

# Contents

<b>1</b>	<b>1. setting up your machine learning application</b>	<b>1</b>
1.1	1.1. train/dev/test sets . . . . .	1
1.2	1.2. bias/variance . . . . .	3
1.3	1.3. basic recipe for machine learning . . . . .	3
<b>2</b>	<b>2. regularizing your neural network</b>	<b>7</b>
2.1	2.1. regularization . . . . .	7
2.2	2.2. why regularization reduces overfitting? . . . . .	7
2.3	2.3. dropout regularization . . . . .	7
2.4	2.4. understanding dropout . . . . .	11
2.5	2.5. other regularization methods . . . . .	11
2.5.1	2.5.1. data augmentation . . . . .	11
2.5.2	2.5.2. early stopping . . . . .	11
<b>3</b>	<b>3. setting up your optimization problem</b>	<b>13</b>
3.1	3.1. normalizing inputs . . . . .	13
3.2	3.2. vanishing/exploding gradients . . . . .	14
3.3	3.3. weight initialization for deep networks . . . . .	14
3.4	3.4. numerical approximation of gradients . . . . .	16
3.5	3.5. gradient checking . . . . .	17
3.6	3.6. gradient checking implementation notes . . . . .	17

contents

- 1. setting up your machine learning application
  - 1.1. train/dev/test sets
  - 1.2. bias/variance
  - 1.3. basic recipe for machine learning
- 2. regularizing your neural network
  - 2.1. regularization
  - 2.2. why regularization reduces overfitting?
  - 2.3. dropout regularization
  - 2.4. understanding dropout
  - 2.5. other regularization methods
    - 2.5.1. data augmentation
    - 2.5.2. early stopping
- 3. setting up your optimization problem
  - 3.1. normalizing inputs
  - 3.2. vanishing/exploding gradients
  - 3.3. weight initialization for deep networks
  - 3.4. numerical approximation of gradients
  - 3.5. gradient checking
  - 3.6. gradient checking implementation notes

## 1 1. setting up your machine learning application

### 1.1 1.1. train/dev/test sets

传统机器学习，例如数据总量有 1w，可以划分 train:dev:test=70:0:30，或者，train:dev:test=60:20:20。但对于大数据，例如 100w 的数据，那适当的比例应该是 98:1:1 或者甚至是 99.5:0.25:0.25，或者 99.5:0.4:0.1。

另外，训练集和验证集的分布要保持一致。测试集是为了 no-bias 地验证模型效果的，有些时候，可以甚至不要测试集，只要验证集就可以了。

## Train/dev/test sets

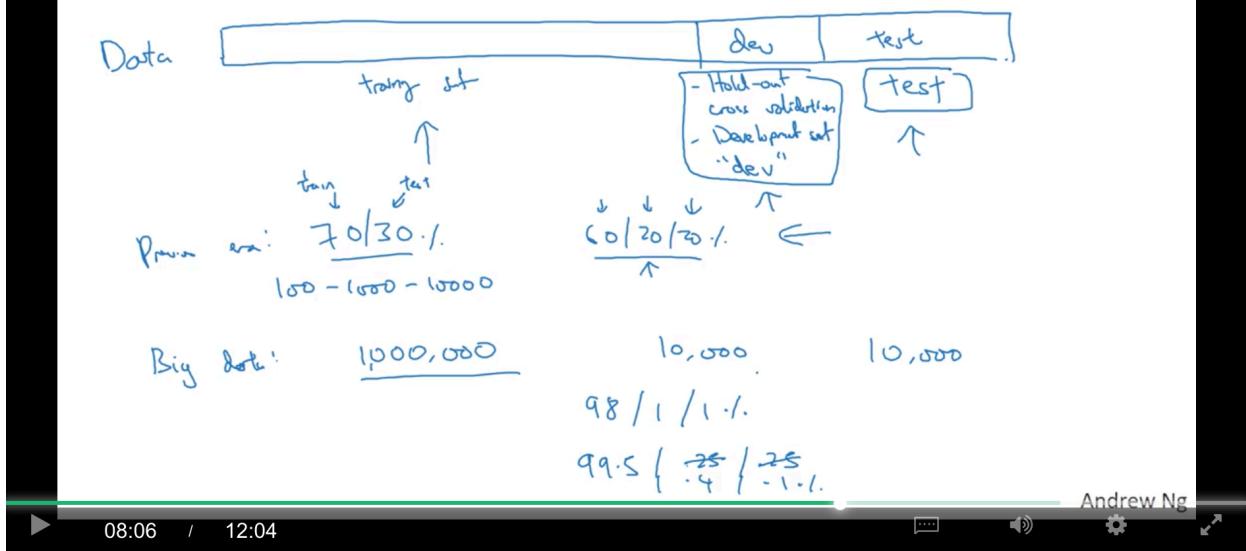
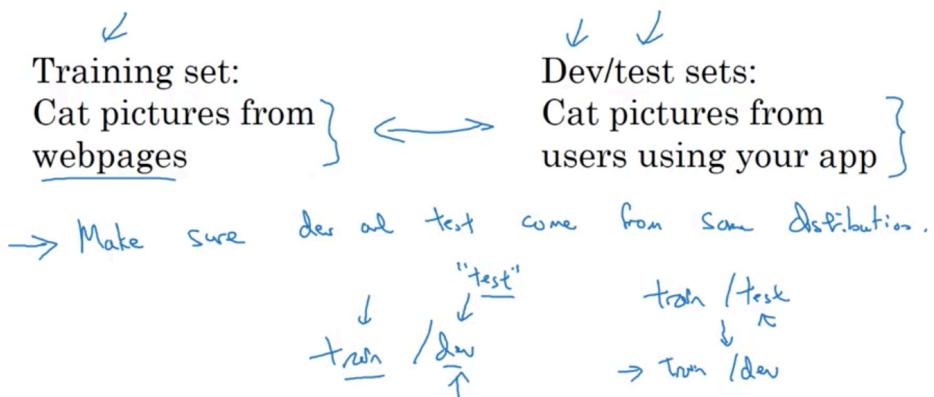


Figure 1: train-dev-test-ratio.png

## Mismatched train/test distribution

Certs



Not having a test set might be okay. (Only dev set.)

Andrew Ng

Figure 2: distribution-between-train-and-dev.png

## 1.2 1.2. bias/variance

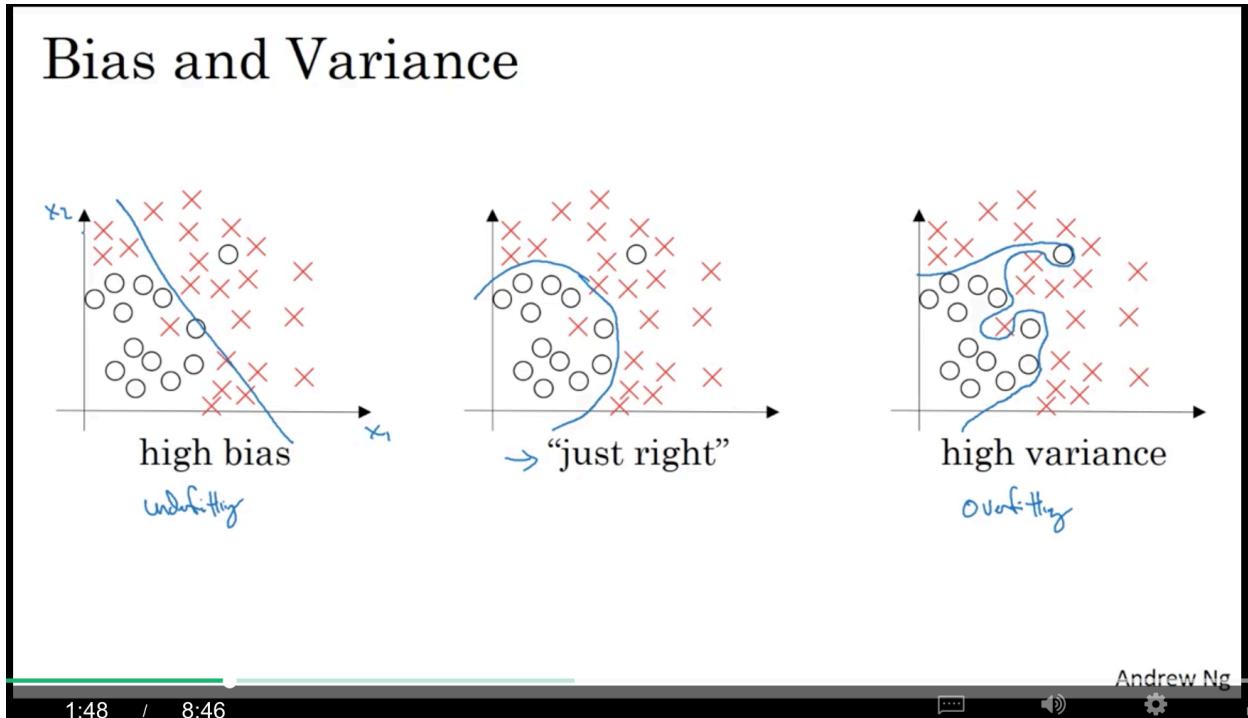


Figure 3: bias-variance-under-overfitting.png

首先  $\text{Error} = \text{Bias} + \text{Variance}$

Error 反映的是整个模型的准确度，Bias 反映的是模型在样本上的输出与真实值之间的误差，即模型本身的精准度，Variance 反映的是模型每一次输出结果与模型输出期望之间的误差，即模型的稳定性。<https://www.zhihu.com/question/27068705/answer/35151681>

