

# 第二门课 改善深层神经网络：超参数调试、正则化以及优化(Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization)

## 第一周：深度学习的实践层面(Practical aspects of Deep Learning)

### 机器学习基础 (Basic Recipe for Machine Learning)

训练神经网络时，我们需要做出很多决策，例如：

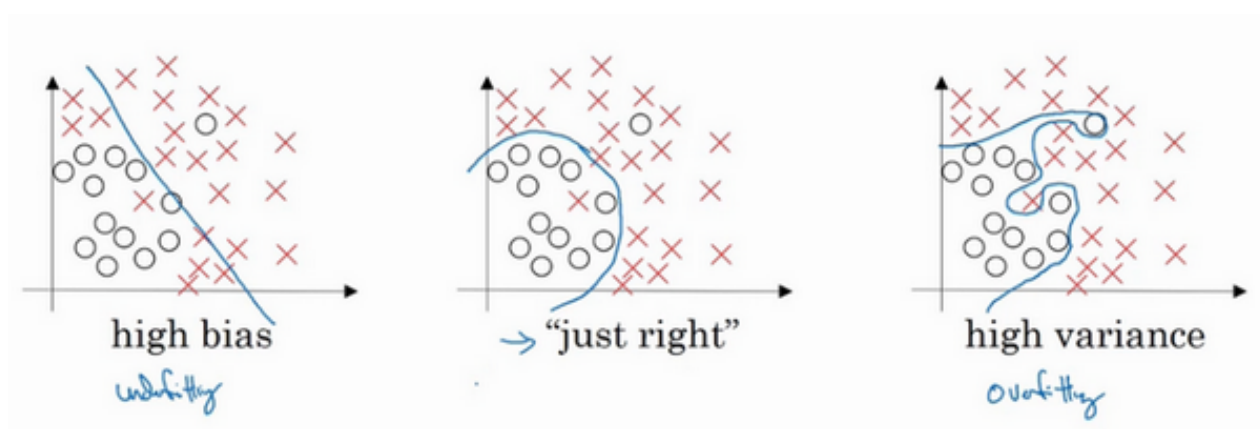
1. 神经网络分多少层
2. 每层含有多少个隐藏单元
3. 学习速率是多少
4. 各层采用哪些激活函数

而这些参数都是未知的，我们一般通过验证集来选择。

机器学习发展的小数据量时代对数据划分：将所有数据三七分，就是人们常说的70%验证集，30%测试集，如果没有明确设置验证集，也可以按照60%训练，20%验证和20%测试集来划分。

大数据时代：将样本分成训练集，验证集和测试集三部分，数据集规模相对较小，适用传统的划分比例，数据集规模较大的，验证集和测试集要小于数据总量的20%或10%。

偏差和方差



如果给这个数据集拟合一条直线，可能得到一个逻辑回归拟合，但它并不能很好地拟合该数据，这是高偏差 (**high bias**) 的情况，我们称为“欠拟合” (**underfitting**) 。

如果我们拟合一个非常复杂的分类器，比如深度神经网络或含有隐藏单元的神经网络，可能就非常适用于这个数据集，但是这看起来也不是一种很好的拟合方式分类器方差较高（**high variance**），数据过度拟合（**overfitting**）。

训练集误差小，而测试集误差大：高方差

训练集误差大，而测试集误差类似：高偏差

训练集误差大，而测试集误差也大得多：高方差&&高偏差

训练集误差小，而测试集误差类似：低方差&&低偏差

初始模型训练完成后。

首先判断算法的偏差高不高：如果偏差较高，试着评估训练集或训练数据的性能。如果偏差的确很高，甚至无法拟合训练集，那么你要做的就是选择一个新的网络，比如含有更多隐藏层或者隐藏单元的网络，或者花费更多时间来训练网络，或者尝试更先进的优化算法。

然后检查方差：采用更多数据或通过正则化来减少过拟合

只要正则适度，通常构建一个更大的网络便可以，在不影响方差的同时减少偏差，而采用更多数据通常可以在不过多影响偏差的同时减少方差。

## 正则化（Regularization）

如果你怀疑神经网络过度拟合了数据，即存在高方差问题，那么最先想到的方法可能是正则化，另一个解决高方差的方法就是准备更多数据。

逻辑回归正则化：

$$\begin{aligned} & \min_{w,b} J(w,b) \quad w \in \mathbb{R}^{n_x}, b \in \mathbb{R} \\ & J(w,b) = \frac{1}{n} \sum_{i=1}^n \ell(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2n} \|w\|_2^2 \\ & \text{L}_2 \text{ regularization} \quad \underline{\|w\|_2^2} = \sum_{j=1}^{n_x} w_j^2 = w^T w \end{aligned}$$

此方法称为L2正则化。因为这里用了欧几里德法线，被称为向量参数w的L2范数。

L1正则化：

$$L_1 \text{ regularization} \quad \frac{\lambda}{2m} \sum_{j=1}^n |w_j| = \frac{\lambda}{2m} \|w\|_1 \quad w \text{ will be sparse}$$

