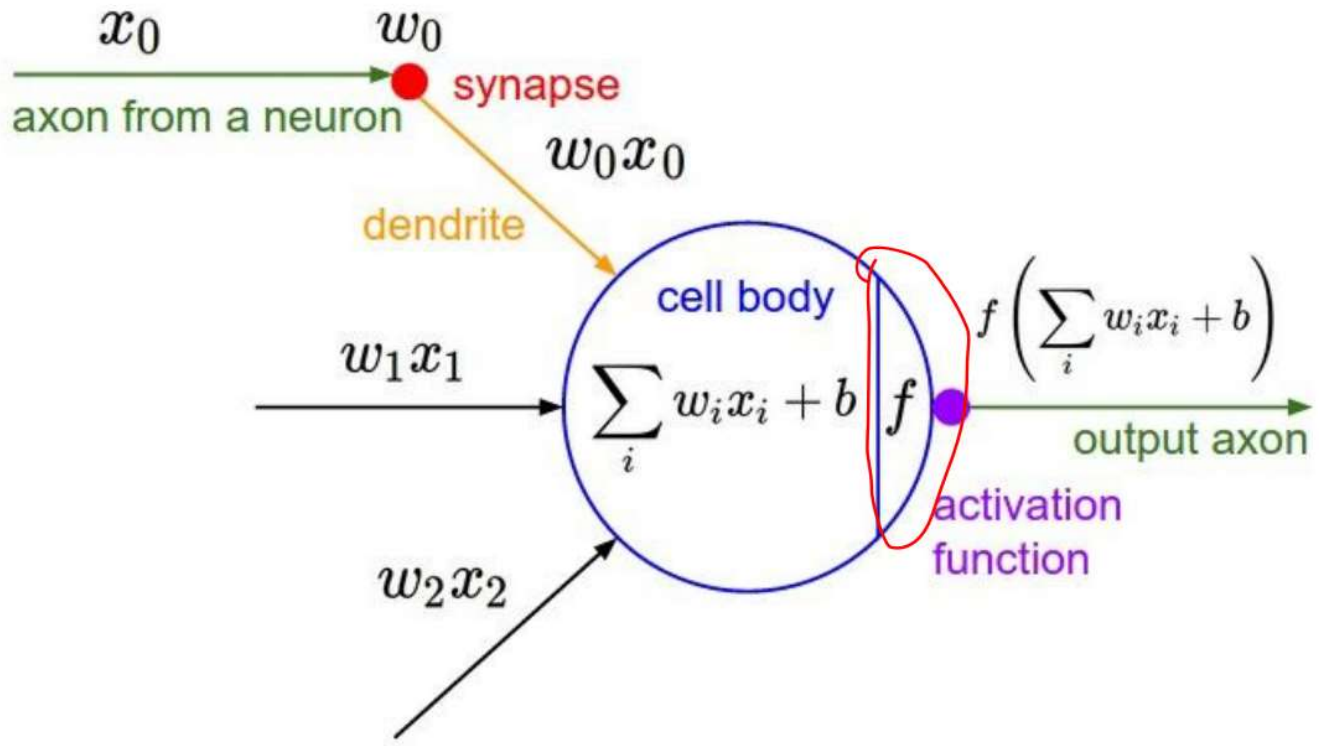
Lecture 6 Training Neutral Networks

一、Activation Functions(激活函数)

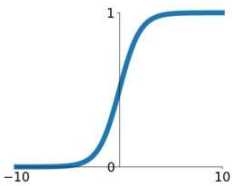
其实激活函数就是在每一层的层运算与输出值之间的一个处理过程



常见的激活函数有以下几类：

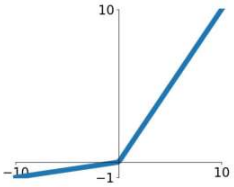
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