low-high-variance-bias 的四象限如下：

对应模型的最优复杂度如下：

当 optimal error(bayes error) 约等于 0% 时（错误率是 0，全部才能识别出来），如下图所示。当 optimal error 是 15% 时，第二个分类器就是 low bias 了。

最差情况就是两个都 high 的，如下紫色曲线（线性部分是 high bias，因为有很多没分对的情况；中间两个点是 high variance，overfitting 了）：

## 1.3 1.3. basic recipe for machine learning

首先，如果你的模型有 high bias(在训练集上表现很差)，那么，试着用更大的网络（更复杂的模型），或者，训练更久。

如果 bias 已经变小了，那么看看是否是 high variance(从训练集到验证集的表现的变化)，如果是，那么可以尝试获取更多数据/正则化。

在传统机器学习领域，有 bias-variance tradeoff 的说法，因为往往优化 bias，会让 variance 变差，反之亦然。

但在 deep learning 中，上述方法在提高一个指标的时候，往往并不会太影响另一个指标。所以，如果你已经使用了正则化，那么，使用更大的模型几乎不会有什么负面影响，造成的影响只是计算量的增加而已 (Training a bigger network almost never hurts. And the main cost of training a neural network that's too big is just computational time, so long as you're regularizing.)

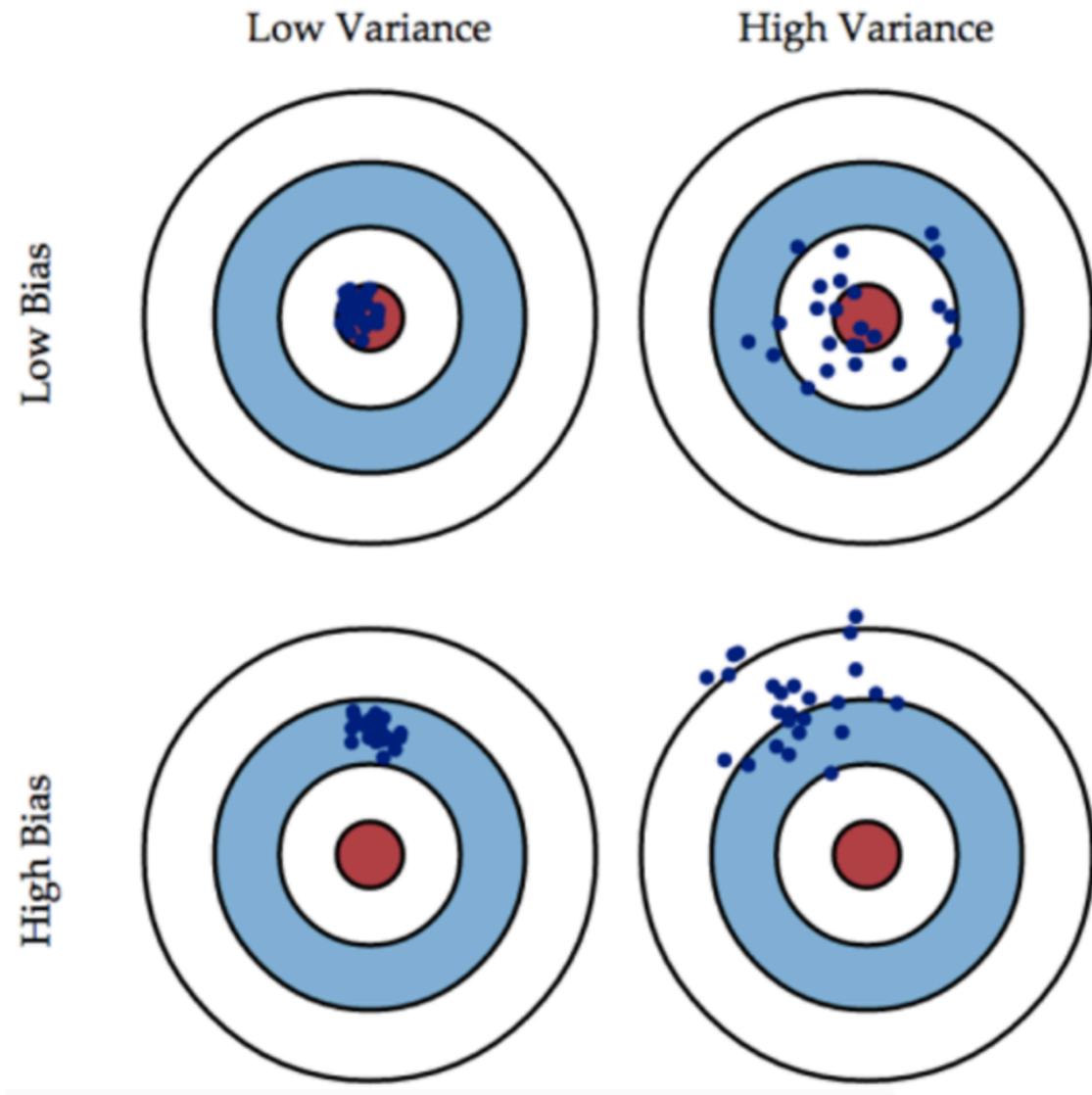


Figure 4: bias-variance-high-low.png

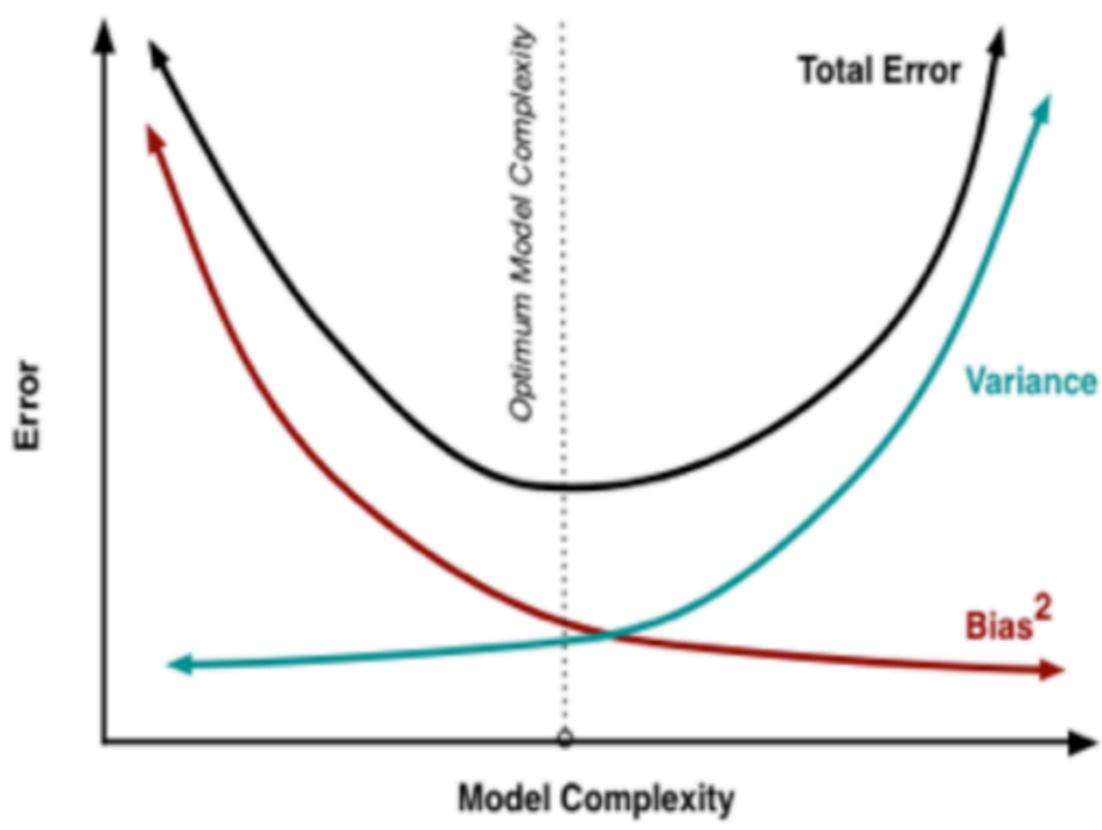
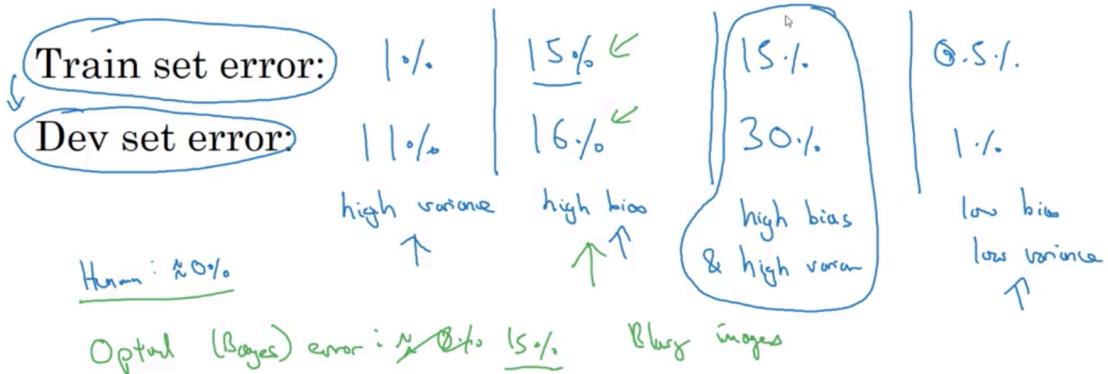


Figure 5: bias-variance-high-low-complexity.png

# Bias and Variance

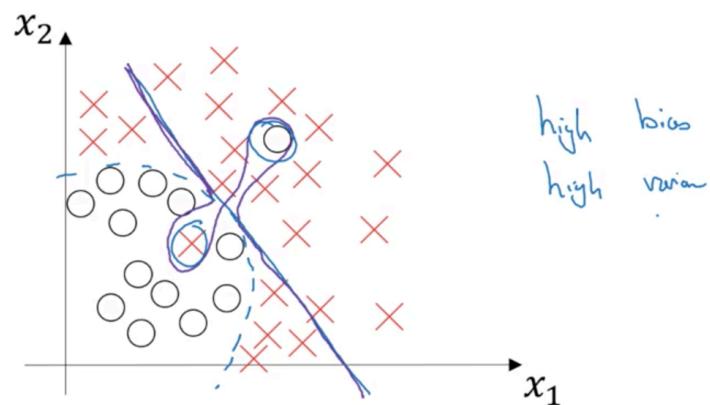
Cat classification



6:47 / 8:46

Andrew Ng

## High bias and high variance



8:04 / 8:46

Andrew Ng

Figure 7: bias-variance-worst.png