神经网络正则化：

## Neural network

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^m \ell(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^L \|w^{[l]}\|_F^2$$

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2$$

"Frobenius norm"

$$\|\cdot\|_2^2$$

$$w: \begin{pmatrix} n^{[l-1]} & n^{[l]} \end{pmatrix}$$

$$\|\cdot\|_F^2$$

该正则项为：

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$$

$$w^{[l]}: \begin{pmatrix} n^{[l]} & n^{[l-1]} \end{pmatrix}$$

该矩阵范数被称作“弗罗贝尼乌斯范数”，用下标F标注，它表示一个矩阵中所有元素的平方和。

使用该范数实现梯度下降：

首先通过后向传播计算dW

$$dw^{[l]} = (\text{from backprop})$$

$$w^{[l]} := w^{[l]} - \alpha dw^{[l]}$$

$$\frac{\partial J}{\partial w^{[l]}}$$

给dW加上这一项：

$$dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} w^{[l]}$$

$$\rightarrow w^{[l]} := w^{[l]} - \alpha dw^{[l]}$$

"Waffel decay"

$$\frac{\partial J}{\partial w^{[l]}} = dw^{[l]}$$

w更新为：

$$W^{(2)} := W^{(2)} - \alpha \left[ (\text{from backprop}) + \frac{\lambda}{n} W^{(2)} \right]$$

$$= W^{(2)} - \frac{\alpha \lambda}{n} W^{(2)} - \alpha (\text{from backprop})$$

Δ

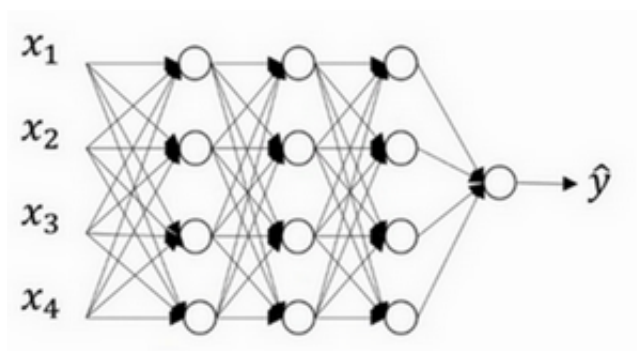
如果正则化设置得足够大，权重矩阵被设置为接近于0的值，直观理解就是把多隐藏单元的权重设为0，于是基本上消除了这些隐藏单元的许多影响。如果是这种情况，这个被大大简化了的神经网络会变成一个很小的网络，小到如同一个逻辑回归单元，可是深度却很大，它会使这个网络从过度拟合的状态更接近高偏差状态。

但是会存在一个中间值lambda，于是会有一个接近“Just Right”的中间状态。

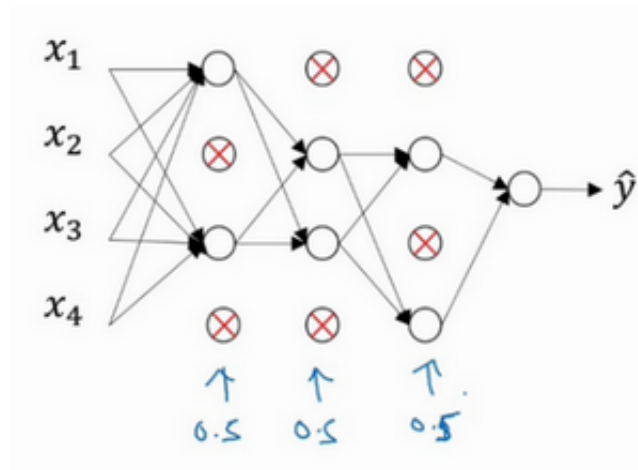
直观理解就是lambda增加到足够大，W会接近于0，实际上是不会发生这种情况的，我们尝试消除或至少减少许多隐藏单元的影响，最终这个网络会变得更简单，这个神经网络越来越接近逻辑回归，我们直觉上认为大量隐藏单元被完全消除了，其实不然，实际上是该神经网络的所有隐藏单元依然存在，但是它们的影响变得更小了。神经网络变得更简单了，貌似这样更不容易发生过拟合，不确定这个直觉经验是否有用，不过在编程中执行正则化时，可以实际看到一些方差减少的结果。

## dropout 正则化 (Dropout Regularization)

**Dropout** (随机失活) 工作原理：



假设你在训练上图这样的神经网络，它存在过拟合，这就是**dropout**所要处理的，我们复制这个神经网络，**dropout**会遍历网络的每一层，并设置消除神经网络中节点的概率。假设网络中的每一层，每个节点都以抛硬币的方式设置概率，每个节点得以保留和消除的概率都是0.5，设置完节点概率，我们会消除一些节点，然后删除掉从该节点进出的连线，最后得到一个节点更少，规模更小的网络，然后用**backprop**方法进行训练。



