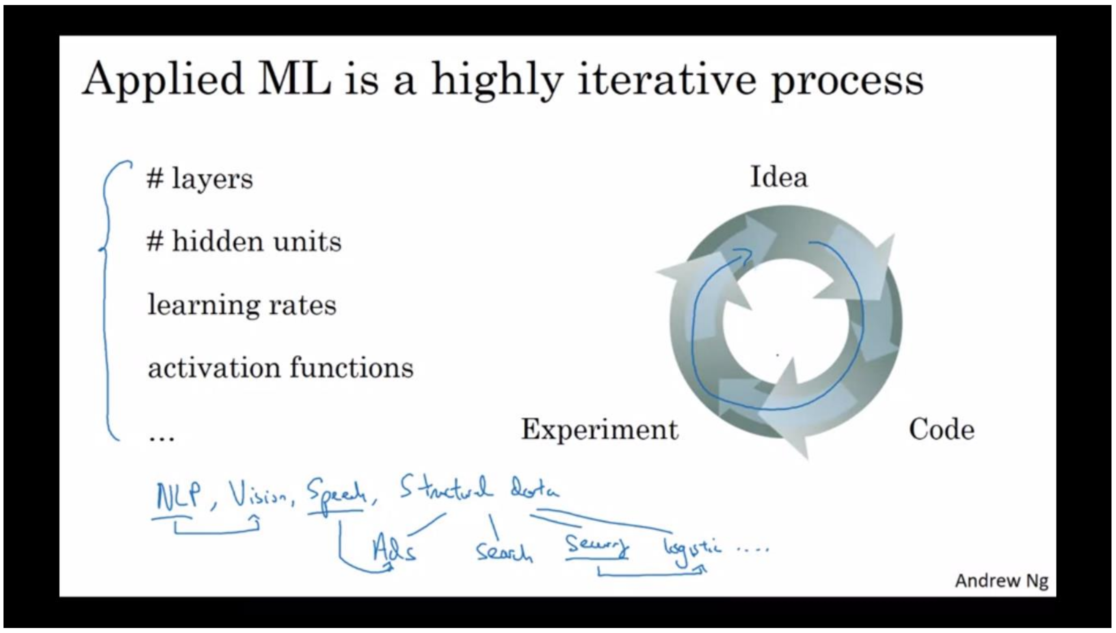
<u>Ahmad Hussameldin Hamed Hassan</u>

Shared Git-hub link: https://github.com/ahmadhassan1993/sharing-github

Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

<u>Week1 Summary:</u>

Goal is to get the suitable hybrid parameters for the specific application, by circulating this loop:



Applied ML is a highly iterative process

layers

hidden units

learning rates

activation functions

...

NLP, Vision, Speech, Structual data

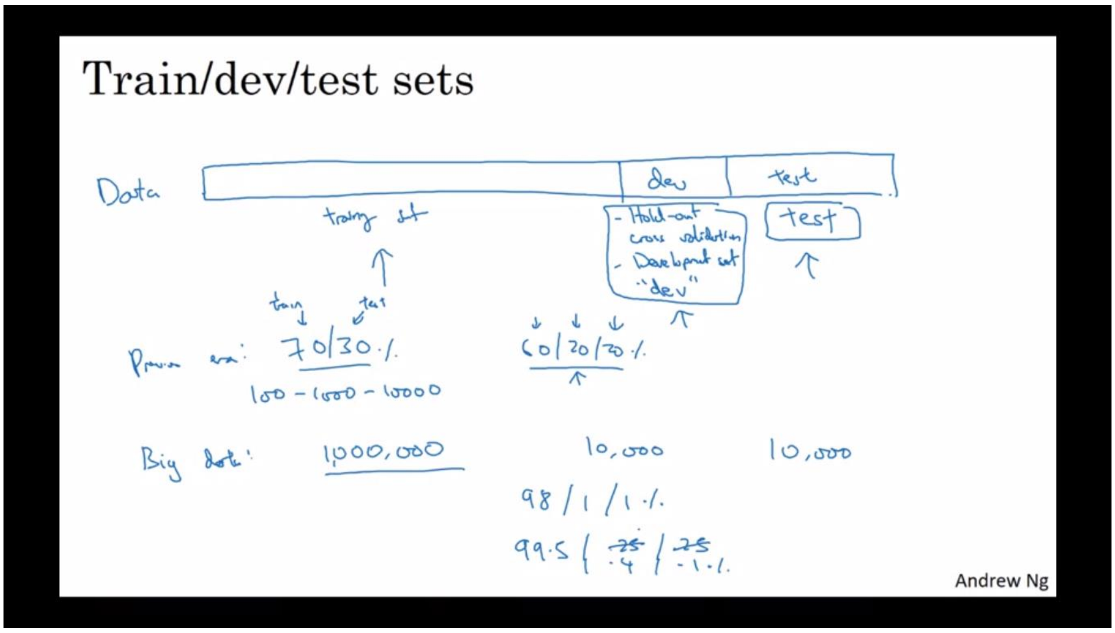
Ads, Search, Security, logistic ....

Andrew Ng

Idea: raw data from application

Code: to implement NN

Experiment: to train and test the chosen hybrid parameter. If not suitable, choose other value and do the loop again.



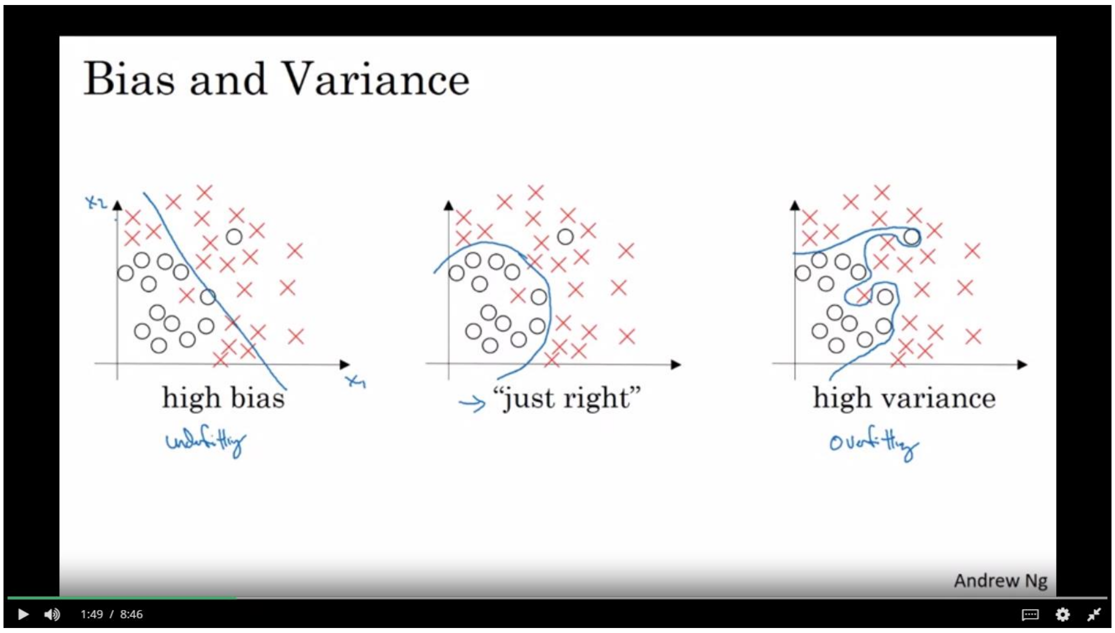
We will divide the data to three sections:

1- Training set
2- Development set
3- Test set

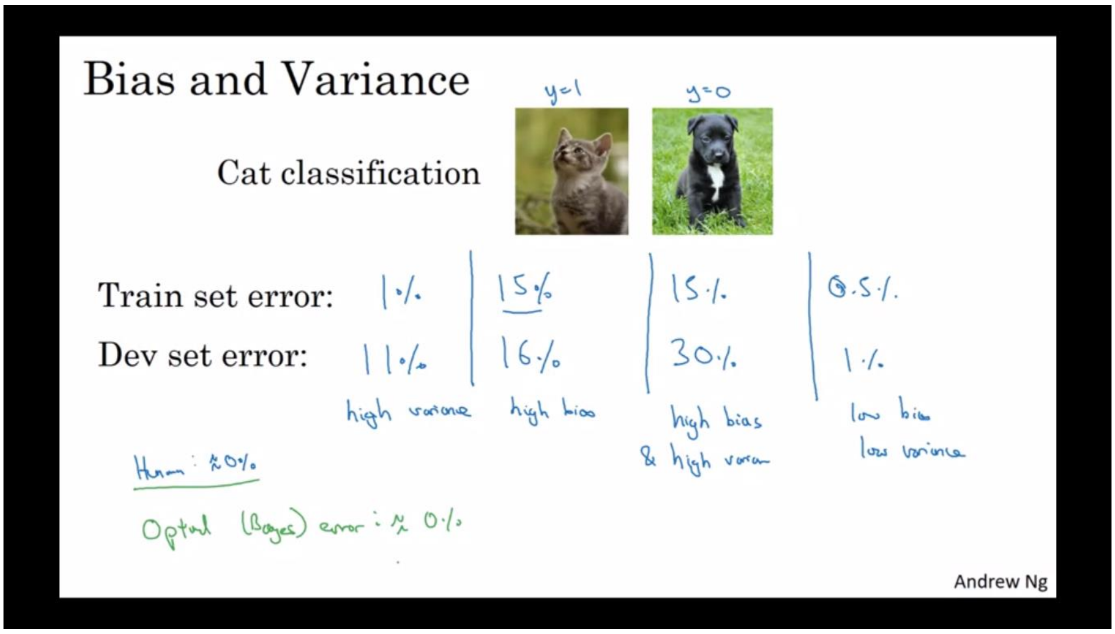
The second part is used to judge whether we shall change the algorithm or not (after seeing training result). This part, as well as the test part, may be large for small data set and vice versa.

Make sure to have both test or dev set and train set from same distribution, to avoid mismatch. For example, recognizing cat from website photo and personal camera photo (the resolution is different).

Not having test set is okay. We use only dev instead. Some people name dev as test, we have to be sure that it is used as dev set.



Bias and Variance



Bias and Variance

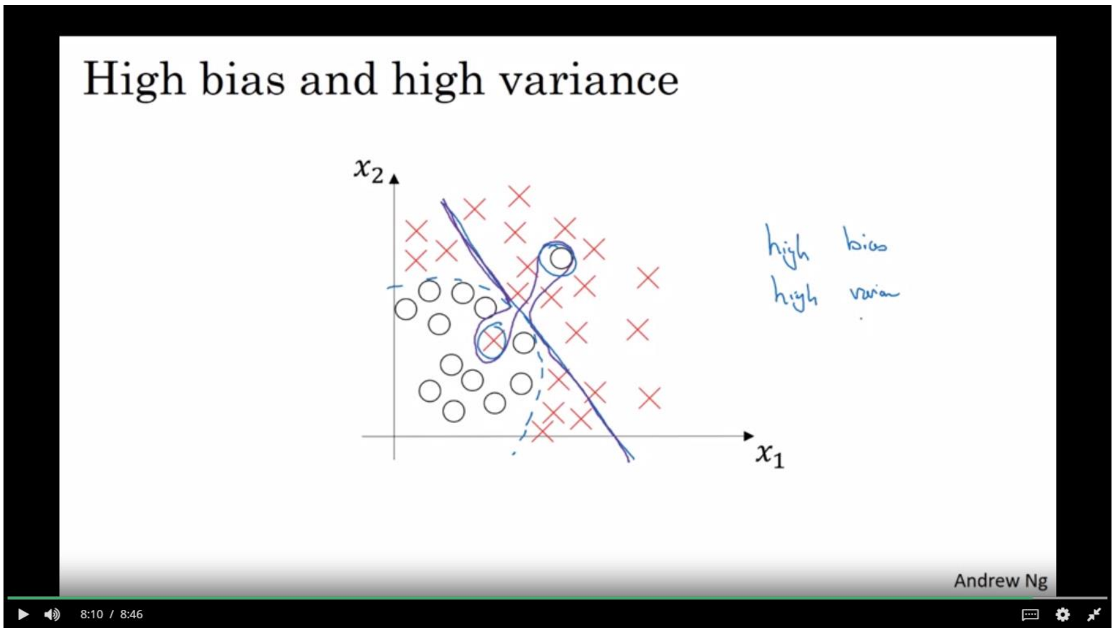
Cat classification

Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high bias & high variance	low bias low variance

Human : ≈ 0%

Optimal (Bayes) error : ≈ 0%

Andrew Ng



In the design:

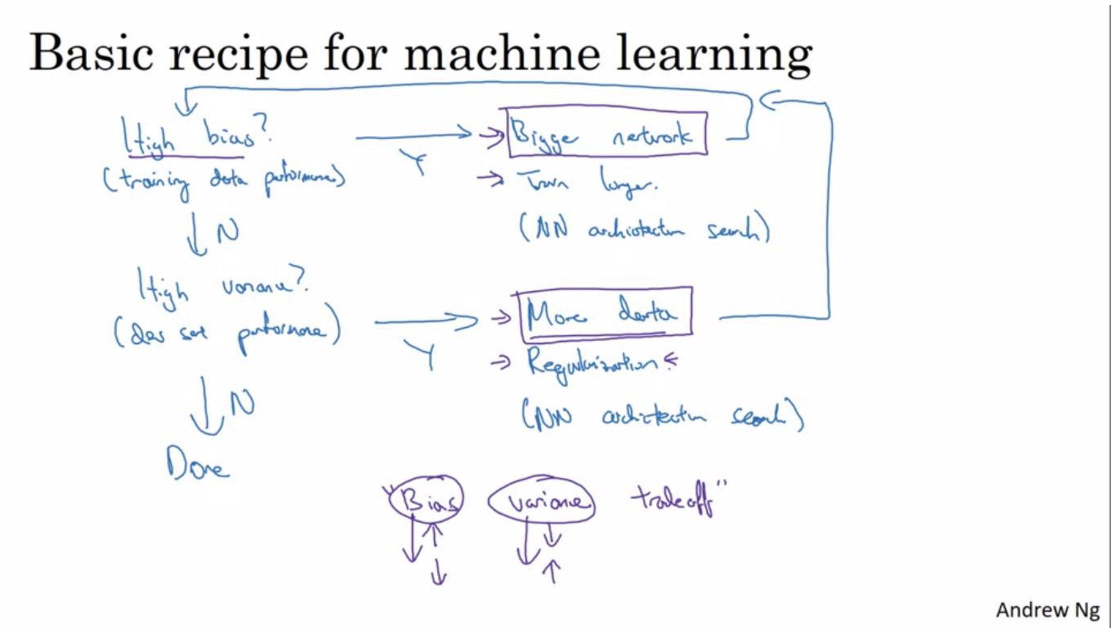
If high bias=underfitting the train set, high variance=overfitting train set.

