

L1 和 L2

作用

优化方法:

L1 坐标下降, LARS 角回归

优化方法

BGD

即 Batch Gradient Descent. 在训练中,每一步迭代都使用训练集的所有内容. 也就是说,利用现有参数对训练集中的每一个输入生成一个估计输出 \hat{y}_i ,然后跟实际输出 y_i 比较,统计所有误差,求平均以后得到平均误差,以此来作为更新参数的依据.

具体实现:

需要:学习速率 ϵ , 初始参数 θ

每步迭代过程:

1. 提取训练集中的所有内容 $\{x_1, \dots, x_n\}$, 以及相关的输出 y_i
2. 计算梯度和误差并更新参数:

$$\begin{aligned}\hat{g} &\leftarrow +\frac{1}{n} \nabla_{\theta} \sum_i L(f(x_i; \theta), y_i) \\ \theta &\leftarrow \theta - \epsilon \hat{g}\end{aligned}$$

优点:

由于每一步都利用了训练集中的所有数据,因此当损失函数达到最小值以后,能够保证此时计算出的梯度为 0,换句话说,就是能够收敛.因此,使用 BGD 时不需要逐渐减小学习速率 ϵ_k

缺点:

由于每一步都要使用所有数据,因此随着数据集的增大,运行速度会越来越慢.

SGD

SGD 全名 stochastic gradient descent, 即随机梯度下降。不过这里的 SGD 其实跟 MBGD(minibatch gradient descent)是一个意思,即随机抽取一批样本,以此为根据

来更新参数.

具体实现:

需要:学习速率 ϵ , 初始参数 θ

每步迭代过程:

1. 从训练集中的随机抽取一批容量为 m 的样本 $\{x_1, \dots, x_m\}$, 以及相关的输出 y_i
2. 计算梯度和误差并更新参数:

$$\begin{aligned}\hat{g} &\leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(x_i; \theta), y_i) \\ \theta &\leftarrow \theta - \epsilon \hat{g}\end{aligned}$$

优点:

训练速度快,对于很大的数据集,也能够以较快的速度收敛.

缺点:

由于是抽取,因此不可避免的,得到的梯度肯定有误差.因此学习速率需要逐渐减小.否则模型无法收敛

因为误差,所以每一次迭代的梯度受抽样的影响比较大,也就是说梯度含有比较大的噪声,不能很好的反映真实梯度.

学习速率该如何调整:

那么这样一来, ϵ 如何衰减就成了问题.如果要保证 SGD 收敛,应该满足如下两个要求:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty$$

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$

而在实际操作中,一般是进行线性衰减:

$$\begin{aligned}\epsilon_k &= (1 - \alpha)\epsilon_0 + \alpha\epsilon_{\tau} \\ \alpha &= \frac{k}{\tau}\end{aligned}$$

其中 ϵ_0 是初始学习率, ϵ_{τ} 是最后一次迭代的学习率. τ 自然代表迭代次数.一般来说, ϵ_{τ} 设为 ϵ_0 的 1% 比较合适.而 τ 一般设为让训练集中的每个数据都输入模型上百次比较合适.那么初始学习率 ϵ_0 怎么设置呢?书上说,你先用固定的学习速率迭代 100 次,找出效果最好的学习速率,然后 ϵ_0 设为比它大一点就可以了.

Momentum

上面的 SGD 有个问题,就是每次迭代计算的梯度含有比较大的噪音. 而 Momentum 方法可以比较好的缓解这个问题,尤其是在面对小而连续的梯度但是含有很多噪音的时候,可以很好的加速学习.Momentum 借用了物理中的动量概念,即前几次的梯度也会参与运算.为了表示动量,引入了一个新的变量 v (velocity). v 是之前的梯度的累加,但是每回合都有一定的衰减.