**Dropout**实现：

最常用的实现方法：**inverted dropout**（反向随机失活）

首先定义一个向量d，用一个三层（l=3）网络来举例说明

```
d3 = np.random.rand(a3.shape[0],a3.shape[1])
```

然后看它是否小于某数，我们称之为**keep-prob**，**keep-prob**是一个具体数字，它表示保留某个隐藏单元的概率，此处**keep-prob**等于0.8，它意味着消除任意一个隐藏单元的概率是0.2，它的作用就是生成随机矩阵。

从第三层中获取激活函数a

$a3 = np.multiply(a3, d3)$ ，它的作用就是让d3中所有等于0的元素（输出），而各个元素等于0的概率只有20%，乘法运算最终把d3中相应元素输出，即让d3中0元素与a3中相对元素归零。

最后，我们向外扩展a3，用它除以0.8，或者除以**keep-prob**参数。

```
a3 = a3/keep-prob
```

如果**keep-prop**设置为1，那么就不存在**dropout**，因为它会保留所有节点。反向随机失活（**inverted dropout**）方法通过除以**keep-prob**，确保a3的期望值不变。

## 其他正则化方法（Other regularization methods）

### 1.数据扩增

假设你正在拟合猫咪图片分类器，如果你想通过扩增训练数据来解决过拟合，但扩增数据代价高，而且有时候我们无法扩增数据，但我们可以通过添加这类图片来增加训练集。例如，水平翻转图片