Train set error: more high, high bias

Dev set error: more high, high variance

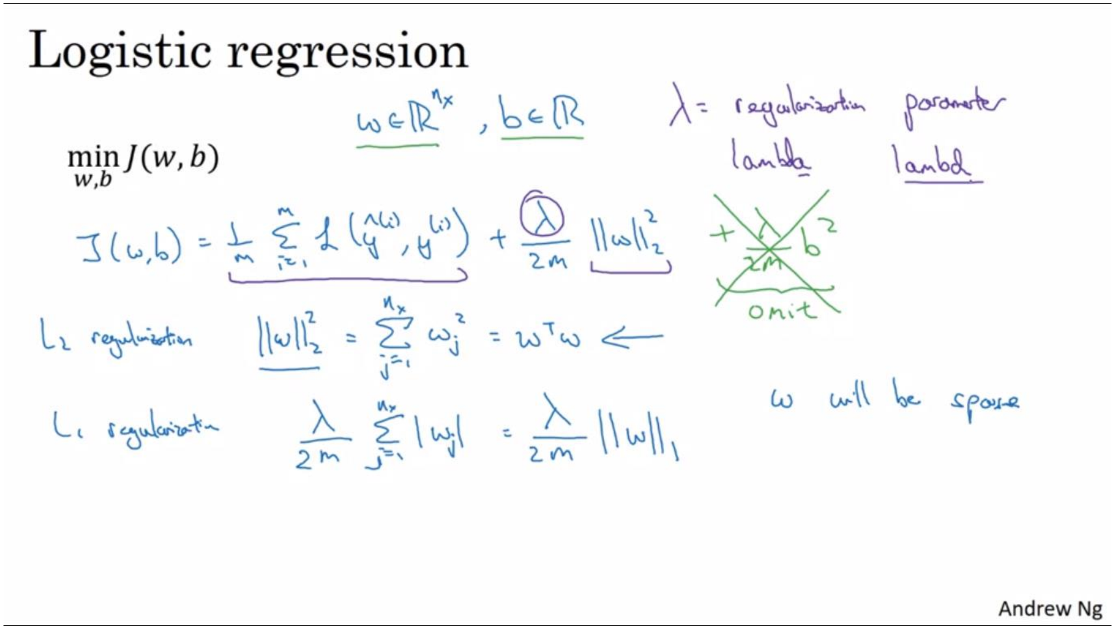
N.B. This is based on human classifier, as the error=0

Recipe for solving both high bias and high variance:



There were a tradeoff between bias and variance, but large network and data set solves this(however with longer time).

Regularization for high variance:



Logistic regression

$\min_{w,b} J(w,b)$ $\quad w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$ $\quad \lambda$ = regularization parameter

lambda lambd

$$J(w,b) = \underbrace{\frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})} + \frac{\lambda}{2m}\underbrace{\|w\|_2^2} \quad \cancel{+\frac{\lambda}{2m}b^2}$$

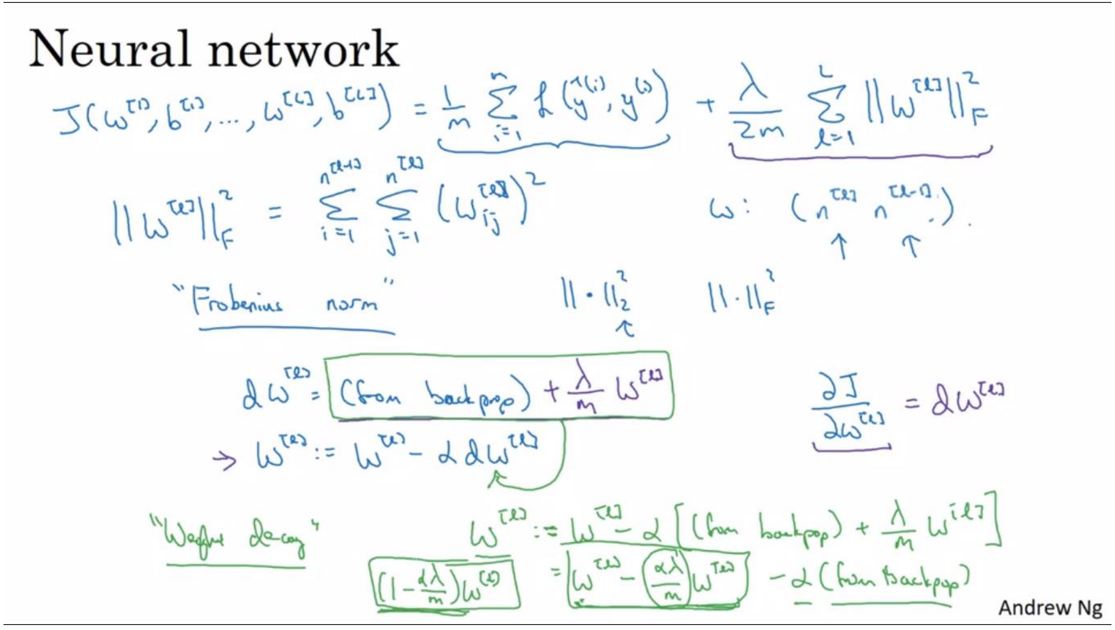
omit

L₂ regularization $\quad \|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w \leftarrow$

L₁ regularization $\quad \frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m}\|w\|_1$ $\quad$ w will be sparse

Andrew Ng

Lambda is a hybrid parameter modified in iterations by dev set. We use l2 regularization more than l1, as the later is sparse (have many zeros).



Neural network

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

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

"Frobenius norm" $\quad \|\cdot\|_2^2 \quad \|\cdot\|_F^2$

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

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

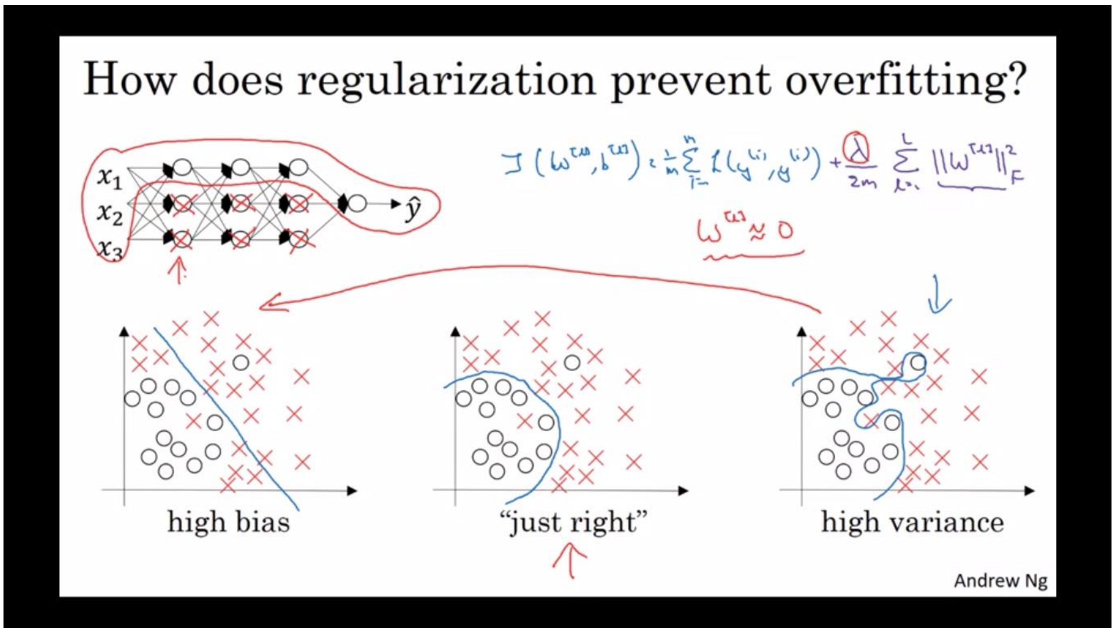
"Weight decay" $\quad w^{[l]} := w^{[l]} - \alpha\left[(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}\right]$

$$\boxed{\left(1 - \frac{\alpha\lambda}{m}\right) w^{[l]}} \qquad = \boxed{w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]}} - \alpha(\text{from backprop})$$

Andrew Ng

We use Forbenius norm regularization in NN instead of L2 in logistic regression. This will lead to Weight Decay formula in W after back propagation.