具体实现:

需要:学习速率 ϵ , 初始参数 θ , 初始速率 v , 动量衰减参数 α

每步迭代过程:

1. 从训练集中的随机抽取一批容量为 m 的样本 $\{x_1, \dots, x_m\}$, 以及相关的输出 y_i
2. 计算梯度和误差并更新速度 v 和参数 θ :

$$\begin{aligned}\hat{g} &\leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(x_i; \theta), y_i) \\ v &\leftarrow \alpha v - \epsilon \hat{g} \\ \theta &\leftarrow \theta - v\end{aligned}$$

其中参数 α 表示每回合速率 v 的衰减程度. 同时也可以推断得到,如果每次迭代得到的梯度都是 g ,那么最后得到的 v 的稳定值为

$$\frac{\epsilon \|g\|}{1 - \alpha}$$

也就是说,Momentum 最好情况下能够将学习速率加速 $\frac{1}{1-\alpha}$ 倍.一般 α 的取值有 0.5,0.9,0.99 这几种.当然,也可以让 α 的值随着时间而变化,一开始小点,后来再加大.不过这样一来,又会引进新的参数.

特点:

前后梯度方向一致时,能够加速学习

前后梯度方向不一致时,能够抑制震荡

Nesterov Momentum

这是对之前的 Momentum 的一种改进,大概思路就是,先对参数进行估计,然后使用估计后的参数来计算误差

具体实现:

需要:学习速率 ϵ , 初始参数 θ , 初始速率 v , 动量衰减参数 α

每步迭代过程:

1. 从训练集中的随机抽取一批容量为 m 的样本 $\{x_1, \dots, x_m\}$, 以及相关的输出 y_i
2. 计算梯度和误差并更新速度 v 和参数 θ :

$$\begin{aligned}\hat{g} &\leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(x_i; \theta), y_i) \\ v &\leftarrow \alpha v - \epsilon \hat{g} \\ \theta &\leftarrow \theta + v\end{aligned}$$

注意在估算 \hat{g} 的时候, 参数变成了 $\theta + \alpha v$ 而不是之前的 θ

AdaGrad

AdaGrad 可以自动变更学习速率, 只是需要设定一个全局的学习速率 ϵ , 但是这并非是实际学习速率, 实际的速率是与以往参数的模之和的开方成反比的. 也许说起来有点绕口, 不过用公式来表示就直白的多:

$$\epsilon_n = \frac{\epsilon}{\delta + \sqrt{\sum_{i=1}^{n-1} g_i \odot g_i}}$$

其中 δ 是一个很小的常亮, 大概在 10^{-7} , 防止出现除以 0 的情况.

具体实现:

需要: 全局学习速率 ϵ , 初始参数 θ , 数值稳定量 δ

中间变量: 梯度累计量 r (初始化为 0)

每步迭代过程:

1. 从训练集中的随机抽取一批容量为 m 的样本 $\{x_1, \dots, x_m\}$, 以及相关的输出 y_i
2. 计算梯度和误差, 更新 r , 在根据 r 和梯度计算参数更新量:

$$\begin{aligned}\hat{g} &\leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(x_i; \theta), y_i) \\ r &\leftarrow r + \hat{g} \odot \hat{g} \\ \Delta\theta &= -\frac{\epsilon}{\delta + \sqrt{r}} \odot \hat{g} \\ \theta &\leftarrow \theta + \Delta\theta\end{aligned}$$

优点:

能够实现学习率的自动更改。如果这次梯度大,那么学习速率衰减的就快一些;如果这次梯度小,那么学习速率衰减的就慢一些。

缺点:

任然要设置一个变量 ϵ

经验表明，在普通算法中也许效果不错，但在深度学习中，深度过深时会造成训练提前结束。

RMSProp

RMSProp 通过引入一个衰减系数，让 r 每回合都衰减一定比例，类似于 Momentum 中的做法。

具体实现:

需要:全局学习速率 ϵ , 初始参数 θ , 数值稳定量 δ , 衰减速率 ρ

中间变量: 梯度累计量 r (初始化为 0)