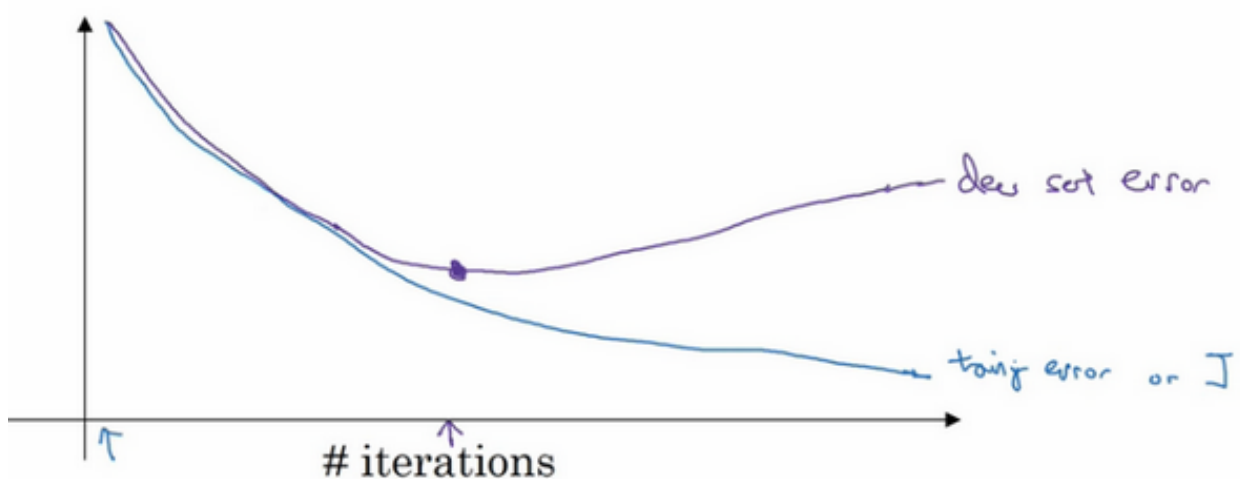


类似于：



## 2.early stopping

绘画训练集和验证集上的分类误差，或验证集上的代价函数



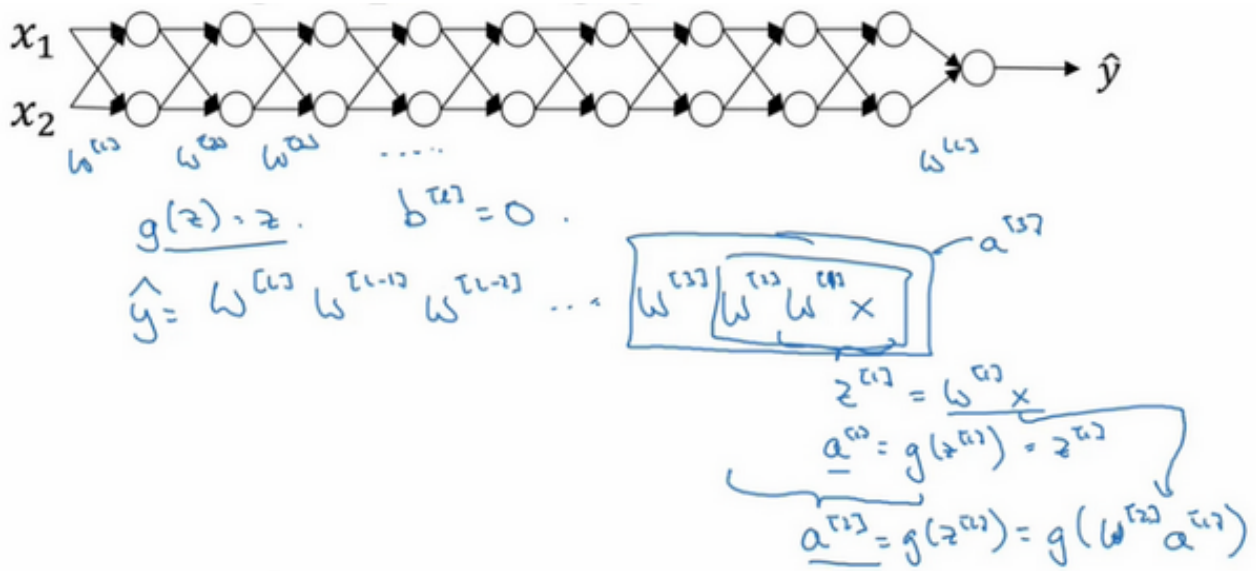
**early stopping**要做就是在中间点停止迭代过程，我们得到一个值中等大小的弗罗贝尼乌斯范数，与正则化相似，选择参数 $w$ 范数较小的神经网络，以保证神经网络过度拟合不严重。

主要缺点：不能独立地处理优化算法和预防过拟合这两个问题【能实现称为正交化】

优点：只运行一次梯度下降，你可以找出的较小值，中间值和较大值，而无需尝试L2正则化超级参数 $\lambda$ 的很多值。

## 梯度消失/梯度爆炸 (Vanishing / Exploding gradients)

假设模型如下，且参数 $b$ 为0



假设每个权重矩阵  $W^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$

对于一个深度神经网络， $y$  的值将爆炸式增长。

若  $W^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$

激活函数的值以指数级递减。

即深度神经网络是产生梯度消失或爆炸问题

一个不完整的解决方案：

设置

$n$  表示神经元的输入特征数量

即设置权重矩阵为：

$$w^{[l]} = np.random.randn(shape) * np.sqrt(\frac{1}{n^{[l-1]}})$$

使用 **Relu** 激活函数：

$$np.sqrt(\frac{2}{n^{[l-1]}});$$

使用 **tanh** 函数：

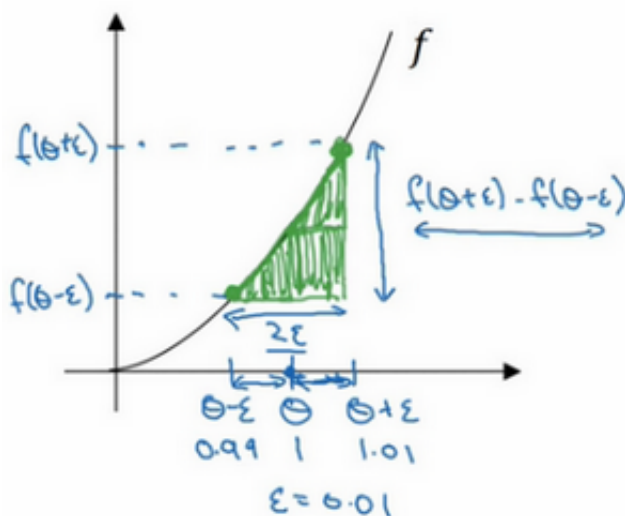
$$\sqrt{\frac{1}{n^{[l-1]}}};$$



## 梯度检验 (Gradient checking)

原理：

使用双边误差的方法更逼近导数



高宽比值为：
$$\frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$$

相对于单边误差的高宽比值更接近于函数导数

使用**grad check**检验**backprop**的实施是否正确：

假设你的网络中含有参数 $w$ 和 $b$ ，为了执行梯度检验，首先要做的就是，把所有参数转换成一个巨大的向量数据，你要做的就是将矩阵 $W$ 转换成一个向量，把所有 $W$ 矩阵转换成向量之后，做连接运算，得到一个巨型向量 $\theta$ ，该向量表示为参数 $\theta$ ，代价函数是所有 $W$ 和 $b$ 的函数，现在你得到了一个的代价函数（即） $J(\theta)$

同理用 $dw$ 和 $db$ 得到 $d\theta$ 。

$J$ 是超参数 $\theta$ 的一个函数,使用双边误差对每个组成元素 $\theta$ 计算 $d\theta_{\text{approx}}$ 的值

$$d\theta_{\text{approx}}[i] = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \epsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i - \epsilon, \dots)}{2\epsilon}$$

$d\theta_{\text{approx}}$ 应该接近于 $d\theta$

Check 
$$\frac{\|d\theta_{\text{approx}} - d\theta\|_2}{\|d\theta_{\text{approx}}\|_2 + \|d\theta\|_2}$$