How does Regularization reduces overfitting (high variance):

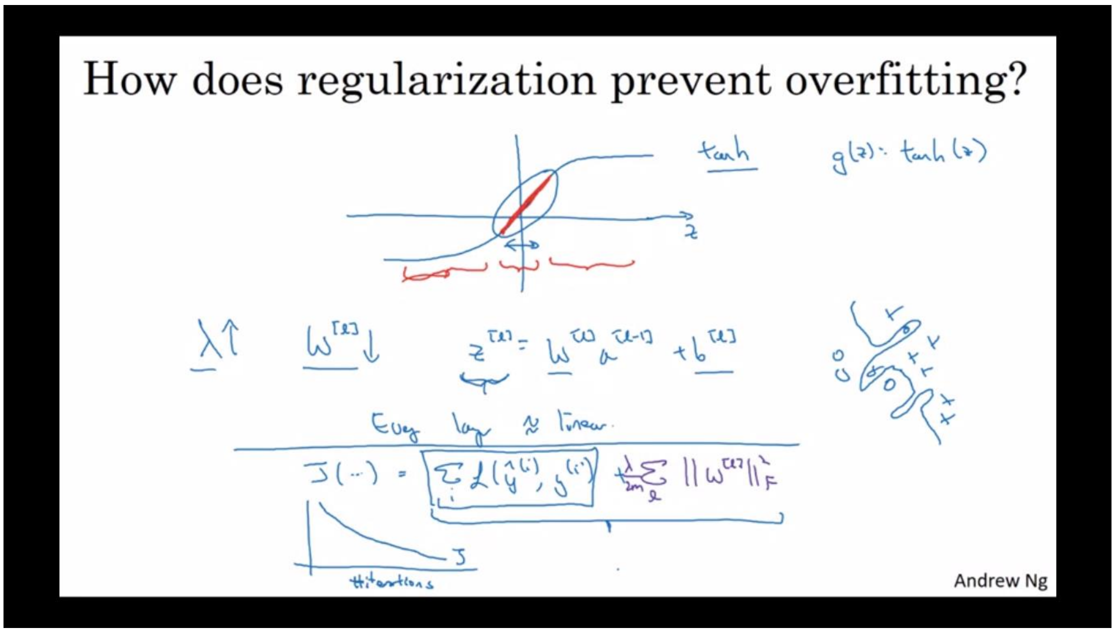


How does regularization prevent overfitting?

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

$w^{[l]} \approx 0$

Higher lambda, lower W (which is not practical). So, lower Z, which tend to make activation functions linear. This is not practical for Deep NN at all.



How does regularization prevent overfitting?

$g(z) = \tanh(z)$

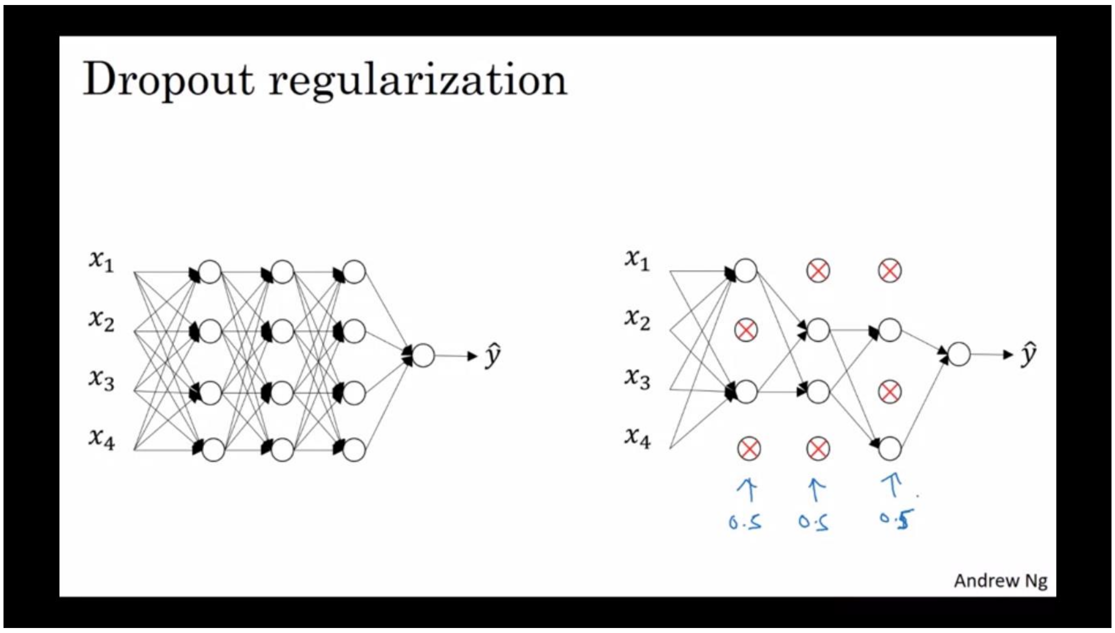
$\lambda \uparrow \quad w^{[l]} \downarrow$

$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$

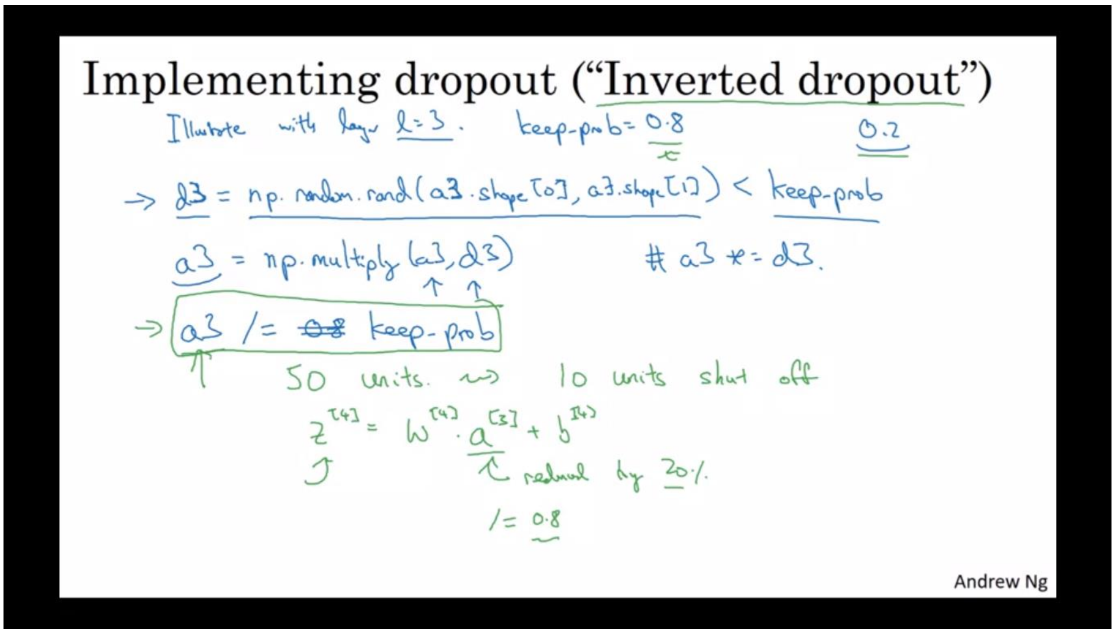
Every layer ≈ linear.

