

Copyright Notice

These slides are distributed under the Creative Commons License.

[DeepLearning.AI](#) makes these slides available for educational purposes. You may not use or distribute these slides for commercial purposes. You may make copies of these slides and use or distribute them for educational purposes as long as you cite [DeepLearning.AI](#) as the source of the slides.

For the rest of the details of the license, see <https://creativecommons.org/licenses/by-sa/2.0/legalcode>

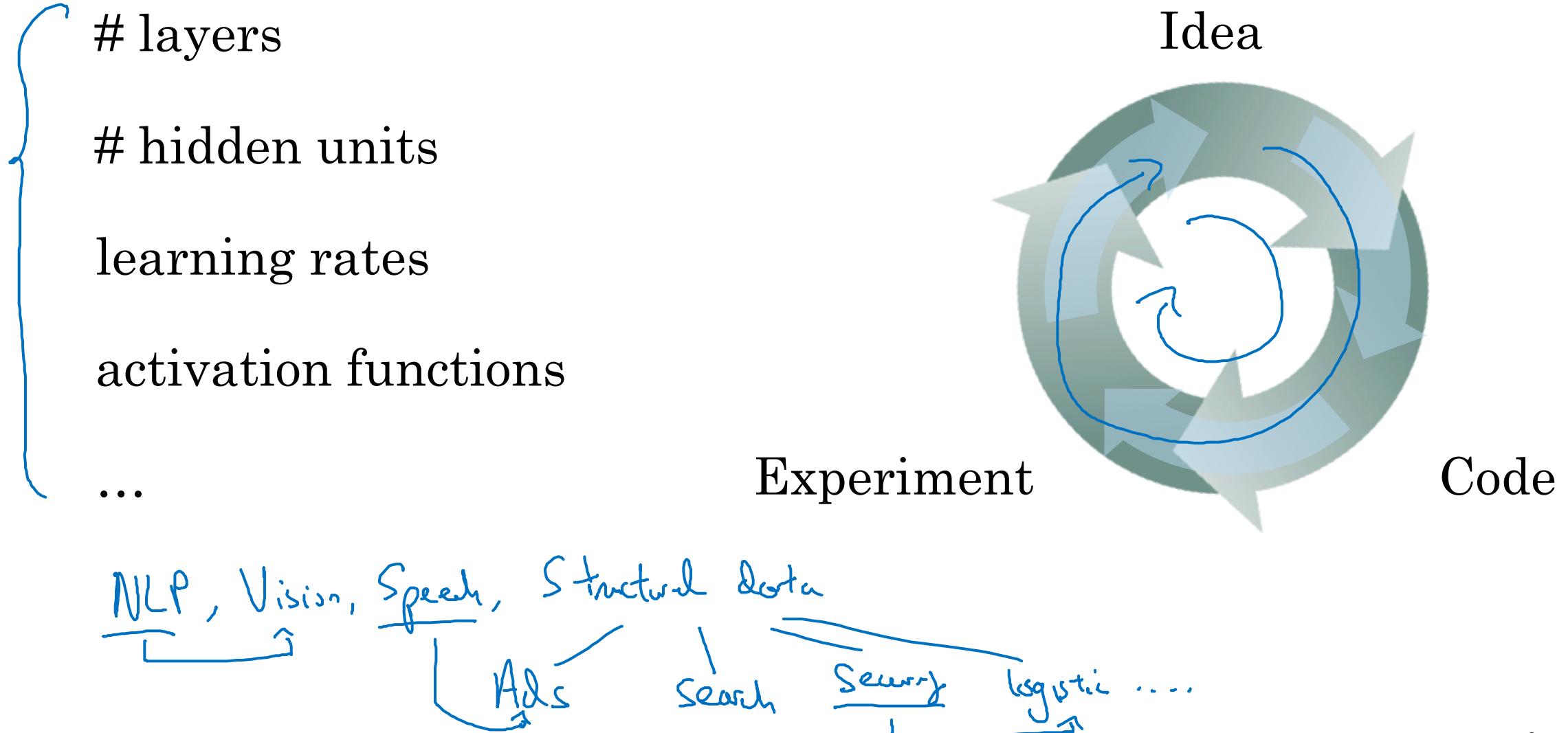


