即**[reward * \gamma \argmax_a Q(s',a)]**，然后是**q_acted**，即**Q(s,a)**所采取的行动。作者在这里将两者减去变量**delta，**然后他们希望在第 295 行使用**tf.reduce_mean(tf.square())** 最小化 L2 损失。到目前为止，一切都很好。

The problem is on line 291. The authors are trying to be robust to outliers, so if the delta is too large, they clip it with **tf.clip_by_value.** This is well-intentioned and looks sensible from the perspective of the forward pass, but it introduces a major bug if you think about the backward pass.

问题出在第 291 行。作者试图对异常值保持鲁棒性，因此如果增量太大，他们会使用**tf.clip_by_value**对其进行剪辑。这是出于好意，从前向传球的角度来看也很合理，但如果你考虑后向传球，它就会引入一个重大错误。

The **clip_by_value** function has a local gradient of zero outside of the range **min_delta** to **max_delta**, so whenever the delta is above min/max_delta, the gradient becomes exactly zero during backprop. The authors are clipping the raw Q delta, when they are likely trying to clip the gradient for added robustness. In that case the correct thing to do is to use the Huber loss in place of **tf.square**:

在**min_delta**到**max_delta**范围之外， **clip_by_value**函数的局部梯度为零，因此每当 delta 高于 min/max_delta 时，梯度在反向传播期间就恰好为零。作者正在削减原始 Q 增量，而他们可能试图削减梯度以增加鲁棒性。在这种情况下，正确的做法是使用 Huber 损失代替 **tf.square**：

```
def clipped_error(x):
  return tf.select(tf.abs(x) < 1.0,
                   0.5 * tf.square(x),
                   tf.abs(x) - 0.5) # condition, true, false
```

It’s a bit gross in TensorFlow because all we want to do is clip the gradient if it is above a threshold, but since we can’t meddle with the gradients directly we have to do it in this round-about way of defining the Huber loss. In Torch this would be much more simple.

这在 TensorFlow 中有点恶心，因为我们想要做的就是在梯度高于阈值时修剪梯度，但由于我们不能直接干预梯度，所以我们必须以这种定义 Huber 损失的迂回方式来做到这一点。在 Torch 中这会简单得多。

I submitted an issue on the DQN repo and this was promptly fixed.

我在 DQN 存储库上提交了一个问题，该问题很快得到了修复。

In conclusion 综上所述

Backpropagation is a leaky abstraction; it is a credit assignment scheme with non-trivial consequences. If you try to ignore how it works under the hood because “TensorFlow automagically makes my networks learn”, you will not be ready to wrestle with the dangers it presents, and you will be much less effective at building and debugging neural networks.

反向传播是一个有漏洞的抽象；这是一个具有重要后果的信用分配方案。如果你因为“TensorFlow 自动让我的网络学习”而试图忽略它在幕后的工作原理，那么你就不会准备好应对它带来的危险，并且在构建和调试神经网络方面的效率也会大大降低。

The good news is that backpropagation is not that difficult to understand, if presented properly. I have relatively strong feelings on this topic because it seems to me that 95% of backpropagation materials out there present it all wrong, filling pages with mechanical math. Instead, I would recommend the CS231n lecture on backprop which emphasizes intuition (yay for shameless self-advertising). And if you can spare the time, as a bonus, work through the CS231n assignments, which get you to write backprop manually and help you solidify your understanding.

好消息是，如果表述得当，反向传播并不难理解。我对这个话题有相对强烈的感受，因为在我看来，95% 的反向传播材料都呈现错误，充满了机械数学。相反，我会推荐关于反向传播的 CS231n 讲座，它强调直觉（是的，无耻的自我广告）。如果您有空闲时间，作为奖励，可以完成CS231n 作业，这可以让您手动编写反向传播并帮助您巩固理解。

That’s it for now! I hope you’ll be much more suspicious of backpropagation going forward and think carefully through what the backward pass is doing. Also, I’m aware that this post has (unintentionally!) turned into several CS231n ads. Apologies for that :)

现在就这样了！我希望您对未来的反向传播更加怀疑，并仔细思考反向传播正在做什么。另外，我知道这篇文章（无意中！）变成了几个 CS231n 广告。对此表示歉意:)