$\epsilon$ 可能为 $10^{-7}$ ，使用这个取值范围内的，如果你发现计算方程式得到的值为或更小，这就很好，这意味着导数逼近很有可能是正确的，它的值非常小。

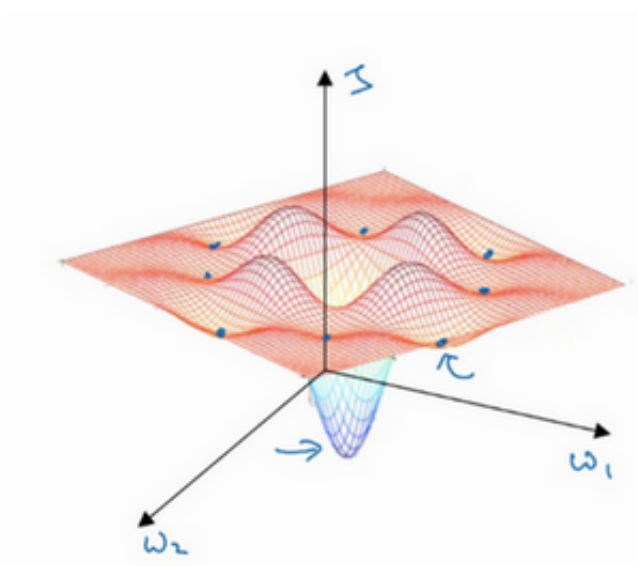
$\epsilon$ 在范围 $10^{-5}$ 内，我就要小心了，也许这个值没问题，但我会再次检查这个向量的所有项，确保没有一项误差过大，可能这里有bug。

$\epsilon$ 在范围 $10^{-3}$ 内，我就会担心是否存在bug，计算结果应该比小很多，如果比大很多，我就会很担心，担心是否存在bug。

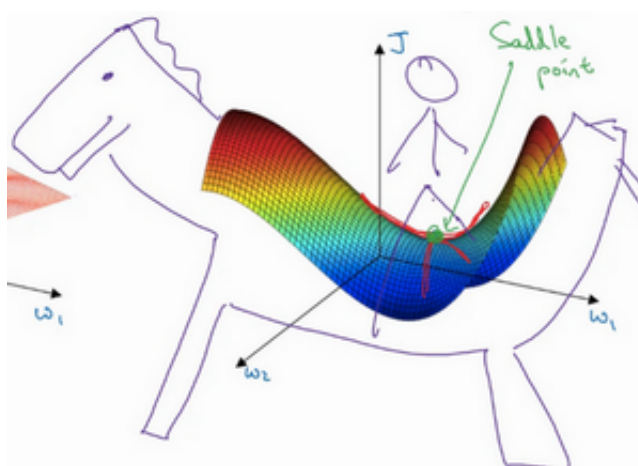
## 第二周：优化算法 (Optimization algorithms)

### 局部最优的问题(The problem of local optima)

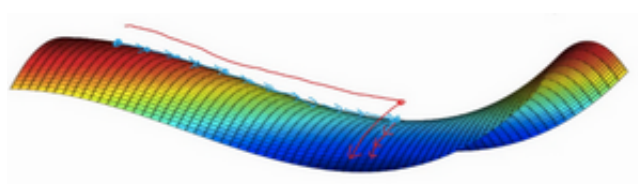
在深度学习研究早期，人们总是担心优化算法会困在极差的局部最优



但实际的高维曲面



更可能碰到鞍点，想象一下，就像是放在马背上的马鞍一样导数为0的点，这个点叫做鞍点



我们可以沿着这段长坡走，直到这里，然后走出平稳段。

此时就需要优化算法加速学习算法。

## Mini-batch 梯度下降 (Mini-batch gradient descent)

在巨大的数据集使用批量梯度下降来训练神经网络速度过慢，每一次需要遍历所有样本

Mini-batch 梯度下降把训练集分割为小一点的子集训练，这些子集被取名为**mini-batch**。假设每一个子集中只有1000个样本，那么把其中1到1000取出来，将其称为第一个子训练集，也叫做**mini-batch**，以此类推。

$$X = \underbrace{[x^{(1)} \ x^{(2)} \ x^{(3)} \ \dots \ x^{(1000)}]}_{X^{\{1\}}} \mid \underbrace{[x^{(1001)} \ \dots \ x^{(2000)}]}_{X^{\{2\}}} \mid \dots \mid \underbrace{[ \dots \ x^{(m)} ]}_{X^{\{5,000\}}}$$

(n,m)

500万个样本的训练集得到5000个**mini-batch**:  $X^{\{5000\}}$

新的符号：把 $x^{(1)}$ 到 $x^{(1000)}$ 称为 $X^{\{1\}}$ ， $x^{(1001)}$ 到 $x^{(2000)}$ 称为 $X^{\{2\}}$