# Basic recipe for machine learning

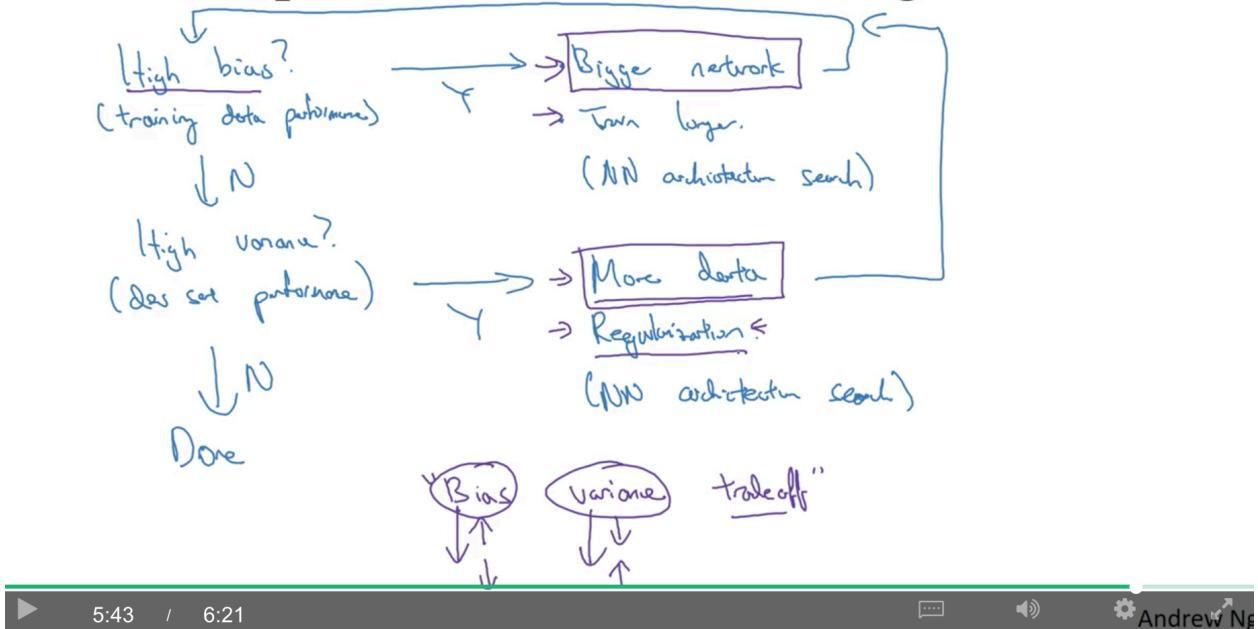


Figure 8: basic-recipe-for-machine-learning.png

## 2 2. regularizing your neural network

### 2.1 2.1. regularization

在  $l_1$  中, 一般只对  $w$  加正则, 因为  $b$  是一个实数, 而  $w$  的维度较高  $n_x$ , 所以  $b$  的影响可以忽略不计。

在神经网络的训练中,  $l_1$  用处不大,  $l_2$  用得很广泛。其中的  $\lambda$  是正则化参数, 一般通过验证集或者 cross-validation 来设置。

神经网络的  $l_2$  里, 范数是 F-范数的平方 ( $F$ -范数 =  $2$ -范数 = 矩阵中所有元素的平方和再开平方), 也叫做 weight decay(因为在对  $w$  进行梯度下降时, 相当于给  $w$  乘上了一个小于 1 的因子:  $(1 - \alpha \frac{\lambda}{m})$ )。

### 2.2 2.2. why regularization reduces overfitting?

直观地看, 当  $\lambda$  很大时, 最小化 loss, 会使得  $w$  趋向 0, 这样, 相当于很多神经元无形地被干掉了, 模型变简单了。

接下来,  $w$  接近 0, 所以  $z$  也比较小, 而如果激活函数是  $\tanh(z)$ , 那么,  $\tanh(z)$  的这个区域里是接近线性的, 所以模型就更像一个比较大的线性回归。

另外, 因为加了正则化项, 所以原来的  $J$  可能不会在每个 elevations(调幅数量?? 看着又像 iteration..) 都单调递减, 要看新的  $J$

### 2.3 2.3. dropout regularization

简单理解, dropout 就是对每一层设置一个 dropout rate, 在训练时, 针对每一条训练样本, 以这个比例把某些神经元及其连接的权重直接去掉。

dropout 的实现方式:  $\text{keep\_prob}=0.8$ , 表示  $\text{dropout\_rate}=0.2$  第三行的  $a3/\text{keep\_prob}$ , 就是 inverted dropout 的精髓, 这是为了让  $a3$  的期望和没有做 dropout 保持一致。加了这句话, 在 test 的时候就更加容易了, 在早期的实现中, 可能没这句, test 就会比较复杂。