$J(\dots) = \boxed{\sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)})} + \frac{\lambda}{2m}\sum_l \|w^{[l]}\|_F^2$

Drop-out regularization:


Dropout regularization

For forward propagation, Inverted Drop-out:


Implementing dropout ("Inverted dropout")

Illustrate with layer $l=3$. keep-prob = 0.8   0.2

→ d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep-prob

a3 = np.multiply(a3, d3)   # a3 *= d3.

→ a3 /= ~~0.8~~ keep-prob

50 units. ⇝ 10 units shut off

$z^{[4]} = w^{[4]} \cdot a^{[3]} + b^{[4]}$

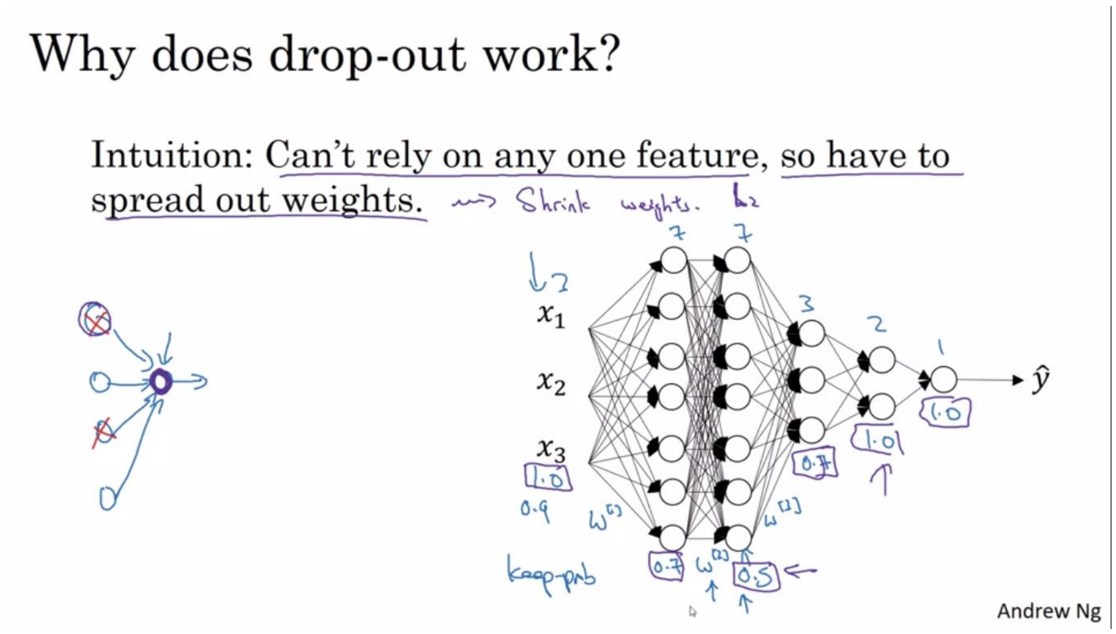
reduced by 20%.

/= 0.8

Andrew Ng

Famous type is Inverted Reg. We will use also d3 for back propagation.

Drop-out is not implemented in test time, as this will add noise.



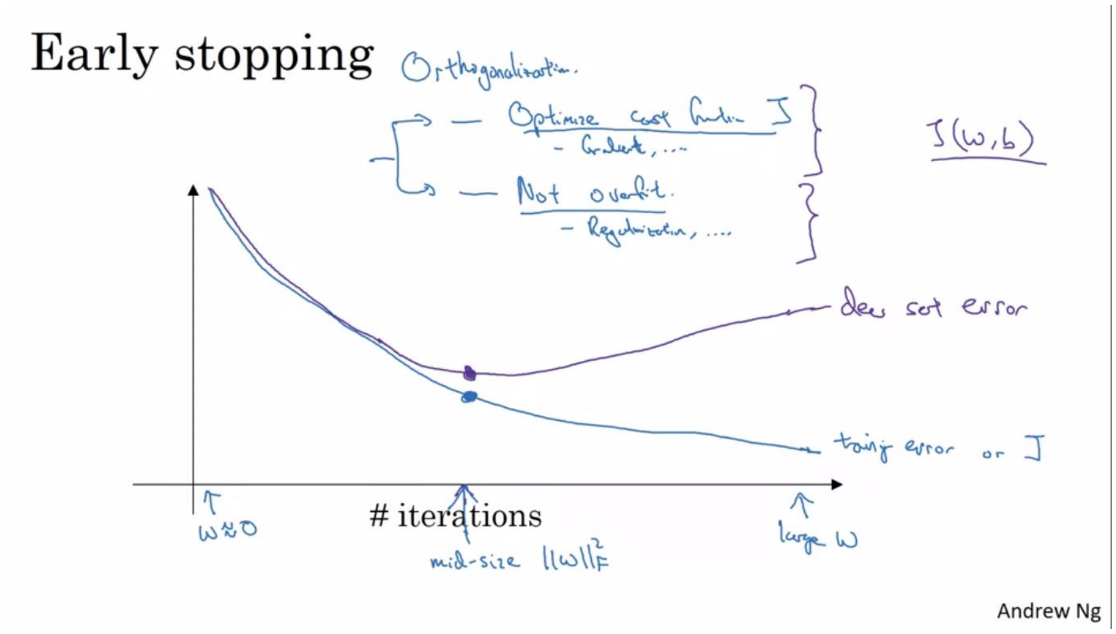
Why does drop-out work?

Intuition: Can't rely on any one feature, so have to spread out weights. ⟿ Shrink weights. L_2

The drop-out is highly used in larger dimensions of hidden layers, and not used for both inputs and output layers (for importance of each element existence).

Downside: cost function is hard to be defined.

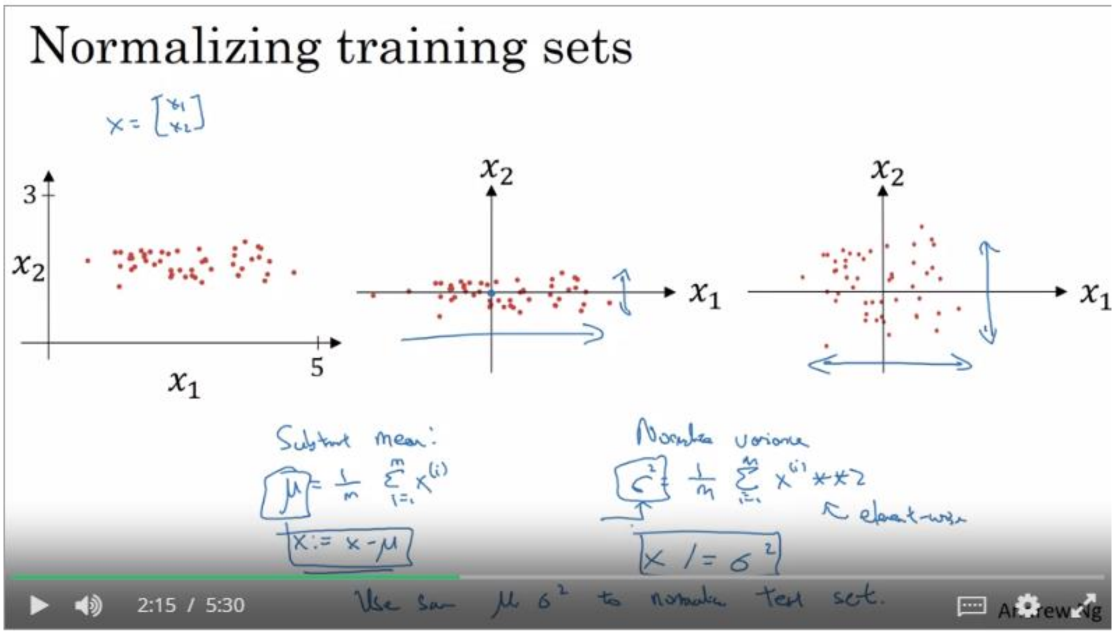
Two methods are now used to reduce both cost function and overfitting:



Early stopping

1- Early stopping: this is an orothogonalization method, where we use gradient descent to reduce cost function and regularization to reduce overfitting.
2- Using l2 regularization with large iterations for lambda.