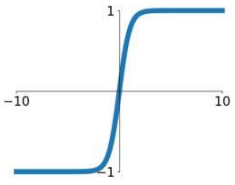
Leaky ReLU

$\max(0.1x, x)$



tanh

$\tanh(x)$

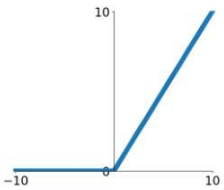


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

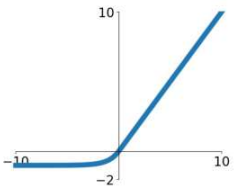
ReLU

$\max(0, x)$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



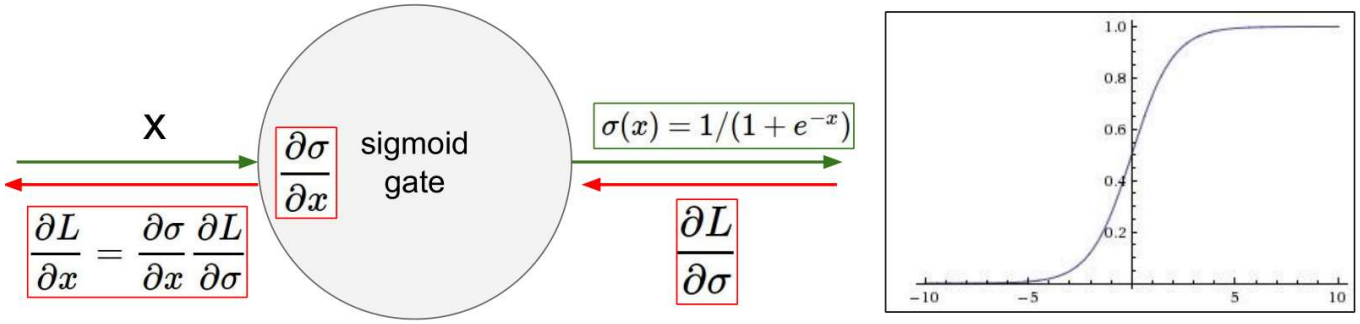
1. Sigmoid

Sigmoid虽然在之前比较常用，也比较的简单，但是他存在着三个有点致命的问题：

(1) Saturated neutrons "kill" the gradients

这个问题的理解，可以看下面这张图。其实看下面的三个问题就明白了。当x为-10和10的时候，Sigmoid的梯度接近于0，当x为0的时候，sigmoid的梯度是一个正常的值。这样问题就暴露出来了。

当激活函数的输入值过大或者过小的时候，在这里的**local gradient**就接近于0，导致的后果就是后一级的梯度几乎无法传到前一级，这样相当于是截断了梯度的传递，是非常不好的。关键的问题在于，这个函数的饱和值比较小，意味着很容易达到饱和，这样就很危险了。