# Logistic regression

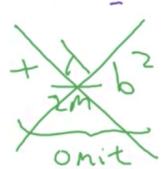
$$\min_{w,b} J(w,b) \quad w \in \mathbb{R}^n, b \in \mathbb{R}$$

$J(w,b) = \underbrace{\frac{1}{m} \sum_{i=1}^m \ell(y^{(i)}, y^{(i)})}_{L_2 \text{ regularization}} + \frac{\lambda}{2m} \|w\|_2^2$

$\|w\|_2^2 = \sum_{j=1}^n w_j^2 = w^T w \leftarrow$

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

$\lambda = \text{regularization parameter}$   
 $\lambda_{\text{lambda}}$   
 $\lambda_{\text{lambd.}}$



$w$  will be sparse

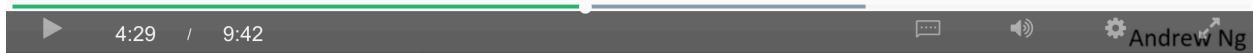


Figure 9: regression-lr.png

# Neural network

$$J(w^{(0)}, b^{(0)}, \dots, w^{(L)}, b^{(L)}) = \underbrace{\frac{1}{m} \sum_{i=1}^m \ell(y^{(i)}, y^{(i)})}_{\text{"Frobenius norm"}} + \frac{\lambda}{2m} \sum_{l=1}^L \|w^{(l)}\|_F^2$$

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

$w: (n^{(0)} \ n^{(1)} \ \dots)$

$\frac{\partial J}{\partial w^{(l)}} = \partial w^{(l)}$

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

$\rightarrow w^{(l)} := w^{(l)} - \alpha \partial w^{(l)}$

$w^{(l)} := w^{(l)} - \alpha \left[ (\text{from backprop}) + \frac{\lambda}{m} w^{(l)} \right]$

$(1 - \frac{\alpha \lambda}{m}) w^{(l)}$

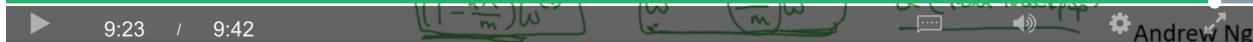
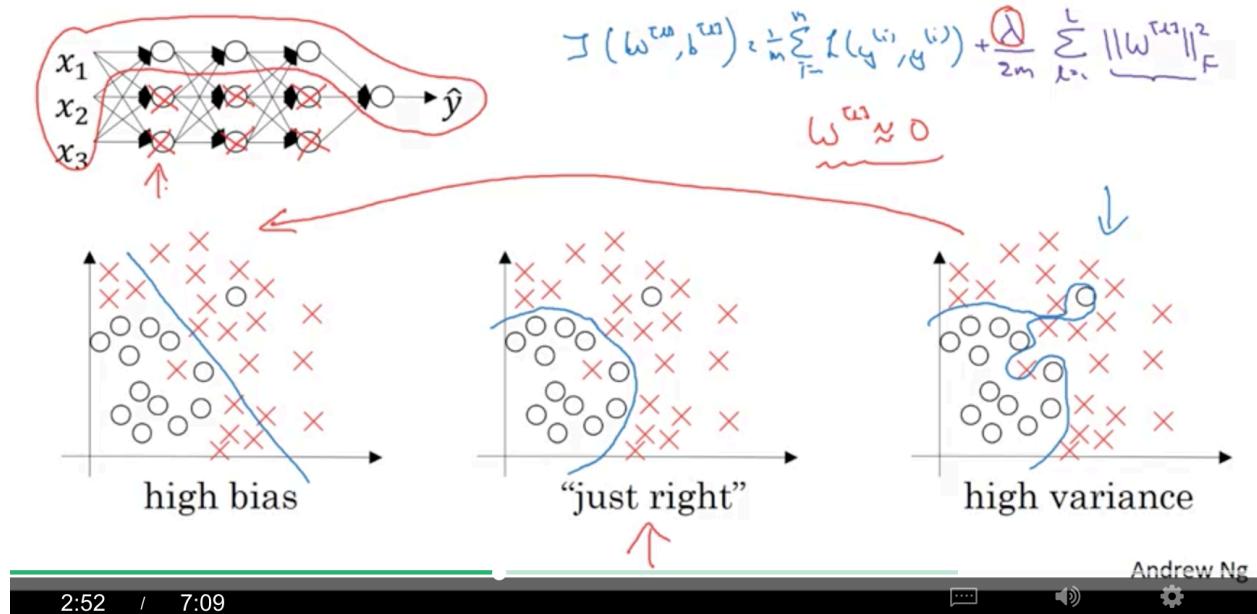


Figure 10: neural-network-lr.png

## How does regularization prevent overfitting?

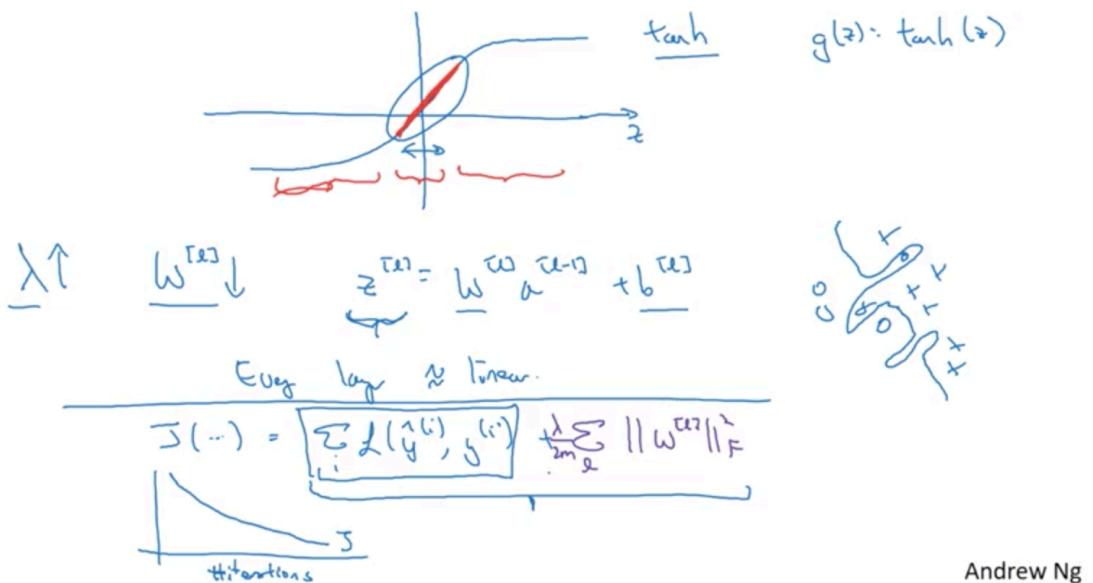


2:52 / 7:09

Andrew Ng

Figure 11: how-does-regularization-reduces-overfitting.png

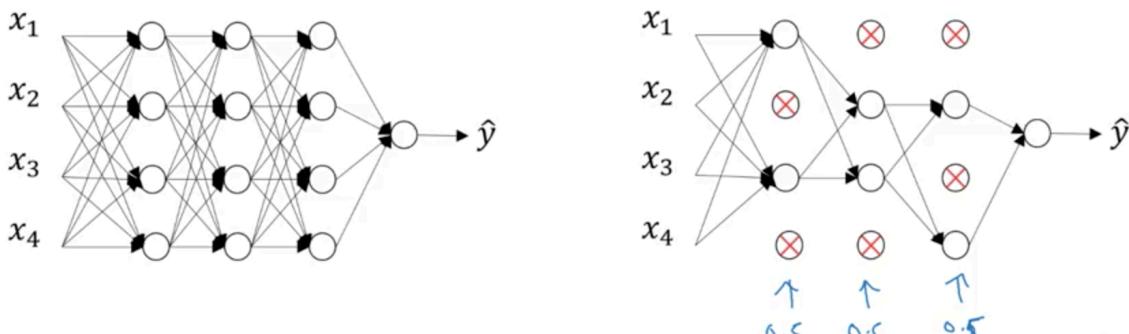
## How does regularization prevent overfitting?



Andrew Ng

Figure 12: how-does-regularization-reduces-overfitting-2.png

# Dropout regularization



1:39 / 9:25

Andrew Ng

Figure 13: dropout-regularization-introduction.png

## Implementing dropout (“Inverted dropout”)

Illustrate with layer  $l=3$ .  $\text{keep-prob} = \frac{0.8}{x}$   $\underline{0.2}$

$\rightarrow \delta^3 = \text{np.random.rand}(a^3.shape[0], a^3.shape[1]) < \text{keep-prob}$

$\underline{a^3} = \text{np.multiply}(a^3, \delta^3)$  #  $a^3 * \delta^3$ .