对应处理Y

$$Y = \underbrace{[y^{(1)} \ y^{(2)} \ y^{(3)} \ \dots \ y^{(1000)}]}_{Y^{\{1\}}} \mid \underbrace{[y^{(1001)} \ \dots \ y^{(2000)}]}_{Y^{\{2\}}} \mid \dots \mid \underbrace{[ \dots \ y^{(m)} ]}_{Y^{\{5,000\}}}$$

(1,m)

实现：

遍历**mini-batch**子集，在**for**循环里你要做得基本就是对每个X的子集t和Y的子集t进行梯度下降

### Mini-batch gradient descent

Repeat  $\sum$  for  $t = 1, \dots, 5000$   $\{$

Forward prop on  $X^{\{t\}}$ .

$$\begin{aligned} z^{(t)} &= W^{(t)} X^{\{t\}} + b^{(t)} \\ A^{(t)} &= \sigma^{(t)}(z^{(t)}) \\ &\vdots \\ A^{(t)} &= \sigma^{(t)}(z^{(t)}) \end{aligned}$$

Compute cost  $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{1000} \mathcal{L}(y^{(i)}, \hat{y}^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l=1}^L \|W^{(l)}\|_F^2$

Backprop to compute gradients w.r.t  $J^{\{t\}}$  (w.r.t  $(X^{\{t\}}, Y^{\{t\}})$ )

$$W^{(t+1)} = W^{(t)} - \alpha \Delta W^{(t)}, \quad b^{(t+1)} = b^{(t)} - \alpha \Delta b^{(t)}$$

$\}$  "1 epoch"   
 pass through training set.

1 step of gradient descent w.r.t  $X^{\{t+1\}}, Y^{\{t+1\}}$ . (as if 1000)

$X, Y$

Andrew Ng

## 动量梯度下降法 (Gradient descent with Momentum)

使用指数加权平均数 (Exponentially weighted averages) 进行计算

$$v_t = \beta v_{t-1} + (1 - \beta) \theta_t$$

指数加权平均的偏差修正 (Bias correction in exponentially weighted averages) : 帮助你在早期获取更好的估测

$\rightarrow v_t = \beta v_{t-1} + (1 - \beta) \theta_t$

$v_0 = 0$

$v_1 = \cancel{0.98 v_0} + 0.02 \theta_1$

$v_2 = 0.98 v_1 + 0.02 \theta_2$

$= 0.98 \times 0.02 \times \theta_1 + 0.02 \theta_2$

$= 0.0196 \theta_1 + 0.02 \theta_2$

$\frac{v_t}{1 - \beta^t}$

$t=2: 1 - \beta^t = 1 - (0.98)^2 = 0.0396$

$\frac{v_2}{0.0396} = \frac{0.0196 \theta_1 + 0.02 \theta_2}{0.0396}$

Andrew Ng

但实际应用中一般不使用修正。

指数加权平均数应用到梯度下降得到 **Momentum**

↓

Momentum:

On iteration  $t$ :

Compute  $\Delta w, \Delta b$  on current mini-batch.

$$v_{\Delta w} = \beta v_{\Delta w} + (1 - \beta) \Delta w$$
$$v_{\Delta b} = \beta v_{\Delta b} + (1 - \beta) \Delta b$$

$$v_{\theta} = \beta v_{\theta} + (1 - \beta) \theta_t$$

Friction  $\rightarrow$

$\uparrow$  velocity

$\uparrow$  acceleration

$w := w - \alpha v_{\Delta w}, \quad b := b - \alpha v_{\Delta b}$

↩

# Implementation details

On iteration  $t$ :

Compute  $dW, db$  on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1 - \beta) dW$$

$$v_{db} = \beta v_{db} + (1 - \beta) db$$

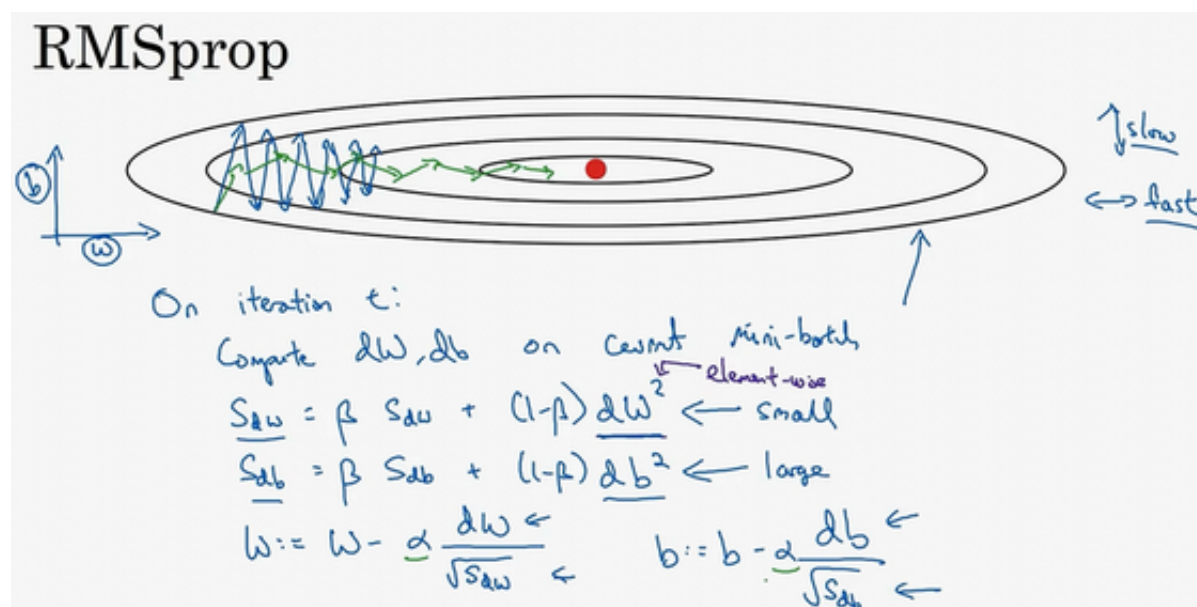
$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