每步迭代过程:

1. 从训练集中的随机抽取一批容量为 m 的样本 $\{x_1, \dots, x_m\}$, 以及相关的输出 y_i
2. 计算梯度和误差,更新 r ,在根据 r 和梯度计算参数更新量:

$$\begin{aligned}\hat{g} &\leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(x_i; \theta), y_i) \\ r &\leftarrow \rho r + (1 - \rho) \hat{g} \odot \hat{g} \\ \Delta\theta &= -\frac{\epsilon}{\delta + \sqrt{r}} \odot \hat{g} \\ \theta &\leftarrow \theta + \Delta\theta\end{aligned}$$

优点:

相比于 AdaGrad,这种方法很好的解决了深度学习中过早结束的问题
适合处理非平稳目标，对于 RNN 效果很好

缺点:

又引入了新的超参，衰减系数 ρ

依然依赖于全局学习速率

RMSProp with Nesterov Momentum

当然，也有将 RMSProp 和 Nesterov Momentum 结合起来的

具体实现:

需要:全局学习速率 ϵ , 初始参数 θ , 初始速率 v , 动量衰减系数 α , 梯度累计量衰减速率 ρ

中间变量: 梯度累计量 r (初始化为 0)

每步迭代过程:

1. 从训练集中的随机抽取一批容量为 m 的样本 $\{x_1, \dots, x_m\}$, 以及相关的输出 y_i
2. 计算梯度和误差,更新 r ,在根据 r 和梯度计算参数更新量:

$$\tilde{\theta} \leftarrow \theta + \alpha v$$

$$\hat{g} \leftarrow +\frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(x_i; \tilde{\theta}), y_i)$$

$$r \leftarrow \rho r + (1 - \rho) \hat{g} \odot \hat{g}$$

$$v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot \hat{g}$$

$$\theta \leftarrow \theta + v$$

Adam

Adam(Adaptive Moment Estimation)本质上是带有动量项的 RMSprop, 它利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率。Adam 的优点主要在于经过偏置校正后, 每一次迭代学习率都有个确定范围, 使得参数比较平稳。

具体实现:

需要:步进值 ϵ , 初始参数 θ , 数值稳定量 δ , 一阶动量衰减系数 ρ_1 , 二阶动量衰减系数 ρ_2

其中几个取值一般为: $\delta = 10^{-8}, \rho_1 = 0.9, \rho_2 = 0.999$

中间变量: 一阶动量 s , 二阶动量 r ,都初始化为 0

每步迭代过程:

1. 从训练集中的随机抽取一批容量为 m 的样本 $\{x_1, \dots, x_m\}$, 以及相关的输出 y_i
2. 计算梯度和误差,更新 r ,在根据 r 和梯度计算参数更新量:

$$\hat{g} \leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(x_i; \theta), y_i)$$

$$s \leftarrow \rho_1 s + (1 - \rho_1) g$$

$$r \leftarrow \rho_2 r + (1 - \rho_2) g \odot g$$

$$\hat{s} \leftarrow \frac{s}{1 - \rho_1}$$

$$\hat{r} \leftarrow \frac{r}{1 - \rho_2}$$

$$\theta \leftarrow \theta + \Delta \theta$$

参考：

<http://blog.csdn.net/u014595019/article/details/52989301>

http://www.360doc.com/content/17/0323/08/1489589_639370019.shtml

激活函数和损失函数

激活函数

关于激活函数，首先要搞清楚的问题是，激活函数是什么，有什么用？不用激活函数可不可以？答案是不可以。激活函数的主要作用是提供网络的非线性建模能力。如果没有激活函数，那么该网络仅能够表达线性映射，此时即便有再多的隐藏层，其整个网络跟单层神经网络也是等价的。因此也可以认为，只有加入了激活函数之后，深度神经网络才具备了分层的非线性映射学习能力。那么激活函数应该具有什么样的性质呢？