What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

(2) Sigmoid outputs are not zero-centered

这个的意思是说，sigmoid的函数均值并不是0。这样似乎本身没啥问题，当时当处于某种特殊的情况的时候，就会出现一些问题，比如：

因为sigmoid本身的函数值是恒大于0的，我们假设有如下情况：

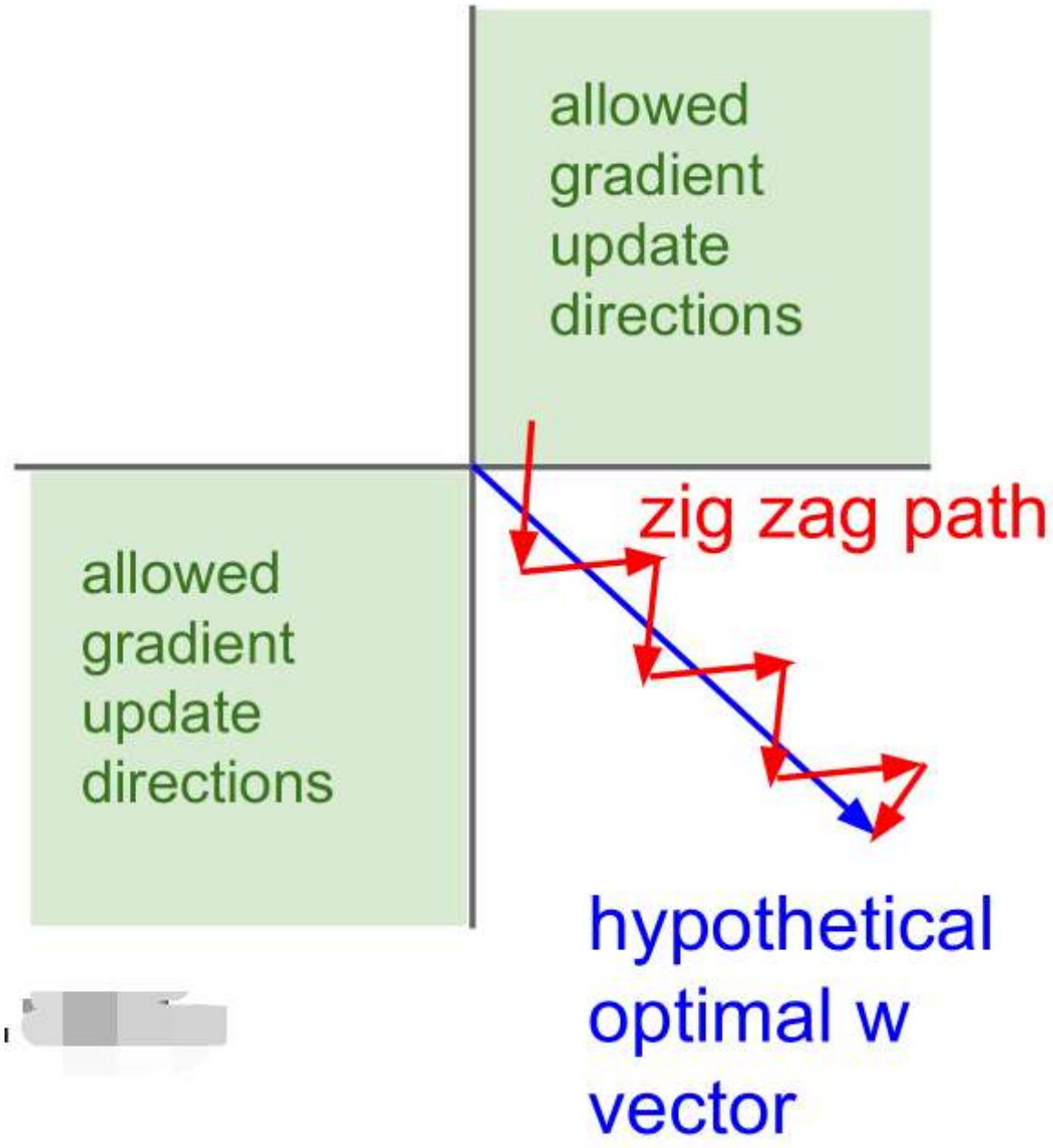
$$f = \sum w^T x + b$$

$$L = \sigma(f)$$

此时的对W的梯度计算应该是：

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial w}$$

因为公式的后半部分，f对w的偏导是x，前半部分就是sigmoid的导数。很明显，上式的符号直接又后半部分，也就是x决定。此时就会出现下图的梯度分布：

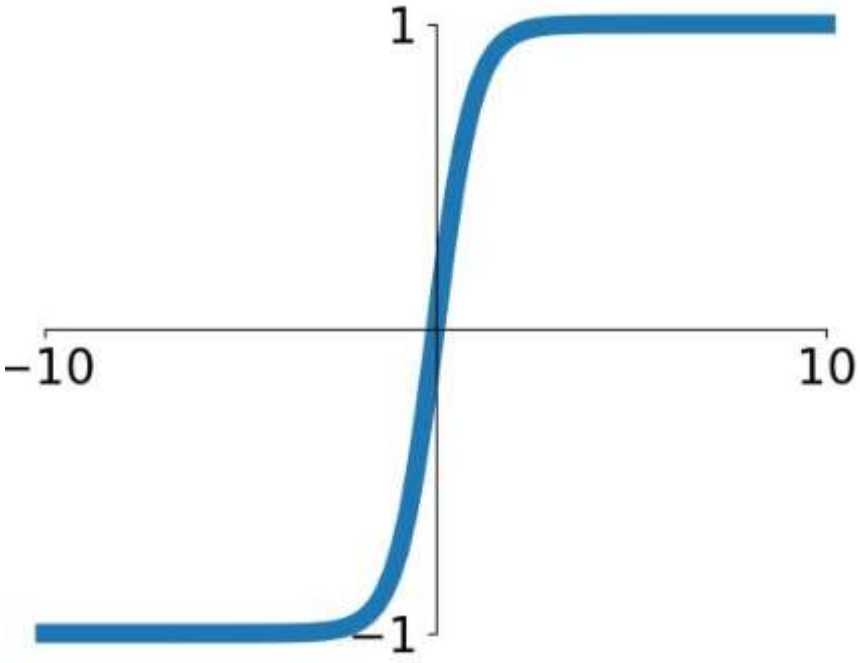


这个时候去随梯度变化的方向只有第一象限和第三象限。这样假设说要去你和上图蓝色箭头的方向的w，就会走红色的折线，这样的效率是非常非常低的。

(3) exp() is a bit compute expensive

这个是必然，没啥问题

2. tanh(x)



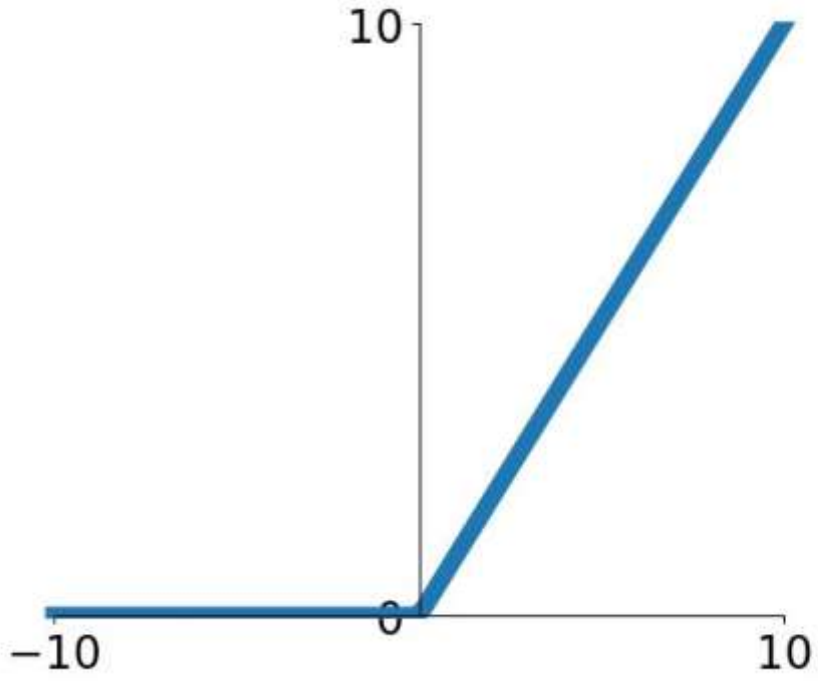
tanh(x)

- Squashes numbers to range [-1, 1]
- zero-centered(nicer than Sigmoid)
- still kill the gradients saturated :(

3. ReLU

计算表达式：

$$f(x) = max\,(0, x)$$



ReLU
(Rectified Linear Unit)