Normalizing inputs will make the cost function diverge faster:



Normalizing training sets

$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

Subtract mean:

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

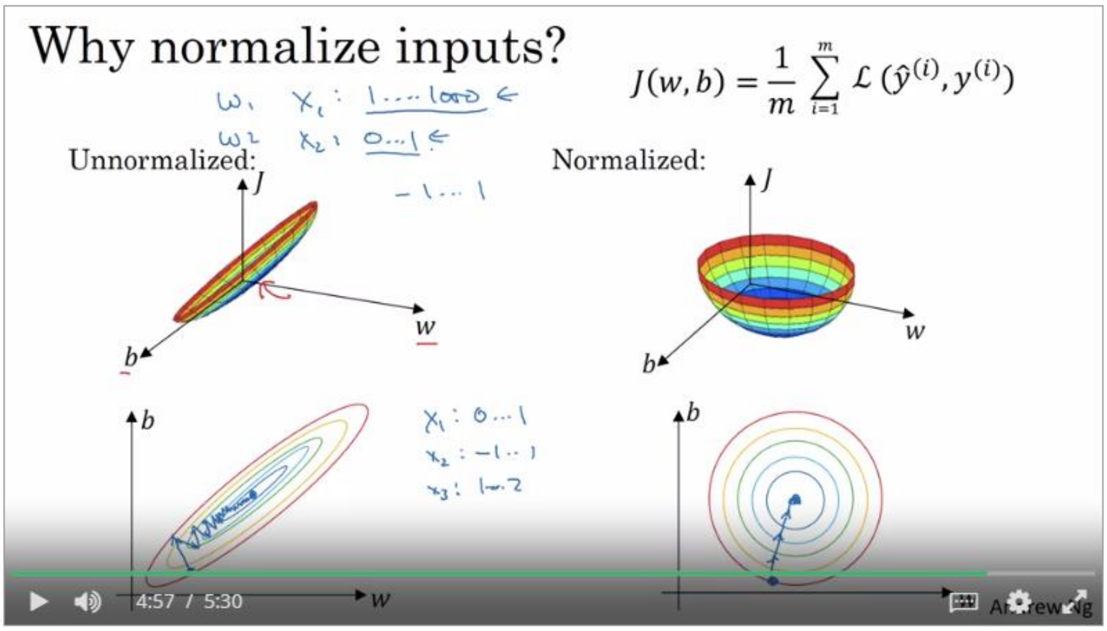
$\boxed{x := x - \mu}$

Normalize variance

$\boxed{\sigma^2} = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} {**} 2$ ← element-wise

$\boxed{x \mathrel{/}= \sigma^2}$

Use same μ σ^2 to normalize test set.



Why normalize inputs?

$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

w_1 x_1: 1.... 1000 ←

w_2 x_2: 0...1 ←

−1 ... 1

Unnormalized:

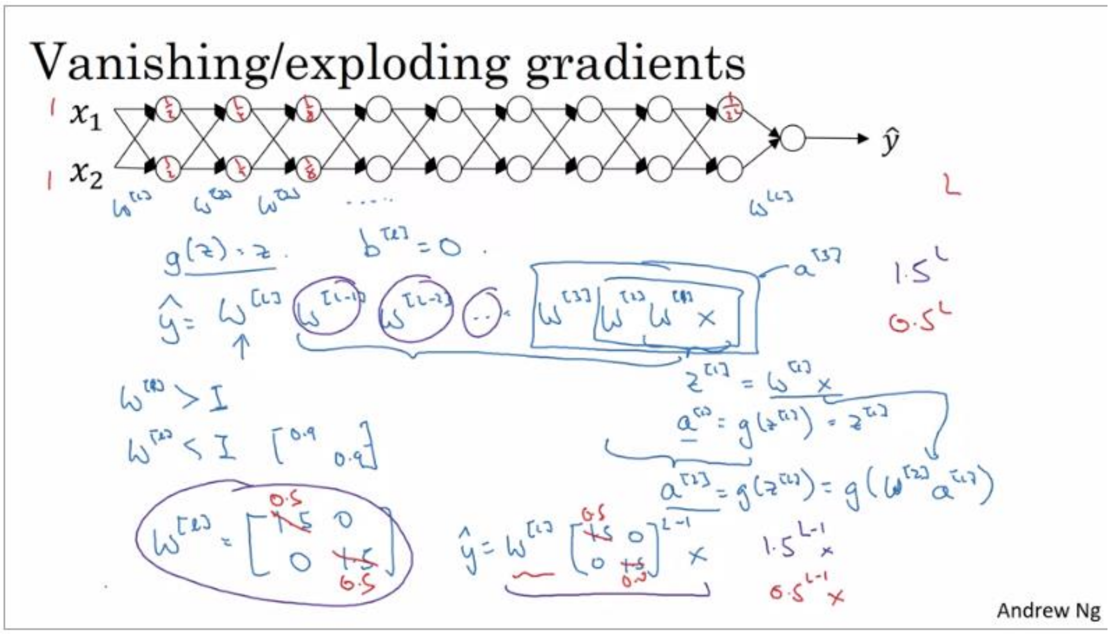
Normalized:

x_1: 0...1

x_2: −1..1

x_3: 1...2

Gradients can increase or decrease exponentially:



Vanishing/exploding gradients

$g(z) = z$, $b^{[l]} = 0$

$\hat{y} = W^{[L]} W^{[L-1]} W^{[L-2]} \cdots W^{[3]} W^{[2]} W^{[1]} x$

$z^{[1]} = W^{[1]} x$

$a^{[1]} = g(z^{[1]}) = z^{[1]}$

$a^{[2]} = g(z^{[2]}) = g(W^{[2]} a^{[1]})$

$W^{[l]} > I$

$W^{[l]} < I \quad \begin{bmatrix} 0.9 & \\ & 0.9 \end{bmatrix}$

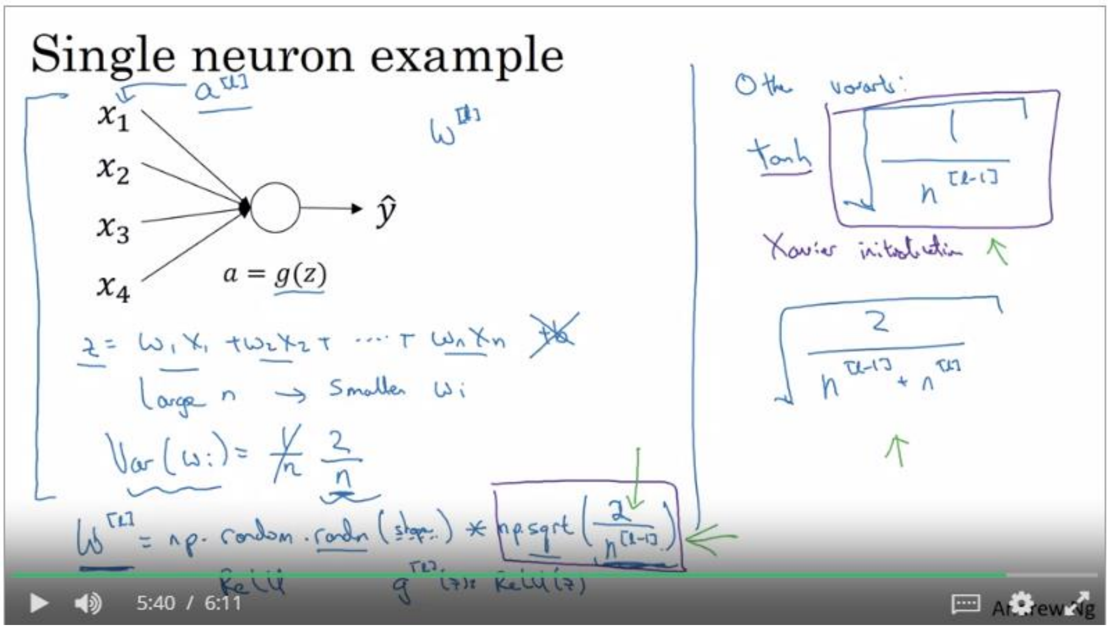
$W^{[l]} = \begin{bmatrix} \overset{0.5}{\cancel{1.5}} & 0 \\ 0 & \underset{0.5}{\cancel{1.5}} \end{bmatrix}$

$\hat{y} = W^{[L]} \begin{bmatrix} \overset{0.5}{\cancel{1.5}} & 0 \\ 0 & \underset{0.5}{\cancel{1.5}} \end{bmatrix}^{L-1} x$

1.5^{L}, 0.5^{L}, $1.5^{L-1} x$, $0.5^{L-1} x$

Andrew Ng

To solve this, we initialize the weights such that it will not be more (greater or less) than 1. This depends on the type of the activation function:



Single neuron example

$a = g(z)$

$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n \; \cancel{+b}$

Large n → Smaller w_i

$Var(w_i) = \cancel{\frac{1}{n}} \; \frac{2}{n}$

$W^{[l]} = $ np.random.randn(shape) * np.sqrt$\left(\frac{2}{n^{[l-1]}}\right)$

ReLU $\quad g^{[l]}(z) = $ ReLU(z)

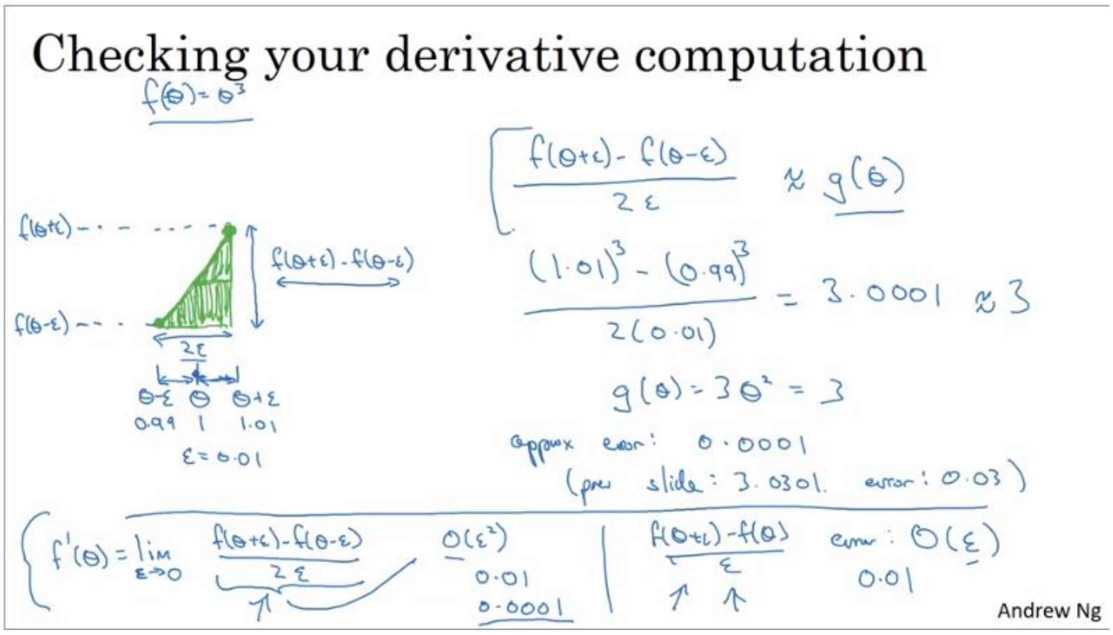
Other variants:

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

Xavier initialization

$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

To check the gradient computation, we have to do the numerical method:



Checking your derivative computation

$f(\theta)=\theta^3$

$$\left[\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon} \approx \underline{g(\theta)}\right.$$

$$\frac{(1.01)^3-(0.99)^3}{2(0.01)} = 3.0001 \approx 3$$

$$g(\theta)=3\theta^2=3$$

approx error: 0.0001

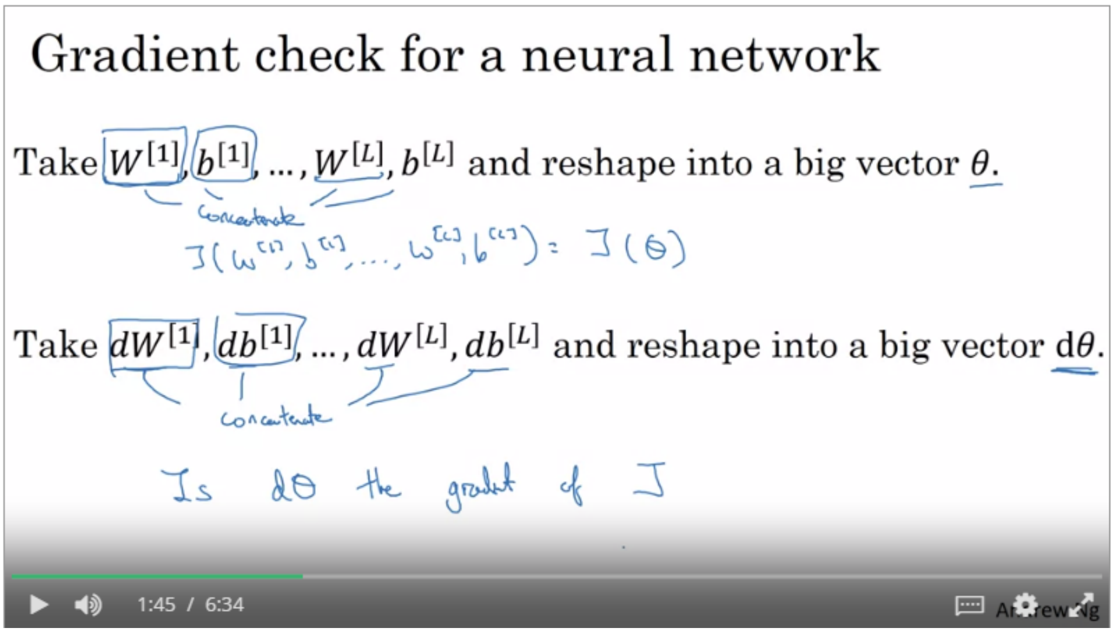
(prev slide: 3.0301. error: 0.03)

$$\left\{ f'(\theta)=\lim_{\varepsilon\to 0}\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon} \quad O(\varepsilon^2) \quad 0.01 \quad 0.0001 \quad \middle| \quad \frac{f(\theta+\varepsilon)-f(\theta)}{\varepsilon} \quad \text{error}: O(\varepsilon) \quad 0.01 \right.$$