可微性：当优化方法是基于梯度的时候，这个性质是必须的。

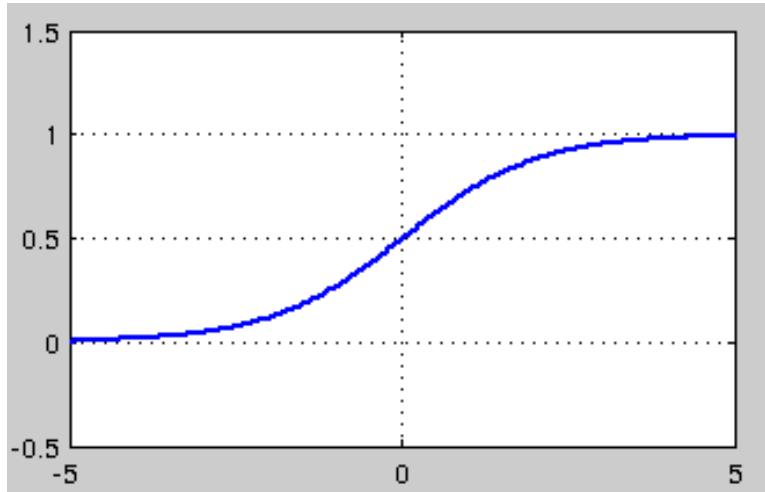
单调性：当激活函数是单调的时候，单层网络能够保证是凸函数。

输出值的范围：当激活函数输出值是有限的时候，基于梯度的优化方法会更加稳定，因为特征的表示受有限权值的影响更显著；当激活函数的输出是无限的时候，模型的训练会更加高效，不过在这种情况下，一般需要更小的 learning rate

从目前来看，常见的激活函数多是分段线性和具有指数形状的非线性函数

sigmoid

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



sigmoid 是使用范围最广的一类激活函数，具有指数函数形状，它在物理意义上最为接近生物神经元。此外， $(0, 1)$ 的输出还可以被表示作概率，或用于输入的归一化，代表性的如 **Sigmoid** 交叉熵损失函数。

然而，**sigmoid** 也有其自身的缺陷，最明显的就是饱和性。从上图可以看到，其两侧导数逐渐趋近于 0

$$\lim_{x \rightarrow \infty} f'(x) = 0$$

具有这种性质的称为软饱和激活函数。具体的，饱和又可分为左饱和与右饱和。与软饱和对应的是硬饱和，即

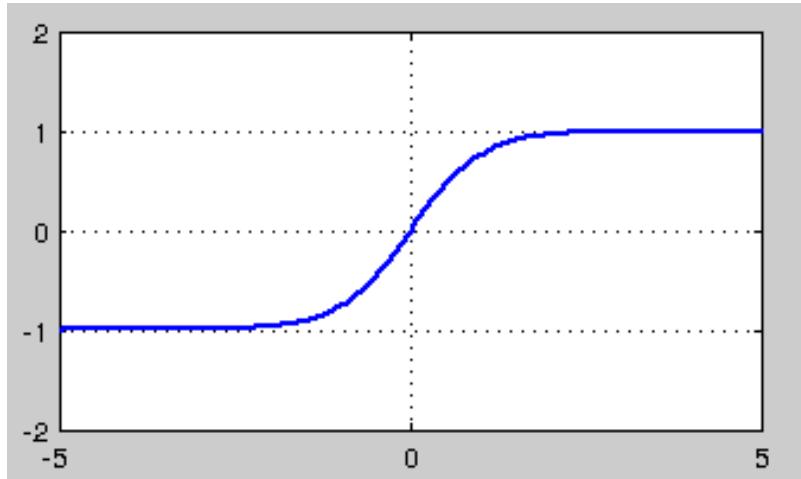
$$f'(x) = 0, \text{ 当 } |x| > c, \text{ 其 } c \text{ 为常数}$$

sigmoid 的软饱和性，使得深度神经网络在二三十年里一直难以有效的训练，是阻碍神经网络发展的重要原因。具体来说，由于在后向传递过程中，**sigmoid** 向下传导的梯度包含了一个 $f'(x)$ 因子 (**sigmoid** 关于输入的导数)，因此一旦输入落入饱和区， $f'(x)$ 就会变得接近于 0，导致了向底层传递的梯度也非常小。此时，网络参数很难得到有效训练。这种现象被称为梯度消失。一般来说，**sigmoid** 网络在 5 层之内就会产生梯度消失现象