特点:

- Does not saturate (in +region) —— 在正数部分不会出现饱和梯度损失的情况
- Very computationally efficient —— 计算的代价相当低
- Converges much faster than sigmoid/tanh in practice (e.g. 6x) —— 收敛的比前两位快好多
- Actually more biologically plausible than sigmoid —— 比sigmoid更有神经合理性

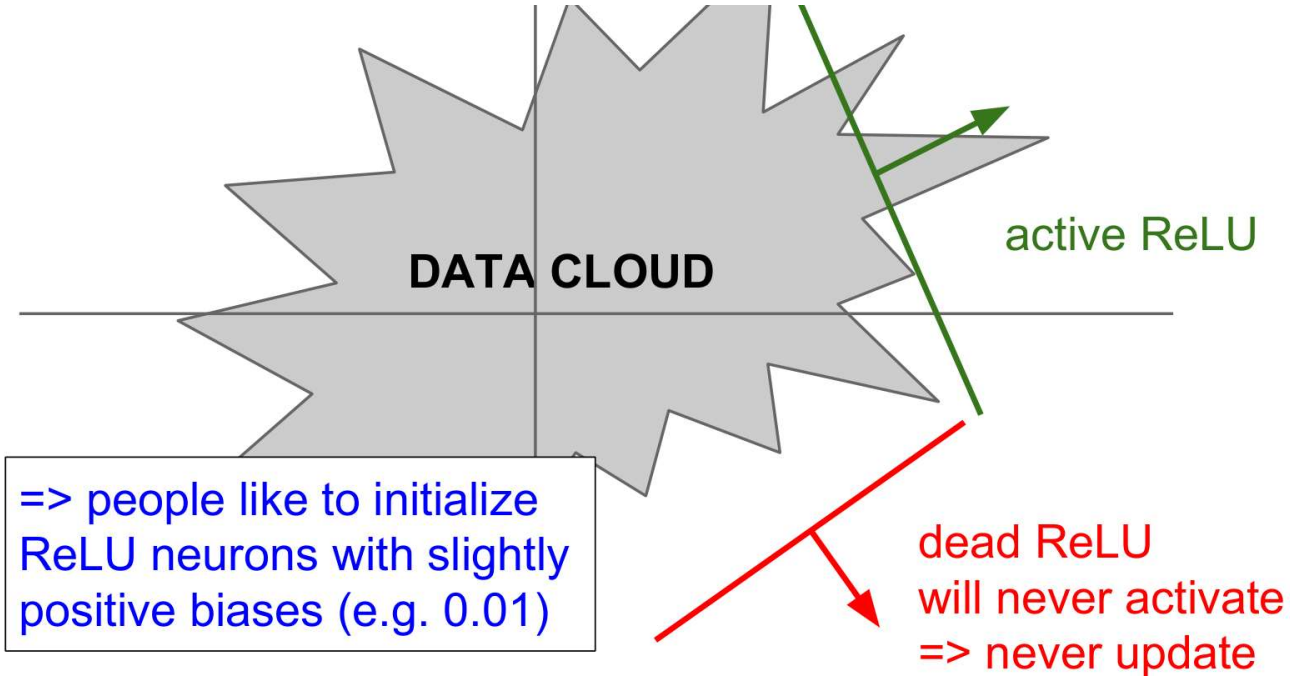
问题:

- Not zero-centered output —— relu的均值仍然不是0，这是个问题
- An annoyance: it still kills gradients when x is less than or equal to zero —— 当x的取值小于0的时候，仍然存在着梯度消失的情况，让人很烦

Dead ReLU:

根据上面的第二个问题，我们做一些分析:

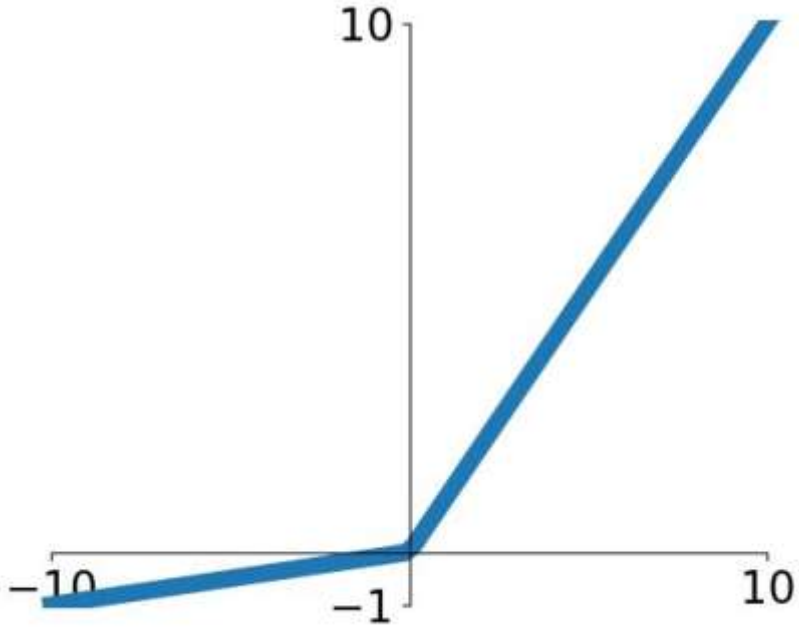
- 首先，ReLU相当于将一半的值给抛弃掉了，这存在着一些的问题，但是这个是可以解决的，解决的方法就是设置一个好的权重初始值，并且该初始值必须要大于0，像下图所说的，一半是取比0稍微大一点的数。
- 另外，这里也暗示了另一个东西，就是你的学习率不能设的特别大，这样存在着根据梯度递减然后一次性小于0而被舍弃的情况，这也是一个无法弥补的错误，所以一般的学习率就要设的比较小，以期可以比较好的拟合。