Hyperparameters:  $\alpha, \beta$        $\beta = 0.9$

个算法肯定优于没有Momentum的梯度下降算法

## RMSprop

全称是root mean square prop算法



## Adam 优化算法(Adam optimization algorithm)

Adam优化算法基本上就是将Momentum和RMSprop结合在一起。

## Adam optimization algorithm

$$V_{dw}=0, S_{dw}=0, V_{db}=0, S_{db}=0$$

On iteration  $t$ :

Compute  $dw, db$  using current mini-batch

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw, \quad V_{db} = \beta_1 V_{db} + (1-\beta_1) db \quad \leftarrow \text{"momentum"} \beta_1$$

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2, \quad S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2 \quad \leftarrow \text{"RMSprop"} \beta_2$$

$$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t), \quad V_{db}^{corrected} = V_{db} / (1-\beta_1^t)$$

$$S_{dw}^{corrected} = S_{dw} / (1-\beta_2^t), \quad S_{db}^{corrected} = S_{db} / (1-\beta_2^t)$$

$$W := W - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected} + \epsilon}}, \quad b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \epsilon}}$$

参数选择

## Hyperparameters choice:

$\rightarrow \alpha$ : needs to be tune

$\rightarrow \beta_1$ : 0.9  $\rightarrow (dw)$