此外，**sigmoid** 函数的输出均大于 0，使得输出不是 0 均值，这称为偏移现象，这会导致后一层的神经元将得到上一层输出的非 0 均值的信号作为输入。

Tanh

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

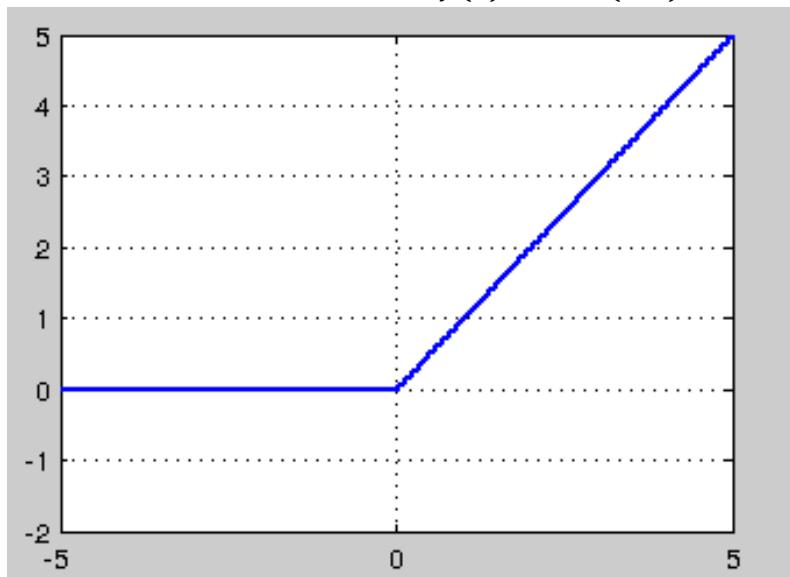


\tanh 也是一种非常常见的激活函数。与 sigmoid 相比，它的输出均值是 0，使得其收敛速度要比 sigmoid 快，减少迭代次数。然而，从途中可以看出， \tanh 一样具有软饱和性，从而造成梯度消失。