4. ReLU的变体

(1) Leaky ReLU

$$f(x) = max\,(0.01x, x)$$



特点：

- Does not saturate —— 无论是在正还是负的部分都不会出现饱和梯度消失的情况，比ReLU好
- Computationally efficient —— 计算起来比较简单高效
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x) —— 拟合比这俩快乐好多倍，在实际应用的过程中
- will not “die”. —— 与第二点一样

(2) PReLU

$$f(x) = max\left(\alpha x, x\right)$$

这里就将 alpha 当成超参数处理就可以，在选取超参的时候加以选择就可以了。

(3) ELU(Exponential Linear Units)

$$f(x) = \begin{cases} x, & if\ x > 0 \\ \alpha(e^x - 1), & if\ x \leq 0 \end{cases}$$

好处：

- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

坏处：

- Computation requires exp() —— 这个缺点是必然的

5. Maxout "Neuron"

$$f(x) = max\left(w_1^T x + b_1, w_2^T x + b_2\right)$$

优点：

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

缺点：

- doubles the number of parameters/neuron :(

个人理解：

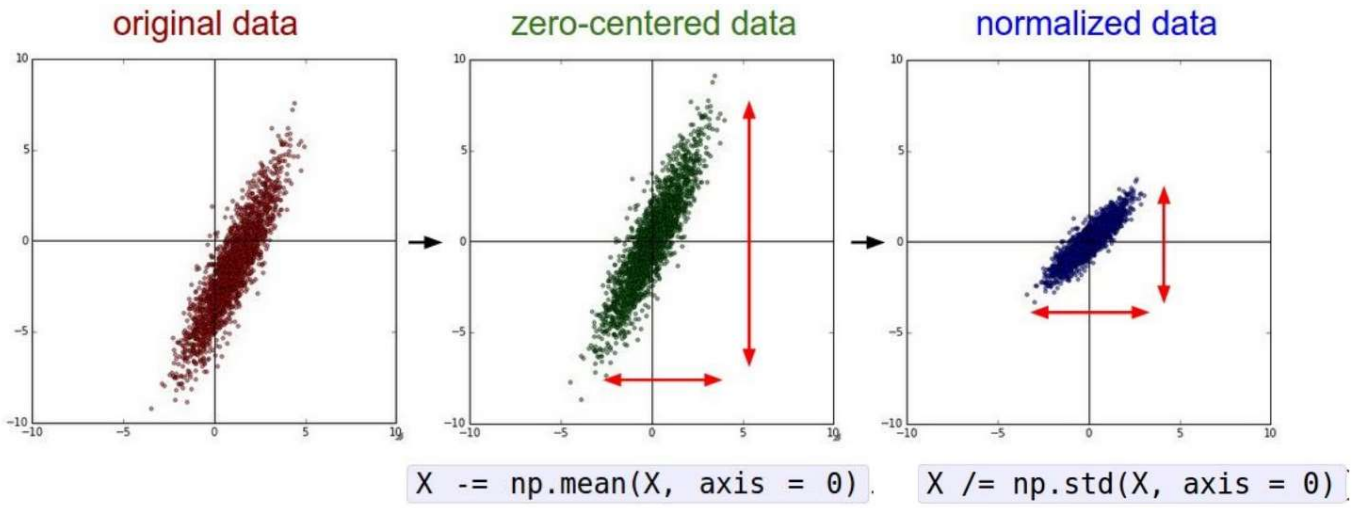
其实maxout这个激活函数的主要意思就是 —— 他相当于在hidden layer 下又加了好几层hidden layer，然后算出来这些值选出最大的。这样的多条线可以拟合出任意函数幂的凸函数，用于拟合是再好不过了。