deeplearning.ai

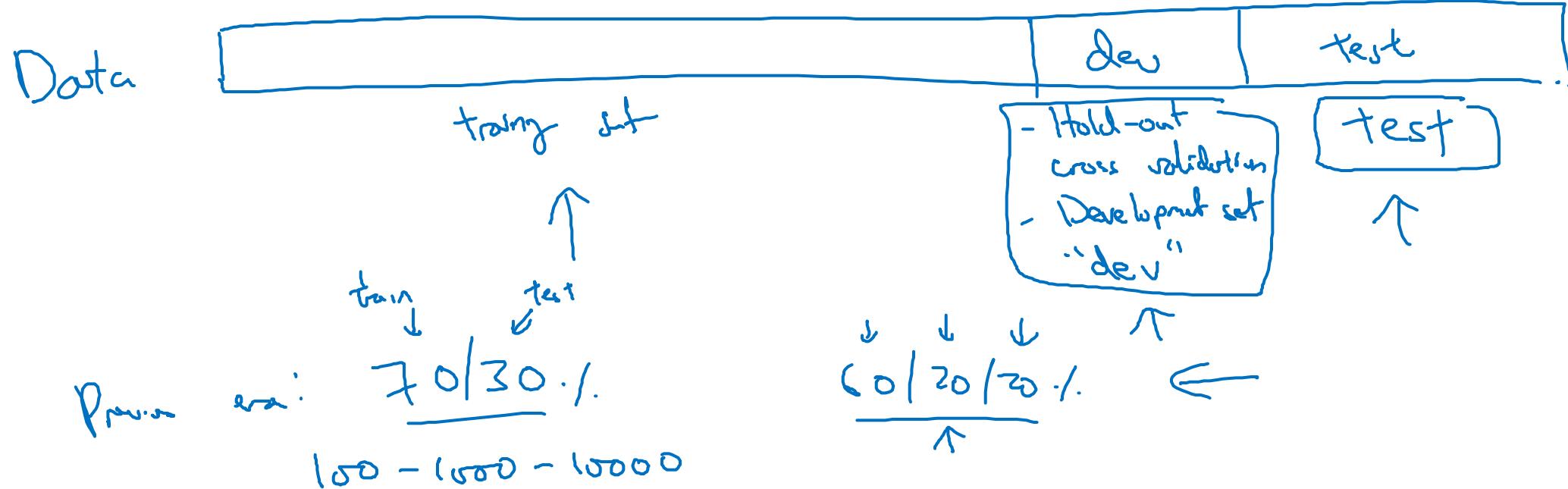
Setting up your
ML application

Train/dev/test
sets

Applied ML is a highly iterative process



Train/dev/test sets



Big data! 1,000,000

10,000 10,000

98 / 1 / 1 .%

99.5 { 25 / 25
 { 4 { 1 .%

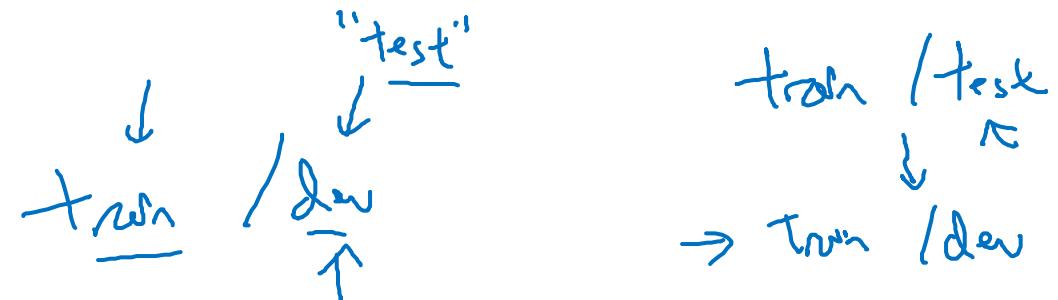
Mismatched train/test distribution

Conts

Training set:
Cat pictures from }
webpages

Dev/test sets:
Cat pictures from }
users using your app

→ Make sure dev and test come from same distribution.