ReLU, P-ReLU, Leaky-ReLU

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

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

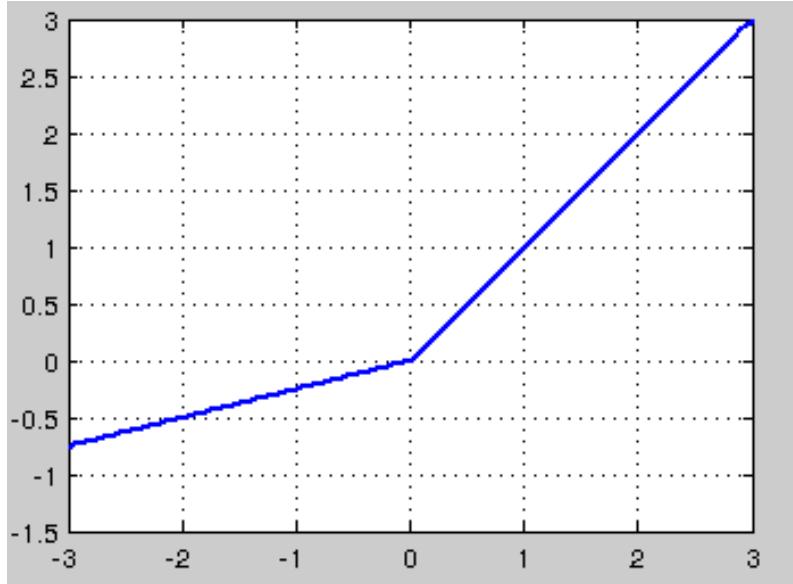


ReLU 的全称是 Rectified Linear Units，是一种后来才出现的激活函数。可以看到，当 $x < 0$ 时，ReLU 硬饱和，而当 $x > 0$ 时，则不存在饱和问题。所以，ReLU 能够在 $x > 0$ 时保持梯度不衰减，从而缓解梯度消失问题。这让我们能够直接以监督的方式训练深度神经网络，而无需依赖无监督的逐层预训练。

然而，随着训练的推进，部分输入会落入硬饱和区，导致对应权重无法更新。这种现象被称为“神经元死亡”。与 sigmoid 类似，ReLU 的输出均值也大于 0，偏移现象和神经元死亡会共同影响网络的收敛性。

针对在 $x < 0$ 的硬饱和问题，我们对 ReLU 做出相应的改进，使得

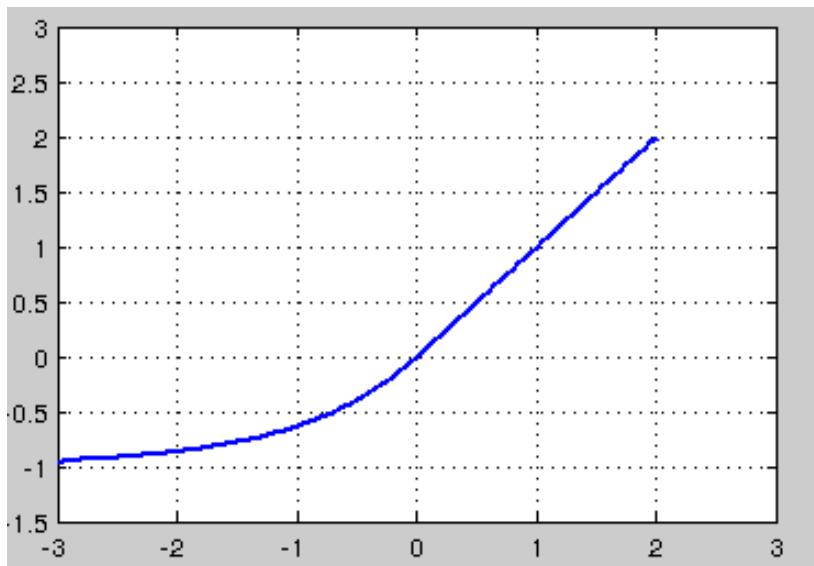
$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases}$$



这就是 Leaky-ReLU，而 P-ReLU 认为， α 也可以作为一个参数来学习，原文献建议初始化 α 为 0.25，不采用正则。

ELU

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



融合了 sigmoid 和 ReLU，左侧具有软饱和性，右侧无饱和性。右侧线性部分使得 ELU 能够缓解梯度消失，而左侧软饱能够让 ELU 对输入变化或噪声更鲁棒。ELU 的输出均值接近于零，所以收敛速度更快。在 ImageNet 上，不加 Batch Normalization 30 层以上的 ReLU 网络会无法收敛，PReLU 网络在 MSRA 的 Fan-in (caffe) 初始化下会发散，而 ELU 网络在 Fan-in/Fan-out 下都能收敛

maxout

$$f(x) = \max(w_1^T x + b_1, w_2^T + b_2, \dots, w_n^T + b_n)$$

在我看来，这个激活函数有点大一统的感觉，因为 maxout 网络能够近似任意连续函数，且当 $w_2, b_2, \dots, w_n, b_n$ 为 0 时，退化为 ReLU。Maxout 能够缓解梯度消失，同时又规避了 ReLU 神经元死亡的缺点，但增加了参数和计算量。