$\rightarrow \underline{a^3} /= \text{keep-prob}$

↑ 50 units.  $\rightsquigarrow$  10 units shut off

$$z^{(4)} = w^{(4)} \cdot \underline{\frac{a^{(3)}}{0.2}} + b^{(4)}$$

↓ reduced by  $\underline{20\%}$ .

Test

$$1 = \underline{0.8}$$

7:21 / 9:25

Andrew Ng

Figure 14: dropout-regularization-implementation.png

test 阶段，不要使用 **dropout**，因为我们并不想让预测的结果是 random 的，或者是有噪音的。如果用了 inverted dropout，在 test 的时候，就不需要做这个/=keep\_prob 的操作了。

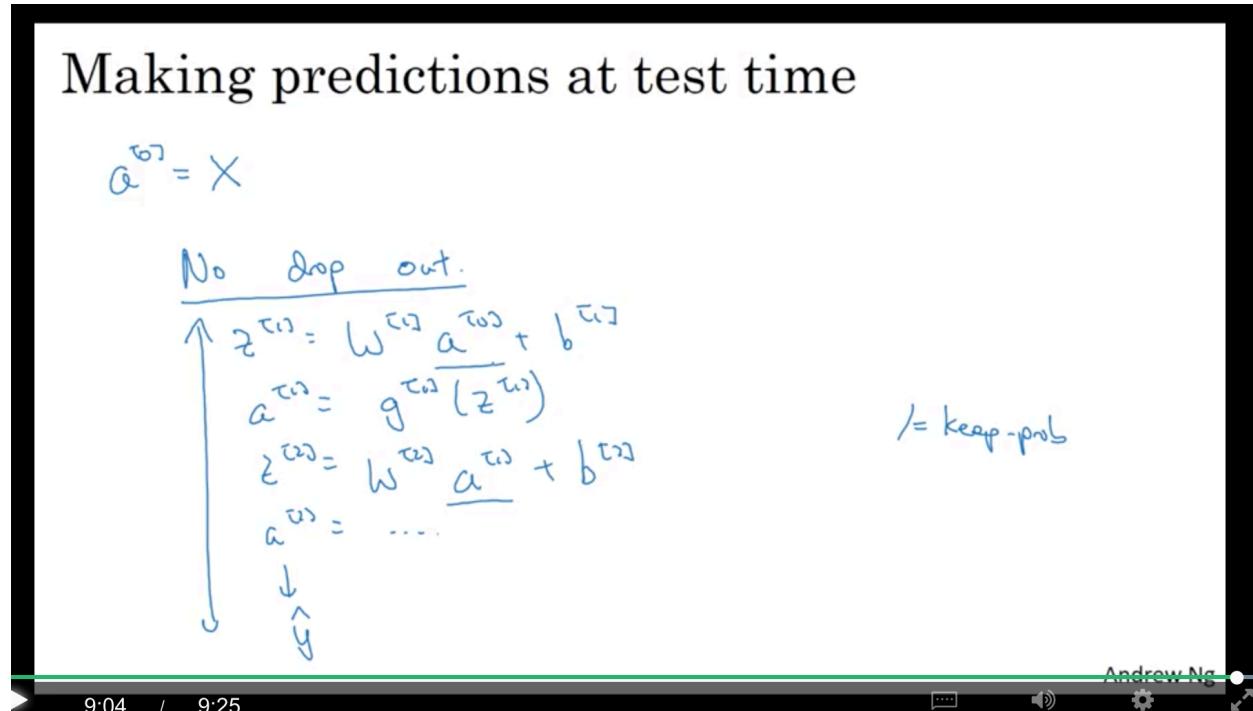


Figure 15: dropout-regularization-test-implementation.png

## 2.4 2.4. understanding dropout

dropout 相当于，每个神经元不只仅依赖某一个输入，所以会倾向于将与各输入神经元的连接权重的值分散开来。==> 可以 shrink squared norm of the weight，类似于 L2 的效果。

一般为不同的层设置不同的 keep\_prob，input 层一般接近 1.0，如果某两层间的矩阵比较大，可以设小一点的 keep\_prob，例如图中的 0.7, 0.5, 0.7。

一般只有在可能 overfitting 的时候才用 dropout，例如 cv 中，往往输入的像素很多，但数据量没那么大，所以常用 dropout 来避免过拟合。

downside of dropout: cost function J is no longer well defined...所以每一轮迭代后 J 的曲线并不一定是单调递减的，一般的做法是，先关闭 dropout (keep\_prob=1)，然后调整模型到 J 曲线是递减的，再开启 dropout，看看 dropout 有没有引入 bug..

## 2.5 2.5. other regularization methods

### 2.5.1 2.5.1. data augmentation

例如将图片水平旋转，垂直旋转，剪切，缩放，旋转一定角度，增加噪音，可以快速地扩充训练样本。

### 2.5.2 2.5.2. early stopping

在图中一方面画出 training set 的 error 或者 cost function J(一般会一直下降)，另一方面，画出 dev set 的 error (一般会下降到一个低谷，然后上升)。

early stopping 就是在 dev set 到达低谷时，停止训练。

# Why does drop-out work?

Intuition: Can't rely on any one feature, so have to spread out weights.  $\rightarrow$  Shrink weights.