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

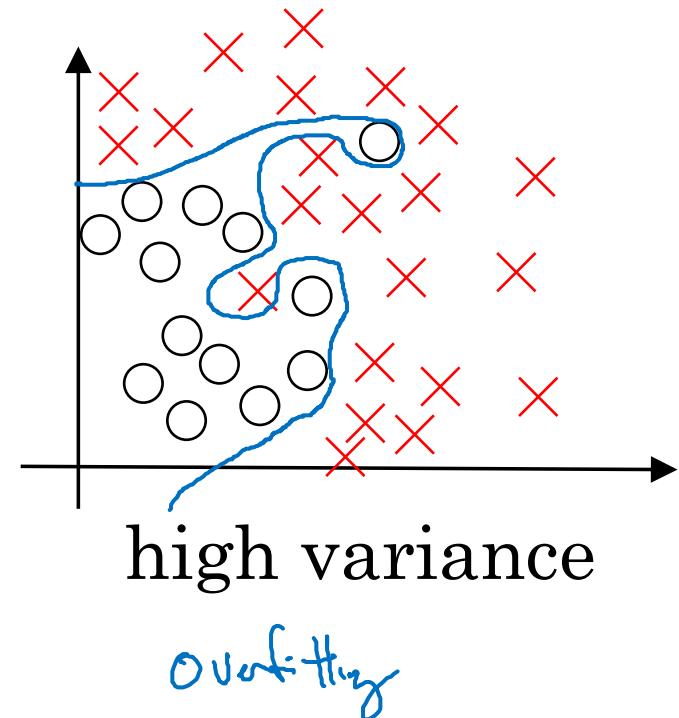
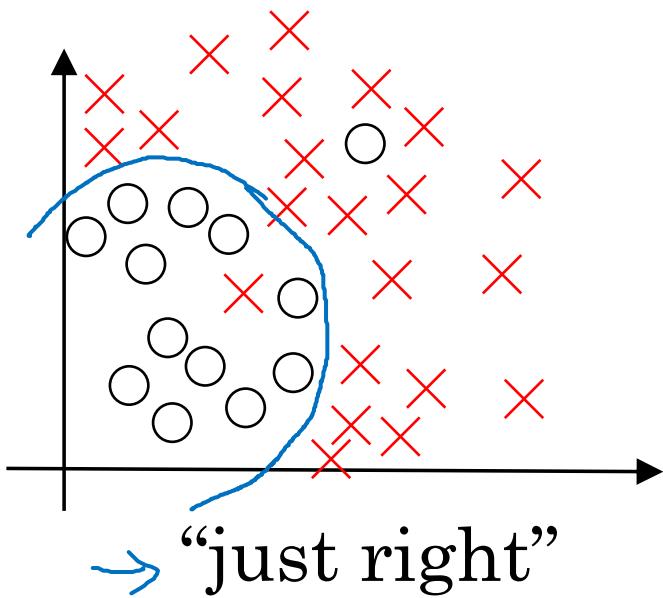
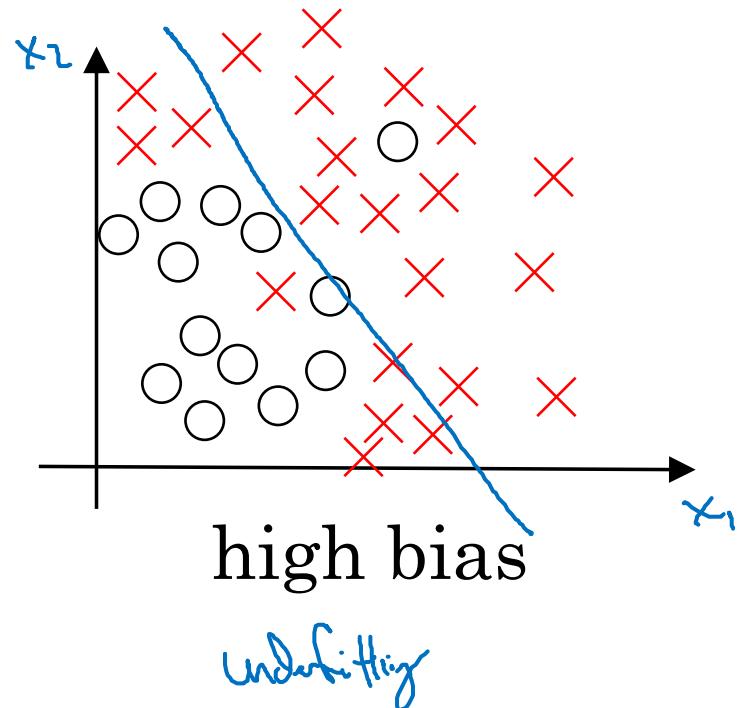