Andrew Ng

$f(\theta+\varepsilon)$
$f(\theta-\varepsilon)$
$f(\theta+\varepsilon)-f(\theta-\varepsilon)$
2ε
$\theta-\varepsilon$ θ $\theta+\varepsilon$
0.99 1 1.01
$\varepsilon=0.01$

First, we make a large variable Ө:



Gradient check for a neural network

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

concatenate

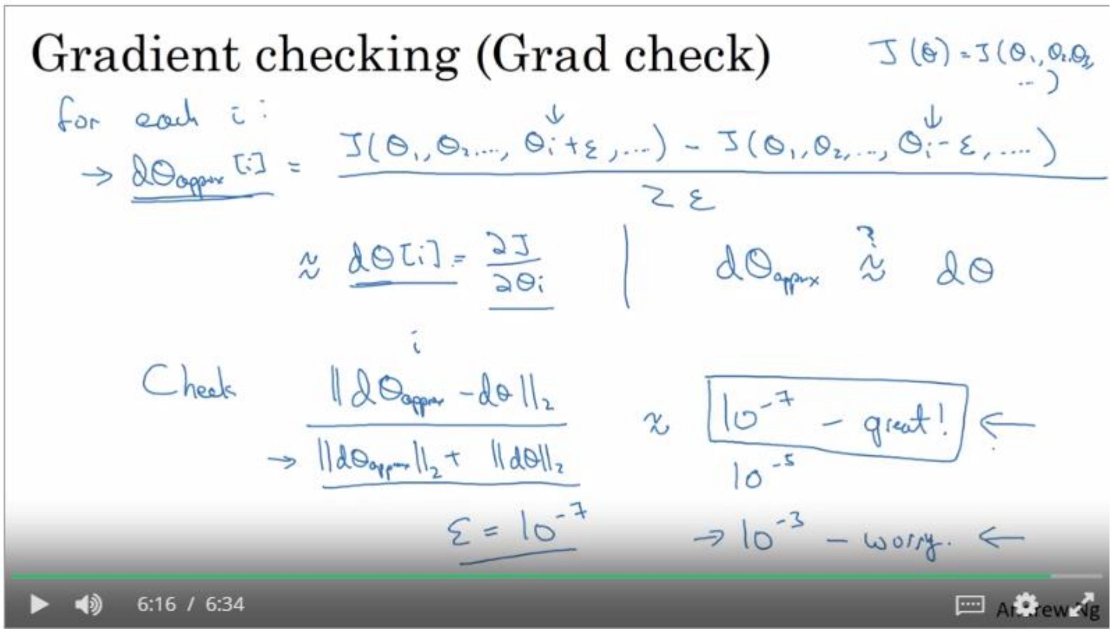
$J(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) = J(\theta)$

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

concatenate

Is $d\theta$ the gradient of J

Then do the check procedure as follows:



Gradient checking (Grad check)

$J(\theta) = J(\theta_1, \theta_2, \theta_3, \dots)$

for each i:

$$\rightarrow \underline{d\theta_{approx}[i]} = \frac{J(\theta_1, \theta_2, \dots, \overset{\downarrow}{\theta_i} + \varepsilon, \dots) - J(\theta_1, \theta_2, \dots, \overset{\downarrow}{\theta_i} - \varepsilon, \dots)}{2\varepsilon}$$

$$\approx \underline{d\theta[i]} = \underline{\frac{\partial J}{\partial \theta_i}} \quad \Big| \quad d\theta_{approx} \overset{?}{\approx} d\theta$$

Check $$\frac{\|d\theta_{approx} - d\theta\|_2}{\rightarrow \|d\theta_{approx}\|_2 + \|d\theta\|_2} \approx \boxed{10^{-7} \text{ - great!}} \leftarrow$$

10^{-5}

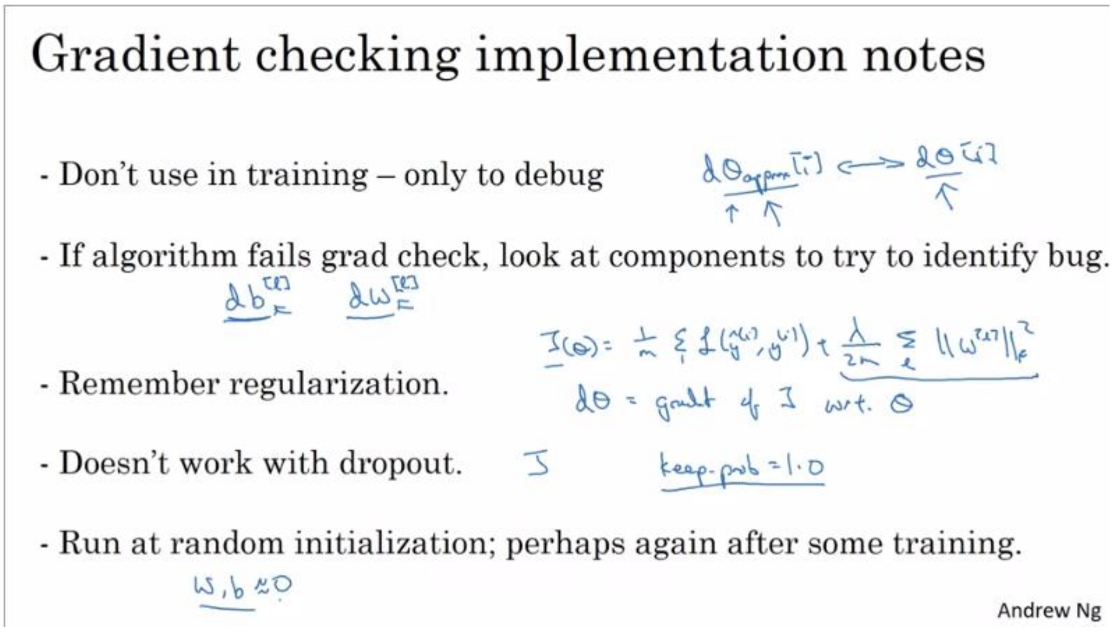
$\underline{\varepsilon = 10^{-7}}$ $\rightarrow 10^{-3}$ - worry. $\leftarrow$

6:16 / 6:34

Andrew Ng

If the check equation equals Ɛ or very near, so it is ok. Otherwise, try to debug the Ɵ(i) variables.

Important notes on applying gradient descent:



Gradient checking implementation notes

- Don't use in training – only to debug $\quad d\theta_{approx}[i] \longleftrightarrow d\theta[i]$
- If algorithm fails grad check, look at components to try to identify bug. $\quad \underline{db^{[l]}} \quad \underline{dW^{[l]}}$
- Remember regularization. $\quad \underline{J(\theta)} = \frac{1}{m} \sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \underbrace{\frac{\lambda}{2m} \sum_l \|W^{[l]}\|_F^2}$ $\quad d\theta$ = grad of J wrt. θ
- Doesn't work with dropout. $\quad J \quad$ $\underline{\text{keep-prob} = 1.0}$
- Run at random initialization; perhaps again after some training. $\quad \underline{W, b} \approx 0$

Andrew Ng

N.B. for drop-out regularization: we first run the gradient check without drop-out, then apply the drop-out.