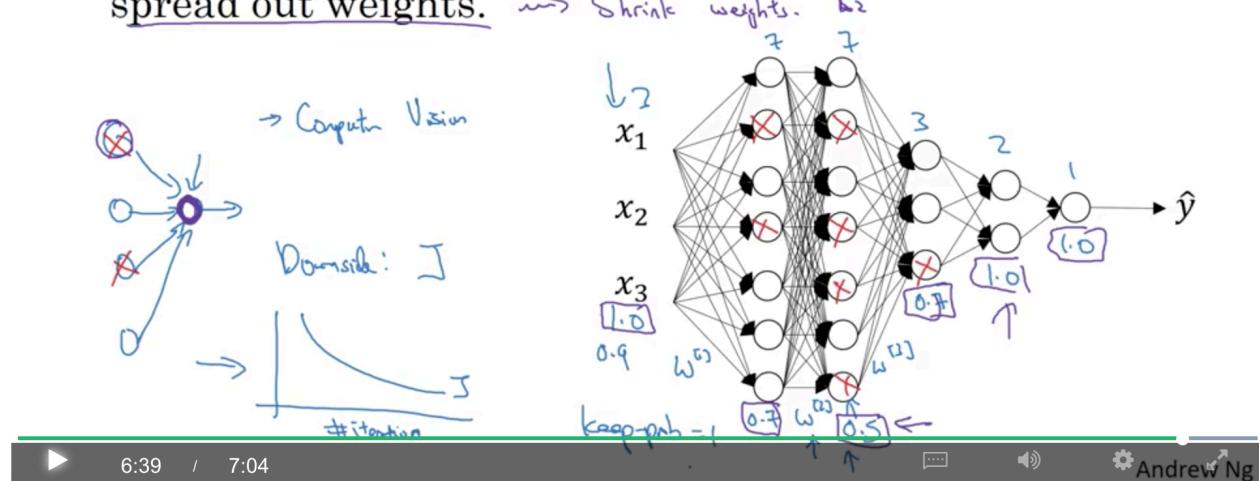


Figure 16: dropout-regularization-understanding-dropout.png

## Data augmentation

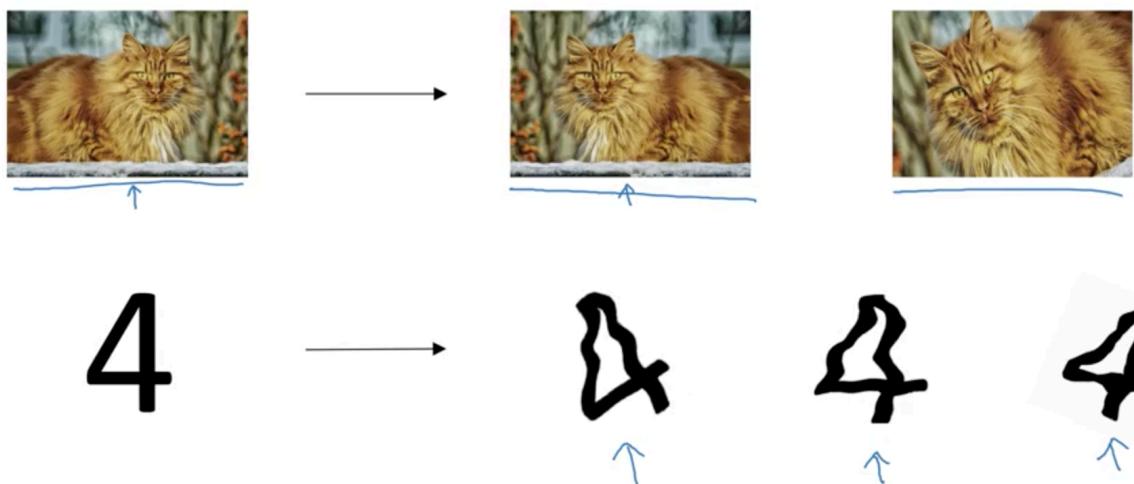


Figure 17: other-regularization-data-augmentation.png

因为在开始迭代时，一般参数  $w$  会接近 0 (一般随机初始化时会初始化成比较小的值)，而训练轮数太多时，可能  $w$  就会变得很大，所以 early stopping 时，可能  $\|w\|_F^2$  正好是中间大小。

orthogonalization:

- optimize cost function  $J$ (例如 sgd, momentum 等)
- not overfitting (例如 regularization 等)

上面二者其实是相互正交的，也就是可以相互独立地优化，但 dropout 的 downside 就在于它将二者结合在一起了，因此无法独立地优化两个 task。

如果不使用 dropout，一般使用 L2 正则，这样就能训练很久，并且会使得超参的搜索空间更加容易分解，也更容易搜索。但 L2 正则的 downside 是需要 search 非常多的  $\lambda$ 。而 early stopping 只需要跑一次梯度下降，就能够试遍 small/large/mid 的  $w$ ，不用试那么多的  $\lambda$ 。

## Early stopping

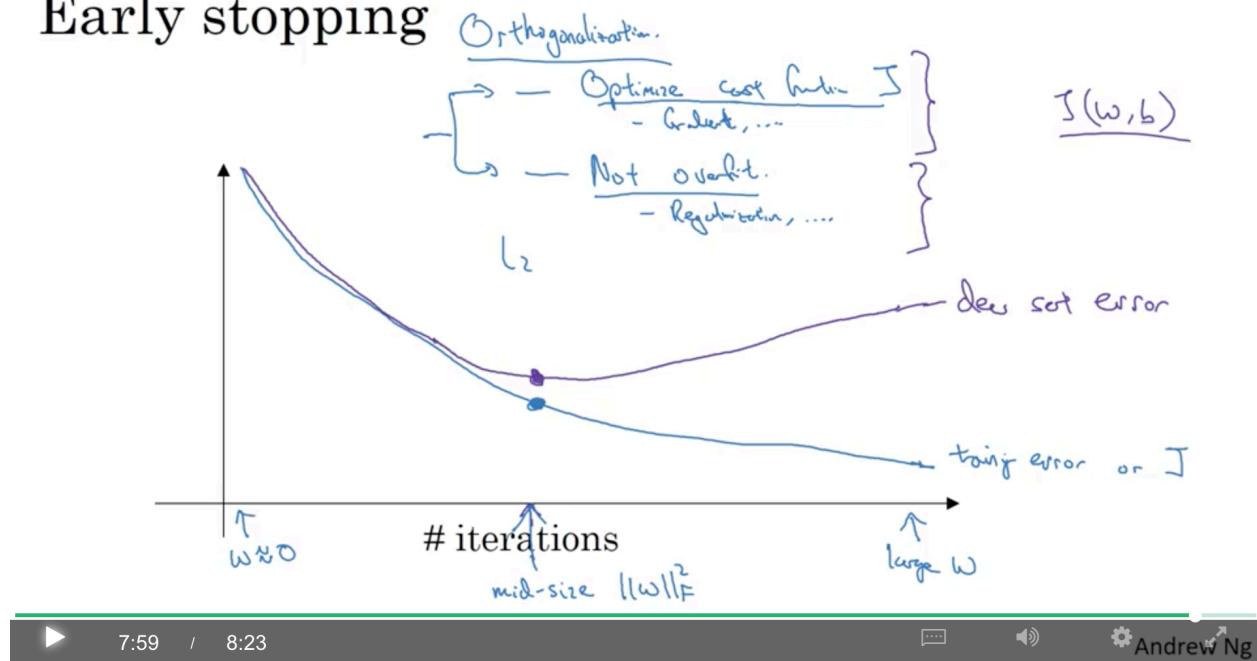


Figure 18: other-regularization-early-stopping.png

### 3 3. setting up your optimization problem

#### 3.1 3.1. normalizing inputs

加速训练的一种方法就是 normalize inputs，分为两步：

1. subtract out(zero out) mean:

$$\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)}$$

$$x = x - \mu$$

变成 0 均值的分布

2. normalize the variance:

如第二张图,  $x_1$  和  $x_2$  的方差差很远, 所以需要

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^m x^{(i)} * x^{(i)}$$

$$x / = \sigma^2$$

其中的  $**$  表示, element-wise squaring。

注意, 对训练集做了上述 normalize 之后, 变为 0 均值, 1 方差, 对 test set 也要用相同的  $\mu$  和  $\sigma$ 。

## Normalizing training sets

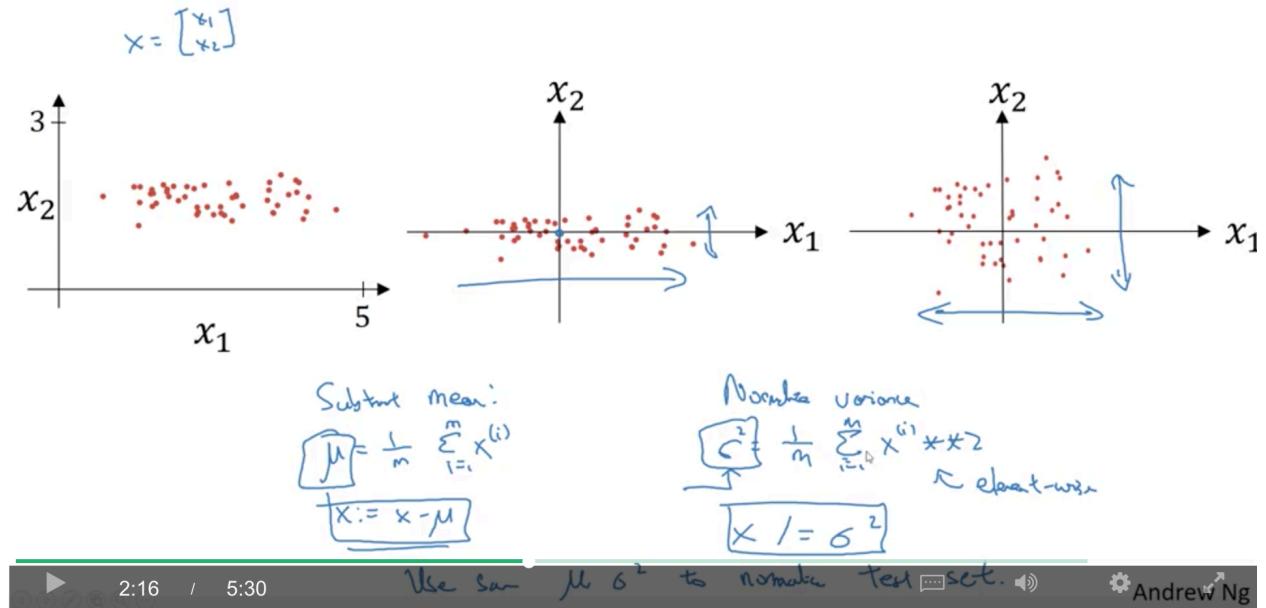


Figure 19: normalizing-training-sets.png

假设  $w$  是一维的, 如果没有 normalize, 那么如左图, 可能  $x_1$  和  $x_2$  的范围差很远, 而  $w_1$  和  $w_2$  (即  $w$  和  $b$ ) 可能也差很远, 就需要用非常小的 learning rate; 而如果 normalize 了, 等高线 (上图沿着 J 轴俯视得到下图) 就相对对称了, 可以采用比较大的 learning rate, 很快地到达 J 的极小点。当然, 现实中  $w$  是多维的, 但也类似。