deeplearning.ai

Setting up your
ML application

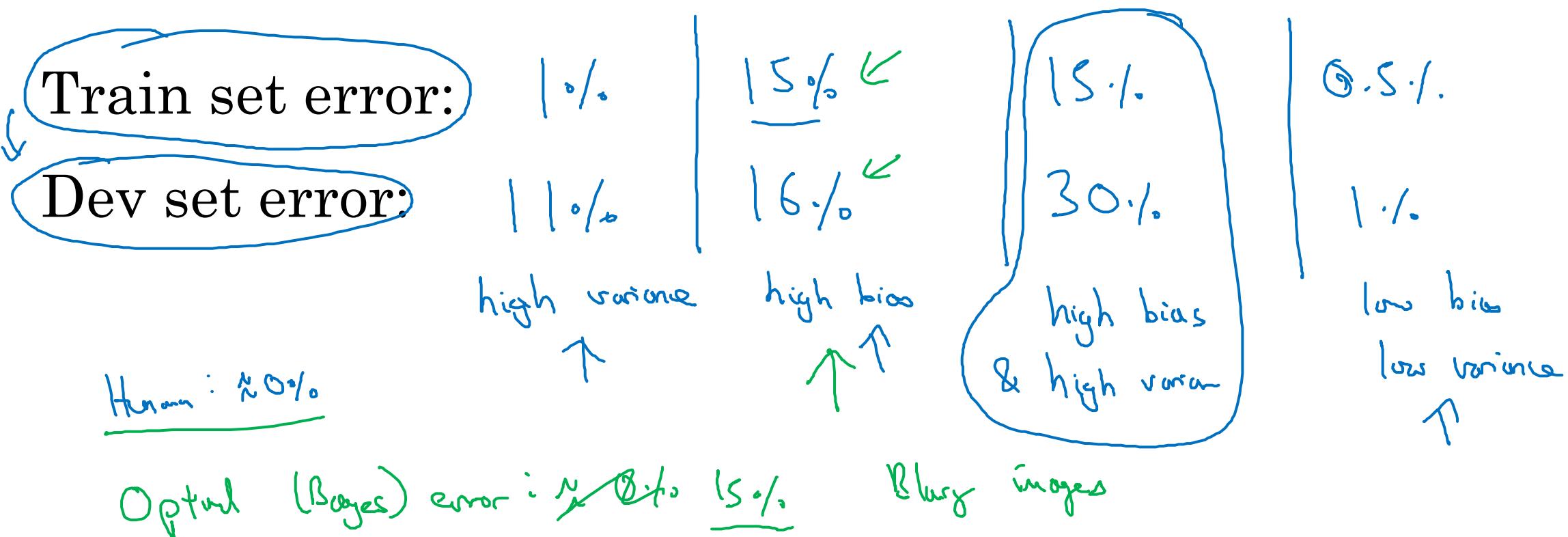
Bias/Variance

Bias and Variance

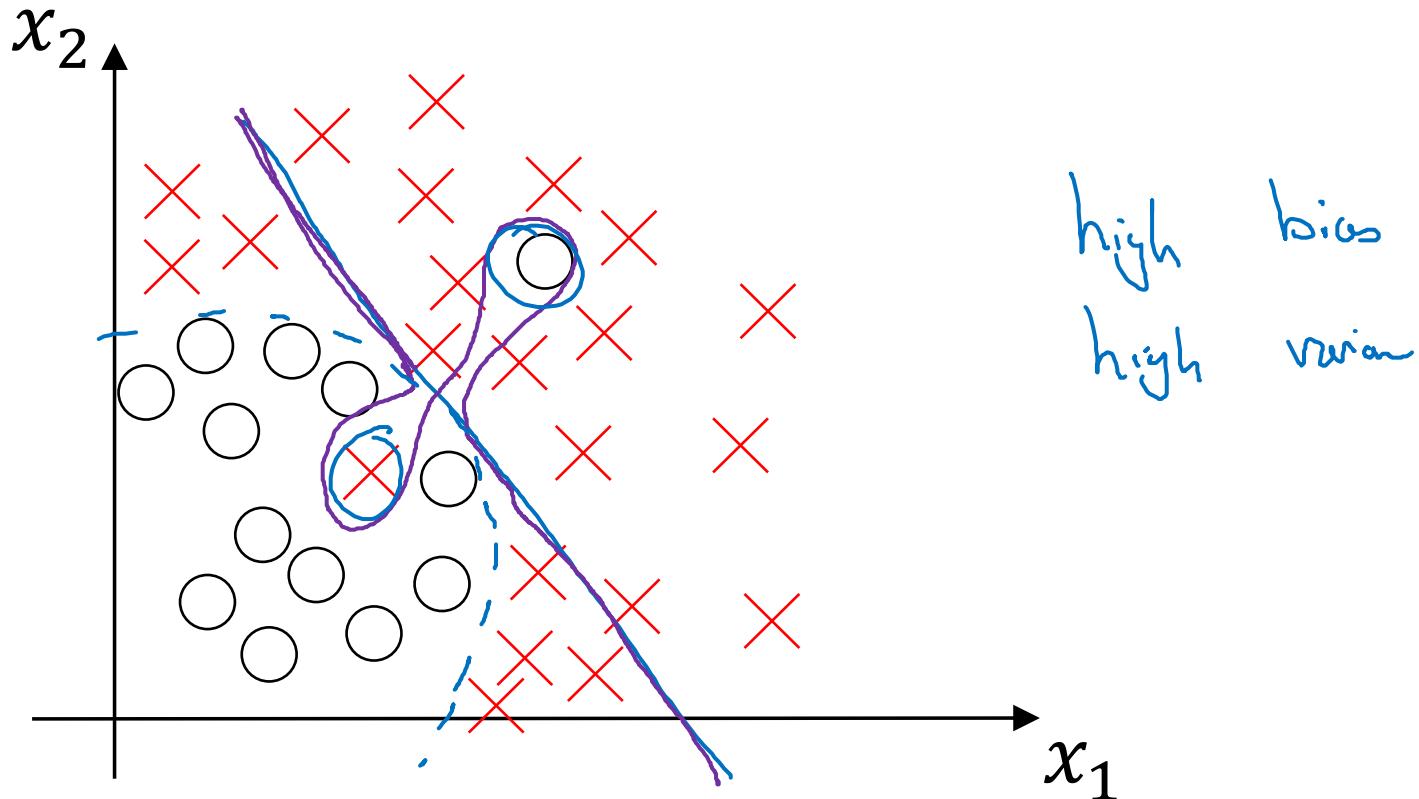


Bias and Variance

Cat classification



High bias and high variance



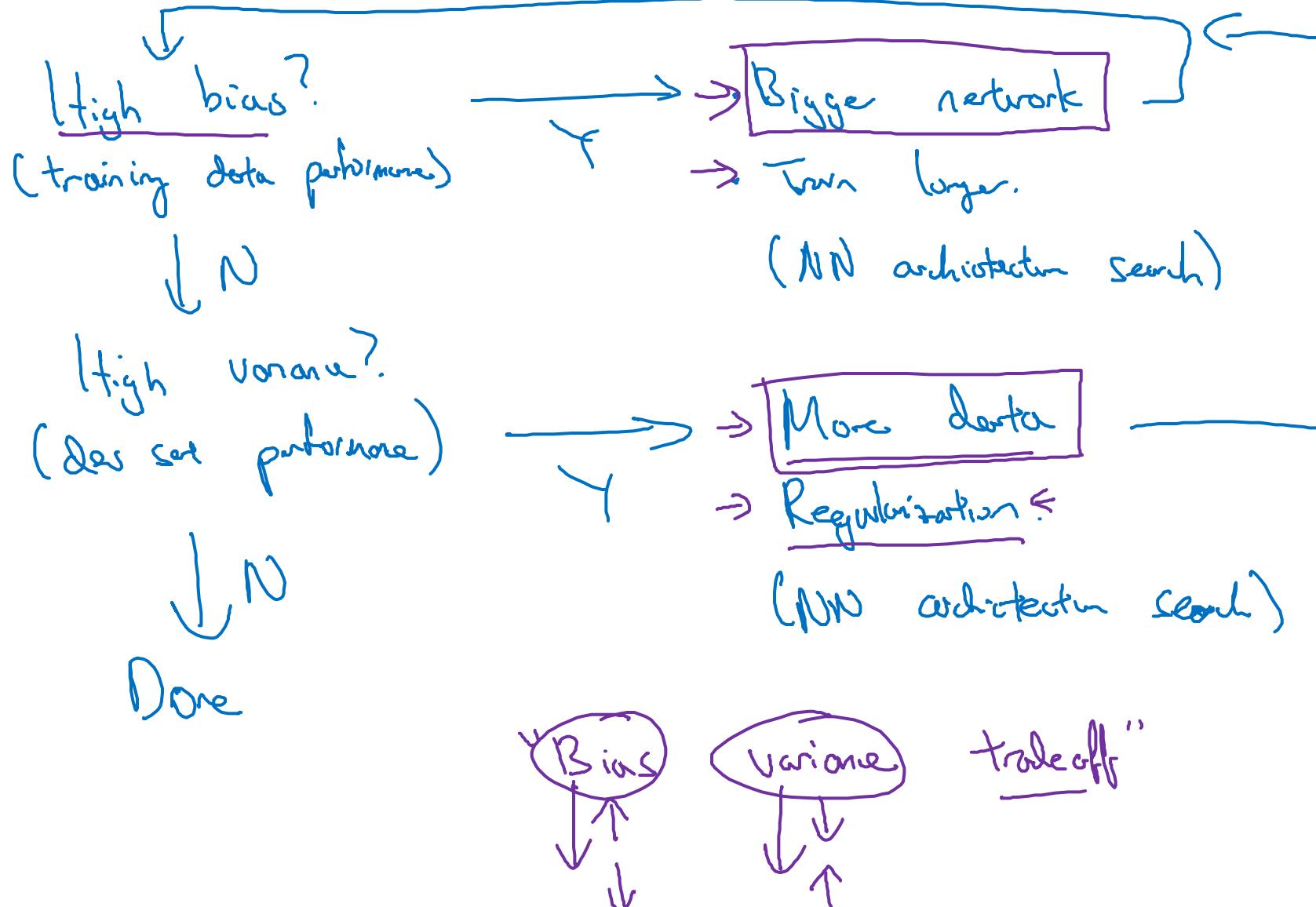


deeplearning.ai

Setting up your ML application

Basic “recipe” for machine learning

Basic recipe for machine learning





deeplearning.ai

Regularizing your
neural network

Regularization

Logistic regression

$$\min_{w,b} J(w, b)$$

$$w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$$

λ = regularization parameter
lambda lambd

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

$$+ \cancel{\frac{\lambda}{2m} b^2}$$

omit

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

L₁ regularization

$$\frac{\lambda}{2m} \sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m} \|w\|_1$$

w will be sparse

Neural network

$$\rightarrow J(w^{(1)}, b^{(1)}, \dots, w^{(L)}, b^{(L)}) = \underbrace{\frac{1}{m} \sum_{i=1}^m f(\hat{y}^{(i)}, y^{(i)})}_{\text{loss function}} + \underbrace{\frac{\lambda}{2m} \sum_{l=1}^L \|w^{(l)}\|_F^2}_{\text{regularization}}$$