$\rightarrow \beta_2$ : 0.999  $\rightarrow (dw^2)$

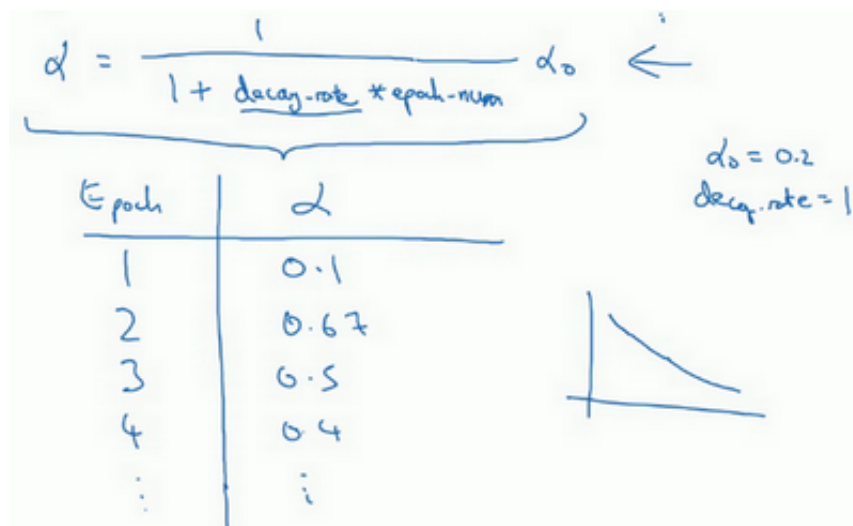
$\rightarrow \epsilon$ :  $10^{-8}$

Adam: Adaptive moment estimation

## 学习率衰减(Learning rate decay)

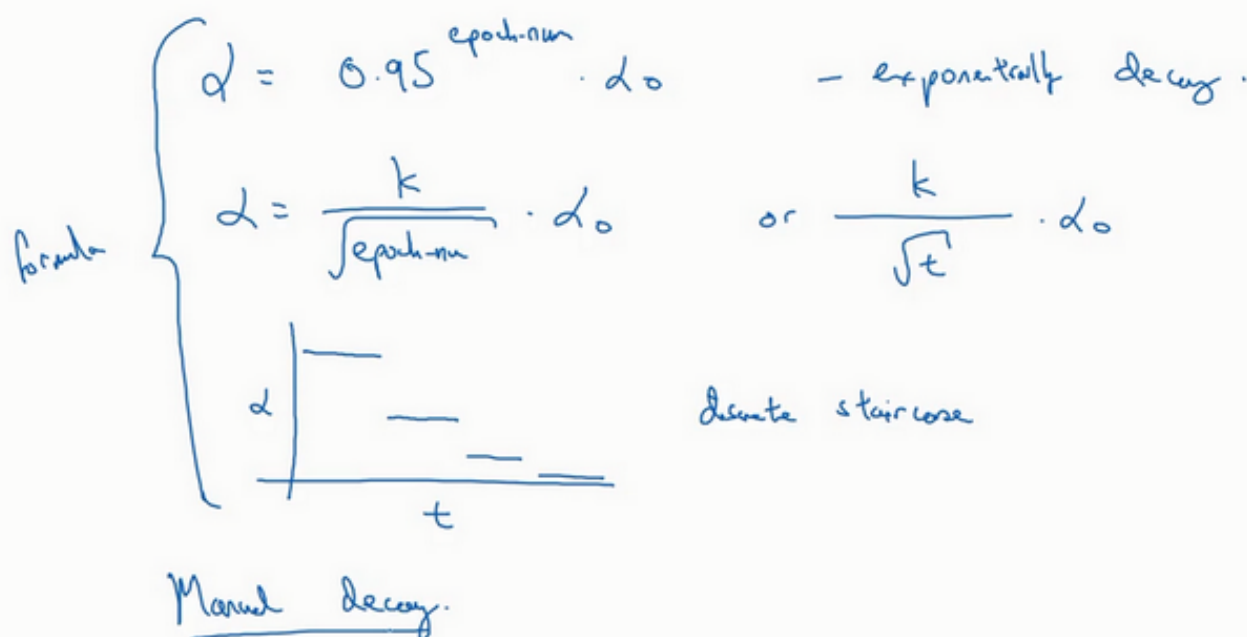
加快学习算法的一个办法就是随时间慢慢减少学习率，我们将之称为学习率衰减。





(**decay-rate**称为衰减率, **epoch-num**为迭代数,  $\alpha_0$ 为初始学习率)

## Other learning rate decay methods



## 第三周 超参数调试、Batch正则化和程序框架 (Hyperparameter tuning)

归一化网络的激活函数 (Normalizing activations in a network)

$\gamma$ 和 $\beta$ 的作用是可以随意设置新的 $z$ 的平均值，通过赋予 $\gamma$ 和 $\beta$ 其它值，可以使你构造含其它平均值和方差的隐藏单元值。

## 将 Batch Norm 拟合进神经网络 (Fitting Batch Norm into a neural network)

### Softmax 回归 (Softmax regression)

logistic回归实现二分类，Softmax实现多分类

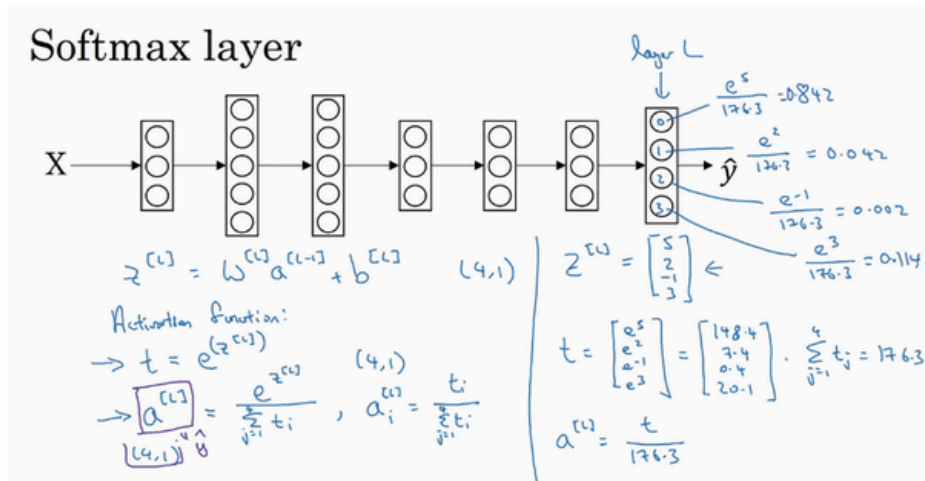
计算：【主要采用不同激活函数获得激活值】



我们来看一个例子，详细解释，假设你算出了 $z^{[l]}$ ， $z^{[l]}$ 是一个四维向量，假设为 $z^{[l]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$ ，我们要做的就是用

这个元素取幂方法来计算 $t$ ，所以 $t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$ ，如果你按一下计算器就会得到以下值 $t = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$ ，我们从向量 $t$

得到向量 $a^{[l]}$ 只需要将这些项目归一化，使总和为1。如果你把 $t$ 的元素都加起来，把这四个数字加起来，得到176.3，最终 $a^{[l]} = \frac{t}{176.3}$ 。



Softmax输出高维向量，向量内各值代表各分类的概率