当然, 如果  $x_1, x_2, x_3$  范围不会差很远, 也不一定要用 normalize, 但用了仍可能会加速。

### 3.2 3.2. vanishing/exploding gradients

假设没有  $b$ , 假设是线性激活函数,  $g(z)=z$ , 如果  $w>1$ , 即使只比 1 大一点, 如果层数很深, 可能最后的激活值就会特别大, 同理, 如果  $w<1$ , 可能最后激活值特别小。

### 3.3 3.3. weight initialization for deep networks

partial solution: 初始化的技巧

input feature 的维度  $n$  越大, 希望  $w_i$  越小, 这样  $z$  才不会太大。

令  $w_i$  的方差  $var(w_i) = \frac{1}{n}$ , 代码就是

## Why normalize inputs?

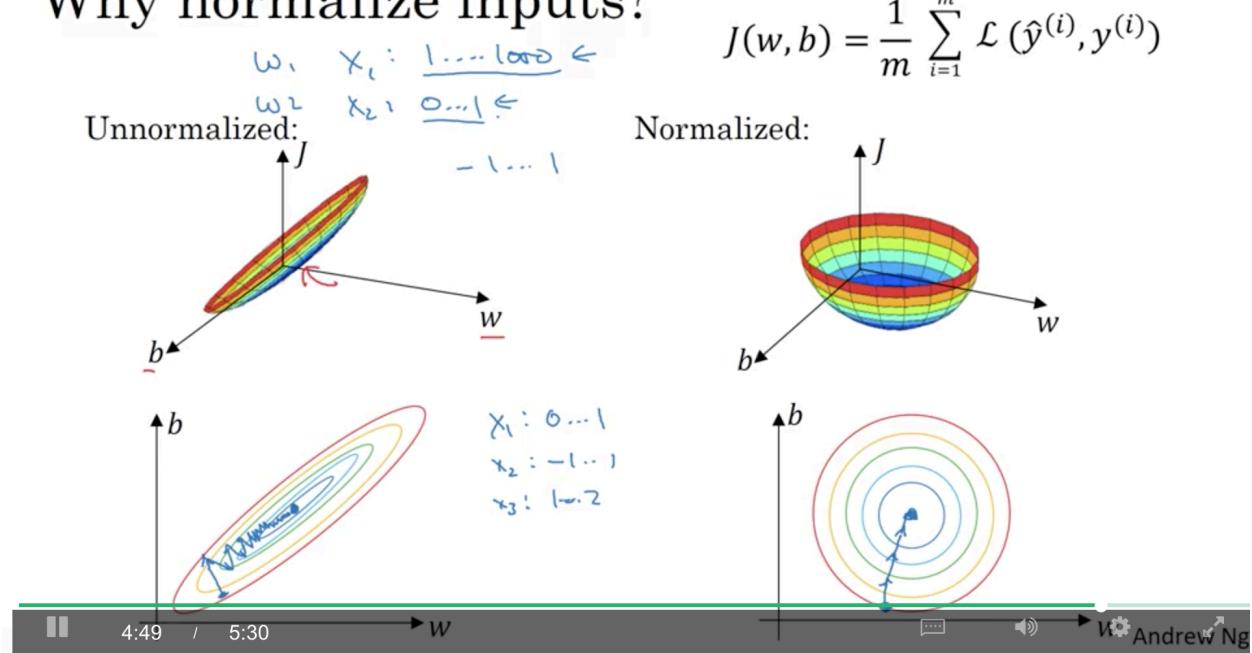


Figure 20: why-normalize-inputs.png

## Vanishing/exploding gradients

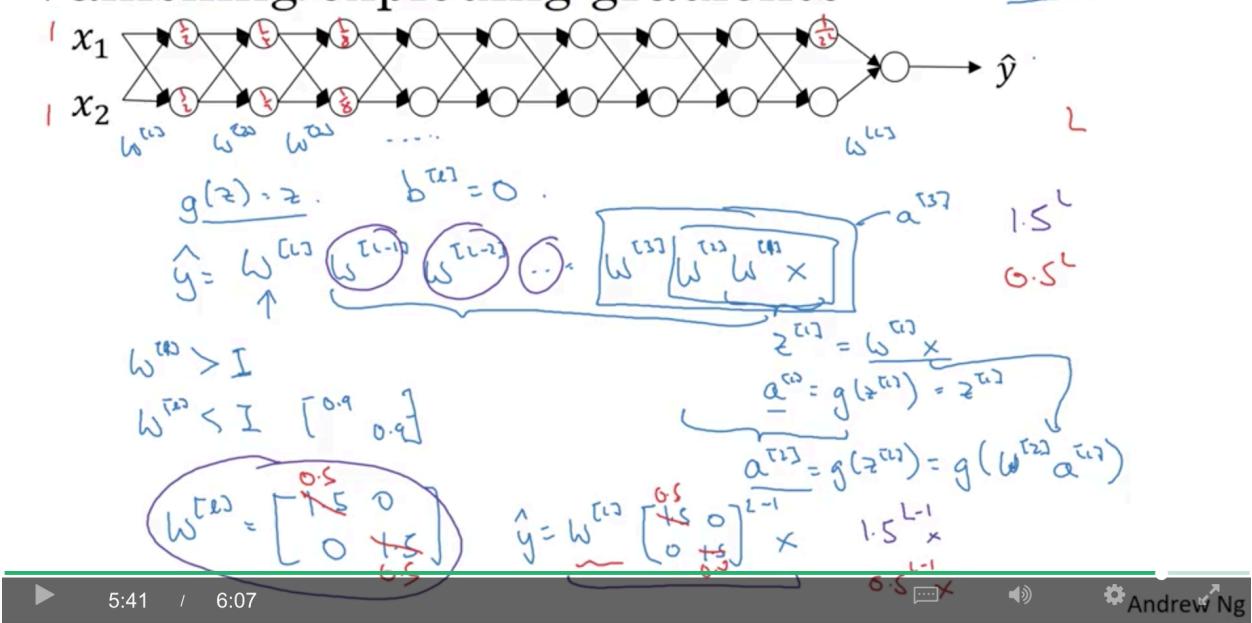


Figure 21: vanishing-exploding-gradients.png

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

如果用的是 ReLU，那么，把上面的 1 改成 2 **【He initialization, He et al., 2015】**:

$$var(w_i) = \frac{2}{n}$$

$$w^{[l]} = np.random.rand(shape(l)) * np.sqrt(\frac{2}{n^{[l-1]}})$$

如果是 tanh，那么，用 **Xavier Initialization**:  $\sqrt{\frac{1}{n^{[l-1]}}}$

如果是 Yoshua Bengio 也提出过如下方法:  $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