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

$w^{(l)}: (n^{(l)}, n^{(l+1)})$

"Frobenius norm"

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

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

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

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

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

"Weight decay"

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

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

$$= \underbrace{\left(1 - \frac{\alpha \lambda}{m}\right)}_{< 1} \underbrace{w^{(l)}}_{> 0} - \alpha (\text{from backprop})$$

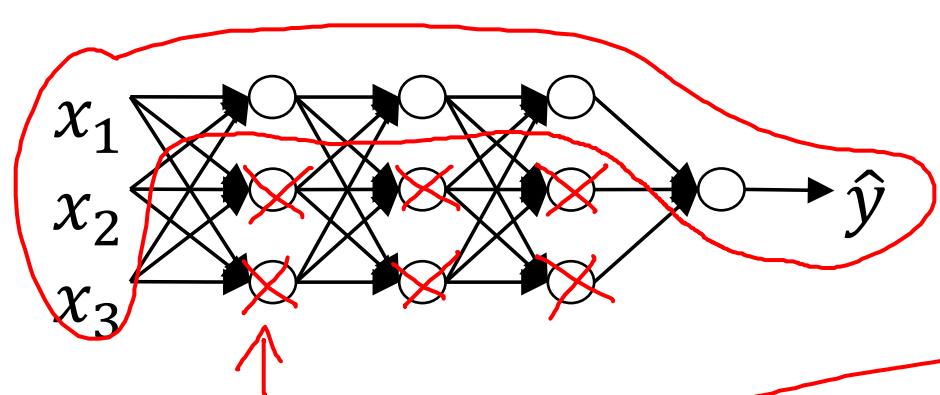


deeplearning.ai

Regularizing your neural network

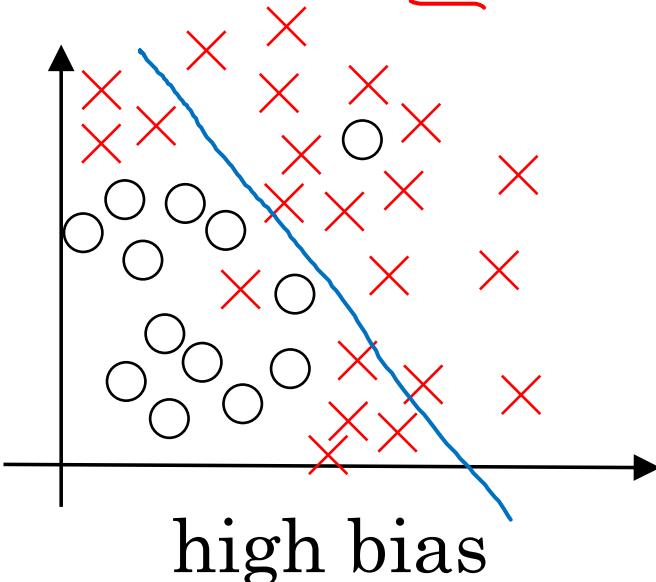
Why regularization reduces overfitting

How does regularization prevent overfitting?