In practice：在实际应用中注意一下几点

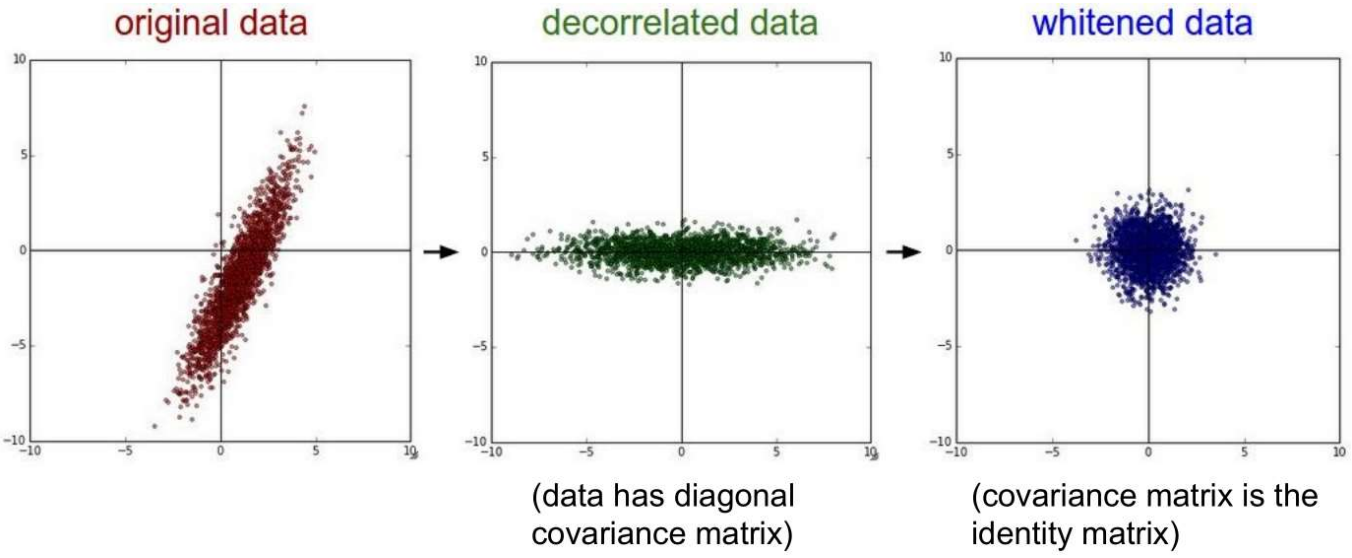
- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’ t expect much
- Don’ t use sigmoid

二、Data Preprocessing(数据的预处理)

第一种方式：均值0化和标准化(zero-centered and normalized)



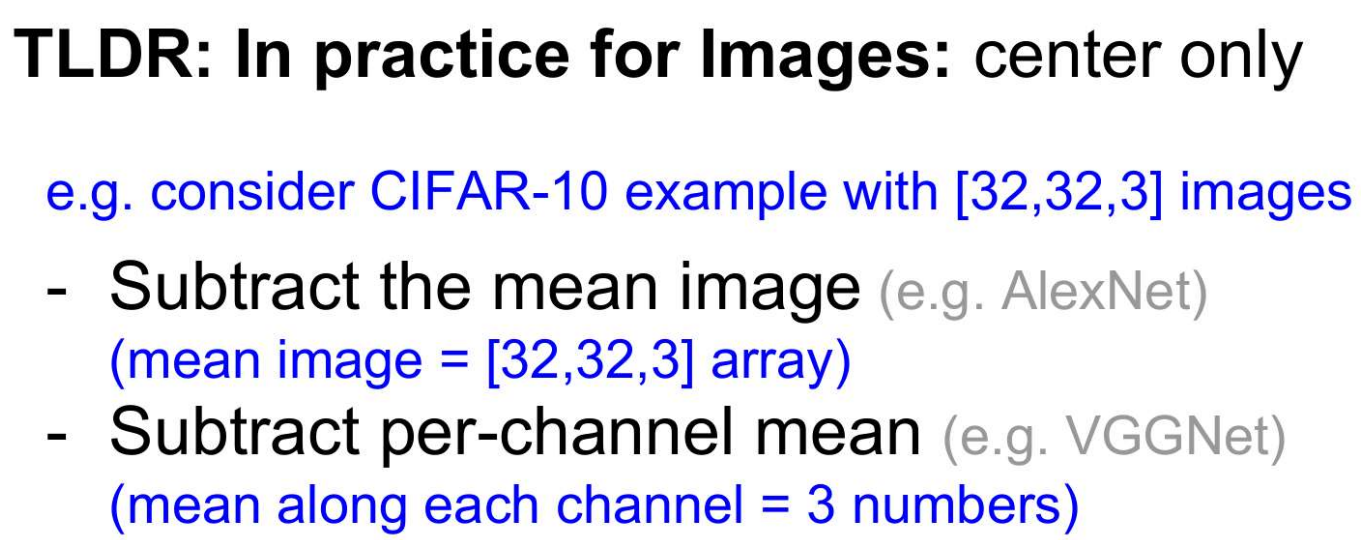
第二种方式：数据降维 —— PCA 和 Whitening



其中PCA的主要方法是：**n维的输入信息重新转换为k维的数据，舍弃掉方差信息影响相当小的n-k维，从而得到最终的结果**

Whitening的主要方法是：**将相邻的含有冗余信息的项简化掉**

总结:



其中，输入信息的最后维度是3其实是RGB三个维度

三、Weight Initialization(权重初始化)

下面考虑三种权重初始化的情况:

(1) 权重初始值全为0:

这个不用说，所有的神经元都将会一样，因为大家都是0，再咋变化都一个样。

(2) 权重初始值较小的情况:

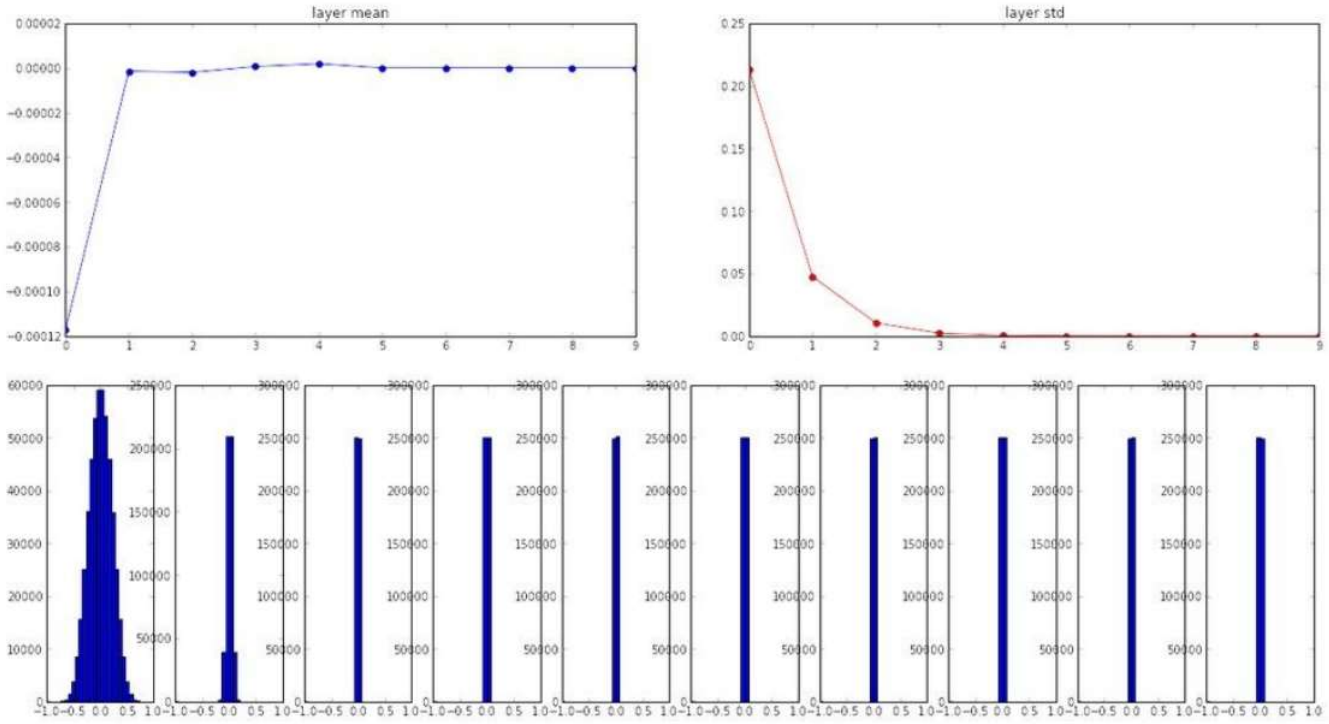
比如:

```
W = 0.01 * np.random.randn(D, H)
```

这个其实在小网络里面的表现比较好，因为毕竟层数少，梯度的计算也比较简单，不会传递很多次。

但是这个在DNN里面就很难说了。

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



上图就是一个示例。当我们对于初始值的设定太小的时候，均值的计算就不必说了，必然是逐渐减小至0，梯度因为Loss对W的梯度计算式的结果是x——输入值，x又因为w太小而变得太小，这样就使得整体的梯度就变得非常小。这就很难受了。

(3) 权重初始值较大的情况

```
W = 1 * np.random.randn(D, H)
```

参考前面，如果权重值太大，那么激活函数的输入值就会很大，对于某些函数，就会出现饱和的情况。

Xavier initialization

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in)
```

一般都这样设置，也比较合理。

但是在激活函数是ReLU的时候就会出现问题，这样就把上面的表达式改一下：

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in / 2)
```

这样主要是由于ReLU会让一般的神经元die掉

四、Batch Normalization(批量归一化)