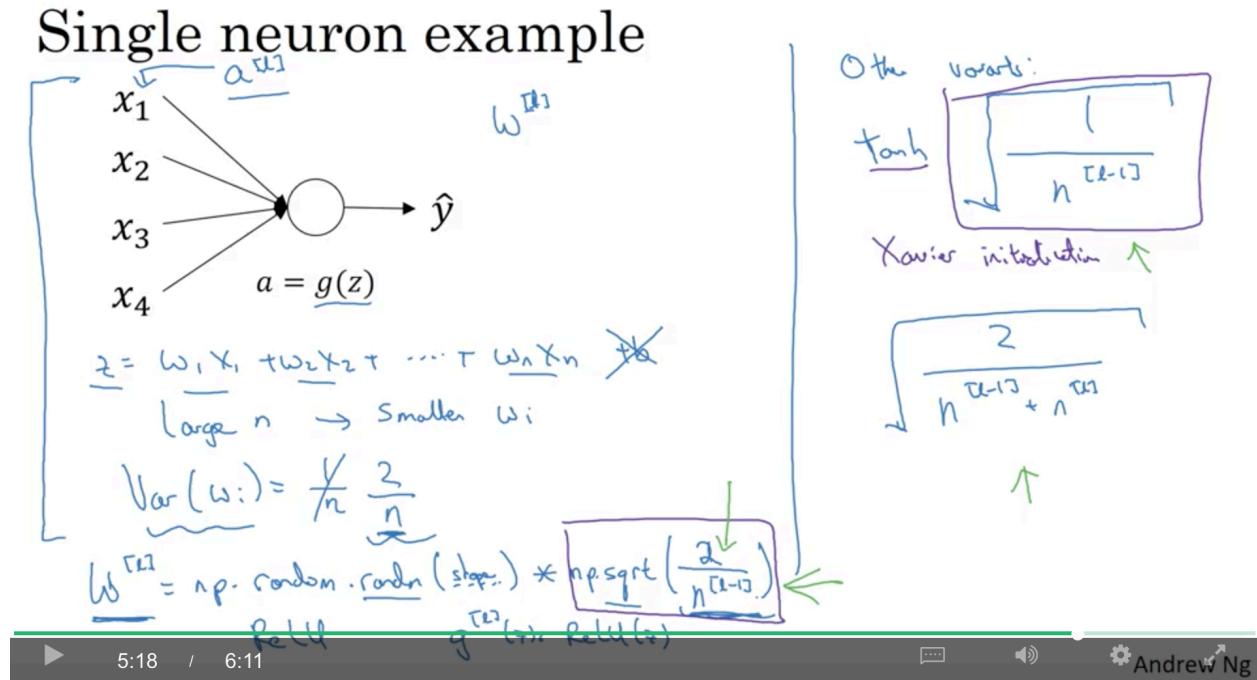


Figure 22: weight-initialization.png

### 3.4 3.4. numerical approximation of gradients

有时写完代码后，不确定梯度算得有没有问题，可以通过逼近的方法来检查梯度。

左边的是 2-sided difference，逼近误差 (approx error) 是  $O(\epsilon^2)$ ，右边是 1-sided difference，逼近误差是  $O(\epsilon)$ 。

如果用 1.01 和 0.99 去算 2-sided difference，得到的结果是 3.0001 和梯度的真实值 3 的 approx error 是 0.0001。

但如果用 1.01 和 1 去算 1-sided difference，得到的结果是  $(1.01 * 3 - 1 * 3) / 0.01 = 3.0301$ ，所以 approx error 是 0.0301

所以，使用 2-sided difference 会更加逼近真实梯度。

# Checking your derivative computation

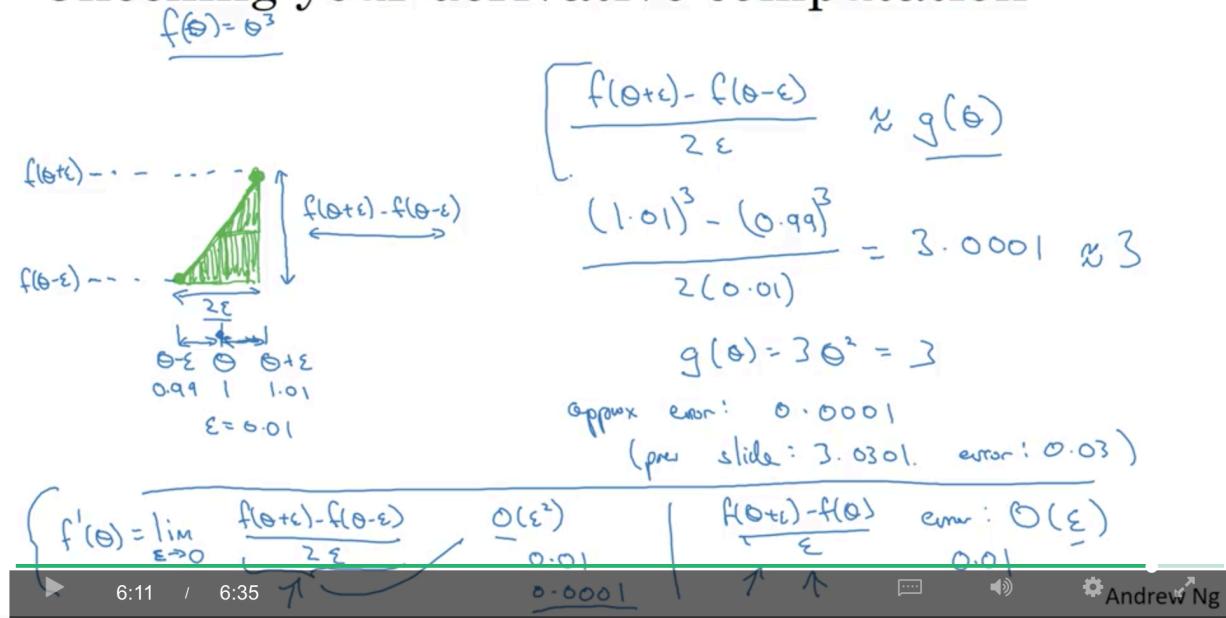


Figure 23: numerical-approximation-of-gradients.png

## 3.5 gradient checking

$W^{[l]}$  和  $dW^{[l]}$  的 dim 是一样的, 如下图, 将所有  $W$  和  $b$  连接在一起, reshape 成一个大 vector  $\theta$ 。同样地, 将所有  $dW$  和  $db$  连接在一起, reshape 成一个大 vector  $d\theta$

gradient check 的步骤如下 ( $\epsilon$  常取  $10^{-7}$ ) , 当下式约等于  $10^{-7}$  时, 应该没问题, 如果大于  $10^{-3}$ , 很可能有问题。

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

## 3.6 gradient checking implementation notes

- 只在 debug 时用 grad check, 训练时不用 (因为计算  $d\theta_{approx}[i]$  很慢)
- 如果 grad check 失败, 看是哪个的 diff 比较大 (例如, 是  $w$  还是  $b$ , 是  $w1, w2, wi$  的哪个), 然后去对应地找 bug
- 记得 regularization term (当损失函数有 regularization 时, 算  $J(\theta)$  时记得带上 regularization term)
- dropout 不适用 grad check。所以如果要用 dropout, 可以在 grad check 时把 `keep_prob` 设成 1.0, 确认 ok 后, 再打开 dropout
- 可能梯度下降在  $W$  和  $b$  接近 0 的时候表现是比较好的, 所以, 可以在 random initialization 后, 进行一次 grad check; 然后在训练了若干轮后, 当  $W$  和  $b$  都比较接近 0 时, 再进行一次 grad check

# Gradient check for a neural network

Take  $\underline{W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}}$  and reshape into a big vector  $\underline{\theta}$ .

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = J(\theta)$$

Take  $\underline{dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}}$  and reshape into a big vector  $\underline{d\theta}$ .

Is  $d\theta$  the gradient of  $J$

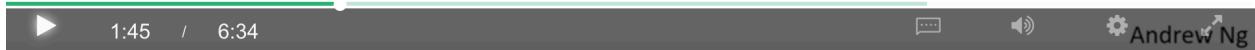


Figure 24: gradient-check-for-a-neural-network.png

## Gradient checking (Grad check)

$$\text{for each } i: \quad \rightarrow \underline{d\theta_{\text{approx}}[i]} = \frac{J(\theta_1, \theta_2, \dots, \overset{\downarrow}{\theta_i + \varepsilon}, \dots) - J(\theta_1, \theta_2, \dots, \overset{\downarrow}{\theta_i - \varepsilon}, \dots)}{2\varepsilon} \quad | \quad \underline{d\theta_{\text{approx}}} \underset{?}{\approx} \underline{d\theta}$$

Check

$$\rightarrow \frac{\|d\theta_{\text{approx}} - d\theta\|_2}{\|d\theta_{\text{approx}}\|_2 + \|d\theta\|_2} \quad \varepsilon = 10^{-7}$$

$\times$   $\begin{cases} 10^{-7} - \text{great!} \\ 10^{-5} \end{cases} \leftarrow$

$$\rightarrow 10^{-3} - \text{worry.} \leftarrow$$



Figure 25: gradient-check.png

## Gradient checking implementation notes

- Don't use in training – only to debug

$$\frac{\partial \Theta_{\text{approx}}[i]}{\uparrow} \longleftrightarrow \frac{\partial \Theta[i]}{\uparrow}$$

- If algorithm fails grad check, look at components to try to identify bug

$$\frac{\partial b^{(l)}}{\uparrow} \quad \frac{\partial w^{(l)}}{\uparrow}$$

$$J(\theta) = \frac{1}{m} \sum_i \ell(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_l \|w^{(l)}\|_F^2$$

$\frac{\partial \theta}{\uparrow} = \text{gradient of } J \text{ wrt. } \theta$

- Remember regularization.

- Doesn't work with dropout.

J

keep-prob = 1.0

- Run at random initialization; perhaps again after some training.



Figure 26: gradient-check-implementation-notes.png