损失函数

在之前的内容中，我们用的损失函数都是平方差函数，即

$$C = \frac{1}{2}(\alpha - y)^2$$

其中 y 是我们期望的输出， α 为神经元的实际输出 ($\alpha = \sigma(Wx + b)$)。也就是说，当神经元的实际输出与我们的期望输出差距越大，代价就越高。想法非常的好，然

而在实际应用中，我们知道参数的修正是与 $\frac{\partial C}{\partial W}$ 和 $\frac{\partial C}{\partial b}$ 成正比的，而根据

$$\begin{aligned}\frac{\partial C}{\partial W} &= (\alpha - y)\sigma'(\alpha)x^T \\ \frac{\partial C}{\partial b} &= (\alpha - y)\sigma'(\alpha)\end{aligned}$$

我们发现其中都有 $\sigma'(\alpha)$ 这一项。因为 sigmoid 函数的性质，导致 $\sigma'(z)$ 在 z 取大部分值时会造成饱和现象，从而使得参数的更新速度非常慢，甚至会造成离期望值越远，更新越慢的现象。那么怎么克服这个问题呢？我们想到了交叉熵函数。我们知道，熵的计算公式是

$$H(y) = - \sum_i y_i \log(y_i)$$

而在实际操作中，我们并不知道 y 的分布，只能对 y 的分布做一个估计，也就是算得的 α 值，这样我们就能够得到用 α 来表示 y 的交叉熵

$$H(y, \alpha) = - \sum_i y_i \log(\alpha_i)$$

如果有多个样本，则整个样本的平均交叉熵为

$$H(y, \alpha) = - \frac{1}{n} \sum_n \sum_i y_{i,n} \log(\alpha_{i,n})$$

其中 n 表示样本编号， i 表示类别编。如果用于 logistic 分类，则上式可以简化成

$$H(y, \alpha) = - \frac{1}{n} \sum_i y \log(\alpha) + (1 - y) \log(1 - \alpha)$$

与平方损失函数相比，交叉熵函数有个非常好的特质，

$$H' = \frac{1}{n} \sum_n (\alpha_n - y_n) = \frac{1}{n} \sum_n (\sigma(z_n) - y_n)$$

可以看到其中没有了 σ' 这一项，这样一来也就不会受到饱和性的影响了。当误差大的时候，权重更新就快，当误差小的时候，权重的更新就慢。这是一个很好的性质。

PCA, LDA, TSNE

<http://www.cnblogs.com/LeftNotEasy/archive/2011/01/08/lda-and-pca-machine-learning.html>

SVD

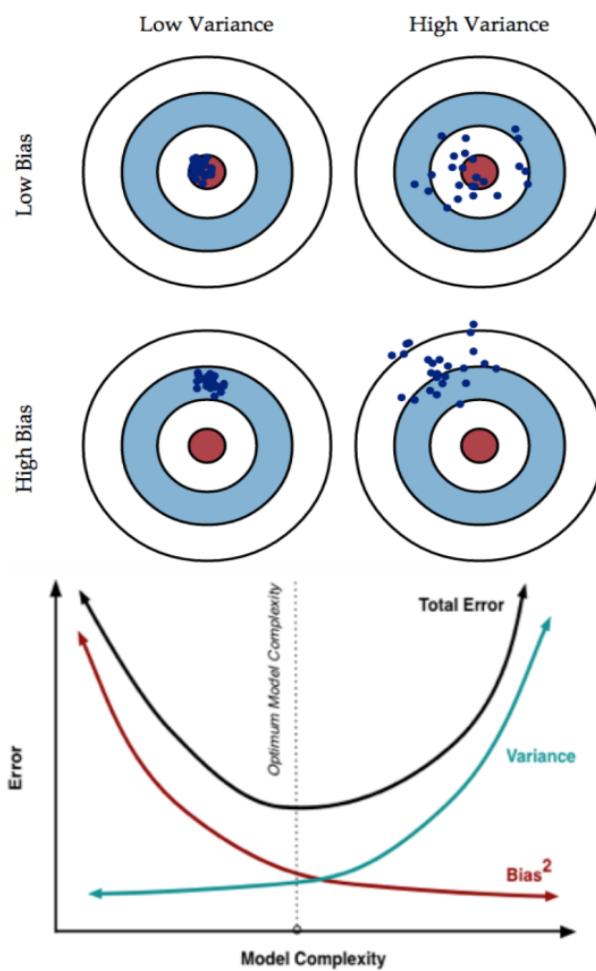
<http://www.cnblogs.com/LeftNotEasy/archive/2011/01/19/svd-and-applications.html>