基本公式：

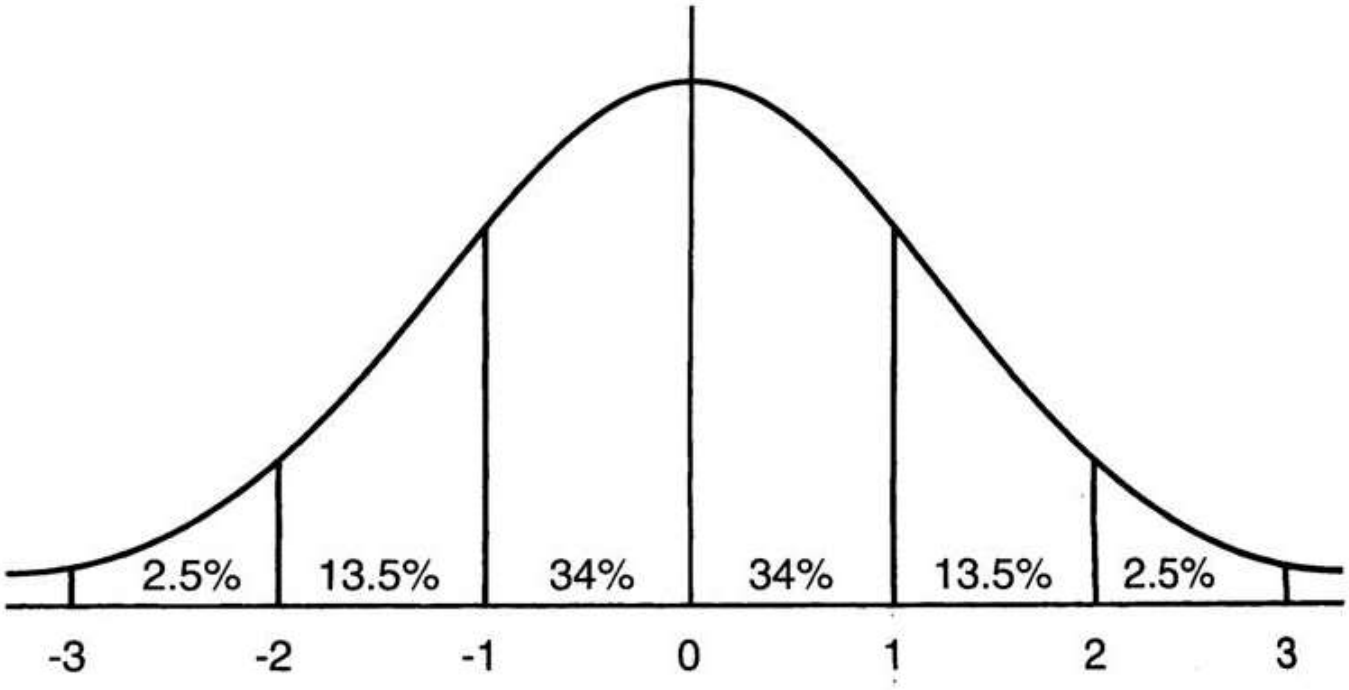
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

啥个意思呢？就是减去均值除以标准差，得到的就是一个均值为0的标准正态分布 —— 高斯分布。

那为啥要这么做呢？这就有门道了。

批量归一化原因：

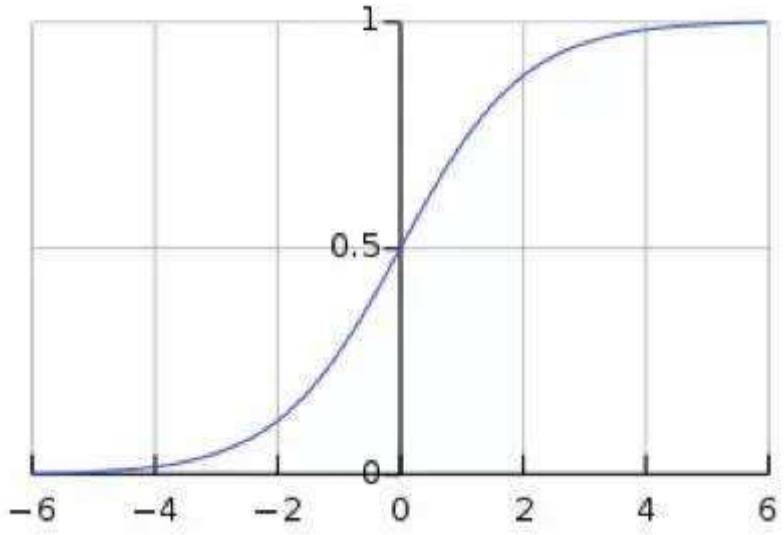
- 进入DL时代，神经网络越来越深，这时候会发现，越来越难以训练到拟合，训练的时间越来越长，这样就让人很烦。
- 同时又发现，高斯分布的输入拟合的就比其他的分布的数据快很多
- 高斯分布的曲线有以下特点：



- 梯度与学习率决定了拟合的快慢。

核心要点：

基于上面的原因，我们再结合Sigmoid这样一个标准函数来看一下：



上面是Sigmoid的曲线，梯度我就不用去画图了。通过这个曲线图，其实我们就可以看出来一些端倪了。

- 首先，大多数的激活函数，如tanh是类似于Sigmoid的，梯度趋势是一样的，那么，是否可以这样说，如果输入的数据尽可能的集中于zero，那么在大梯度下应该可以拟合的更快一些。
- 其次，高斯分布是最简单最容易达到上述所说的一种数据分布形式。
- 另外，深度网络训练慢主要是因为，后面的几层网络的输入与整体网络输入是不一样的，分布更是不可估计，那么就很容易出现数据接近激活函数饱和值的情况，这才是拟合较慢的主要原因。
- 这样，我们就在每一层激活函数前加一个BN(Batch Normalization层)，啥事都搞定。

五、Babysitting the Learning Process(观察训练过程并修改)

就是根据数据调参，不再赘述了。

六、Hyperparameter Optimization(超参数优化)

不再赘述了，大概了解一下就好，似乎不太重点

七、总结

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier init)
- Batch Normalization (use)
- Babysitting the Learning process
- Hyperparameter Optimization (random sample hyperparams, in log space when appropriate)