有序数组的交集（要有代码）

F1, ROC, recall, precision

Bias(偏差), variance(方差) 以及 偏差 - 方差权衡
(bias-variance tradeoff)

准与确



$$E[(y - \hat{f}(x))^2] = \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + \sigma^2$$

$$\text{Bias}[\hat{f}(x)] = E[\hat{f}(x) - f(x)]$$

$$\text{Var}[\hat{f}(x)] = E[\hat{f}(x)^2] - E[\hat{f}(x)]^2$$

σ 是噪声， $f(x)$ 是理论上的函数， $\hat{f}(x)$ 是我们选取的函数。

首先明确一点，Bias 和 Variance 是针对 Generalization (一般化，泛化) 来说的。在机器学习中，我们用训练数据集去训练（学习）一个 model (模型)，通常的做法是定义一个 Loss function (误差函数)，通过将这个 Loss (或者叫 error) 的最小化过程，来提高模型的性能 (performance)。然而我们学习一个模型的目的是为了解决实际的问题(或者说是训练数据集这个领域 (field) 中的一般化问题)，单纯地将训练数据集的 loss 最小化，并不能保证在解决更一般的问题时模型仍然是最优，甚至不能保证模型是可用的。这个训练数据集的 loss 与一般化的数据集的 loss 之间的差异就叫做 generalization error。而 generalization error 又可以细分为 Bias 和 Variance 两个部分。

bias 描述的是根据样本拟合出的模型的输出预测结果的期望与样本真实结果的差距，简单讲，就是在样本上拟合的好不好。要想在 bias 上表现好，low bias，就是复杂化模型，增加模型的参数，但这样容易过拟合 (overfitting)，过拟合对应上图是 high variance，点很分散。low bias 对应就是点都打在靶心附近，所以瞄的是准的，但手不一定稳。

variance 描述的是样本上训练出来的模型在测试集上的表现，要想在 variance 上表现好，low variance，就要简化模型，减少模型的参数，但这样容易欠拟合 (unfitting)，欠拟合对应上图是 high bias，点偏离中心。low variance 对应就是点都打的很集中，但不一定是靶心附近，手很稳，但是瞄的不准。

参考：

https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff

决策树如何处理不完整数据

在 xgboost 里，在每个结点上都会将对应变量是缺失值的数据往左右分支各导流一次，然后计算两种导流方案对 Objective 的影响，最后认为对 Objective 降低更明显的方向（左或者右）就是缺失数据应该流向的方向，在预测时在这个结点上将同样变量有缺失值的数据都导向训练出来的方向。

例如，某个结点上的判断条件是 $A > 0$ ，有些数据是 $A \leq 0$ ，有些是 $A > 0$ ，有些数据的 A 是缺失值。那么算法首先忽略带缺失值的数据，像正常情况下一样将前两种数据分别计算并导流到左子树与右子树，然后将带缺失值的数据导向左子树，计算出这时候模型的 Objective_L；接着将带缺失值的数据导向右子树，计算出这时候模型的 Objective_R；最后比较 Objective_L 和 Objective_R。假设 Objective_L 更小，那么在预测时所有变量 A 是缺失值的数据在这个结点上都会被导向左边，当作 $A \leq 0$ 处理。