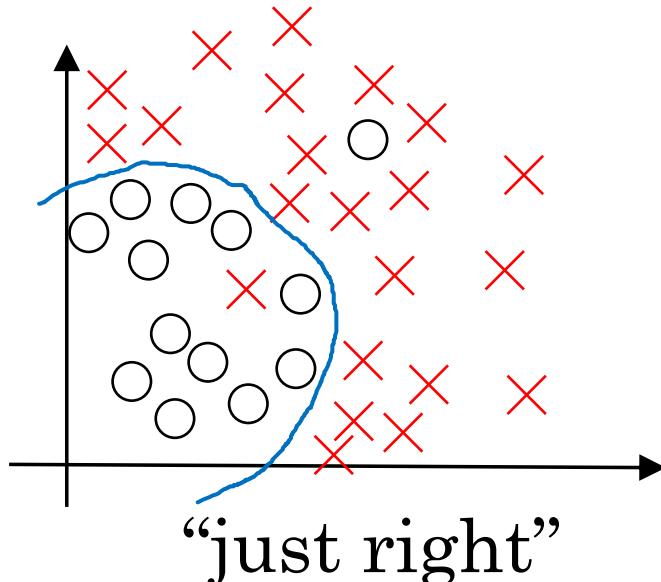


$$J(\boldsymbol{\theta}^{(m)}, \boldsymbol{b}^{(m)}) = \frac{1}{m} \sum_{i=1}^m \ell(y^{(i)}, \hat{y}^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^L \|\boldsymbol{w}^{(l)}\|_F^2$$

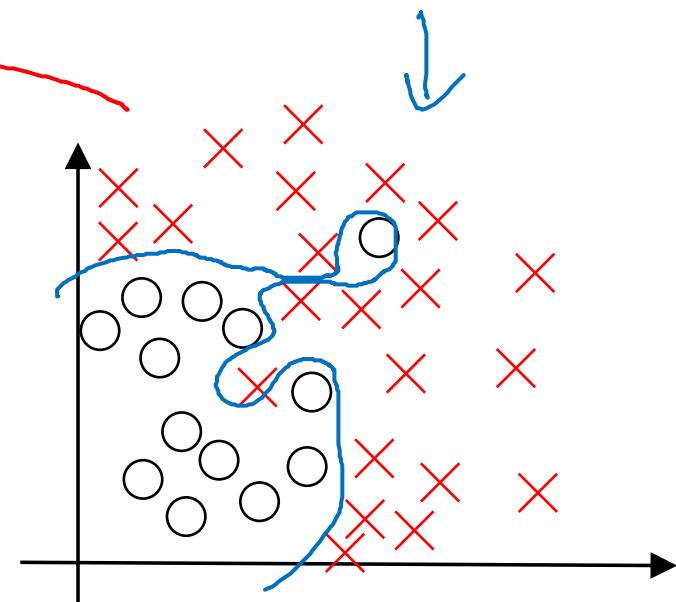
$\boldsymbol{w}^{(l)} \approx 0$



high bias

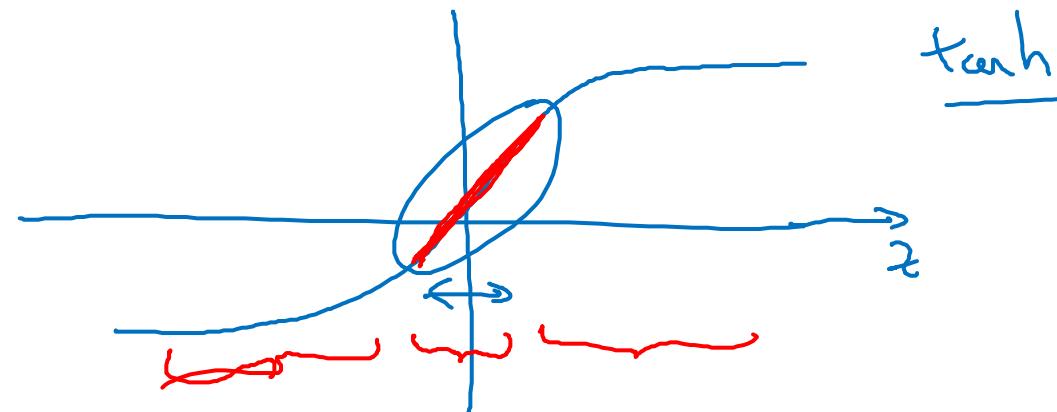


"just right"



high variance

How does regularization prevent overfitting?



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

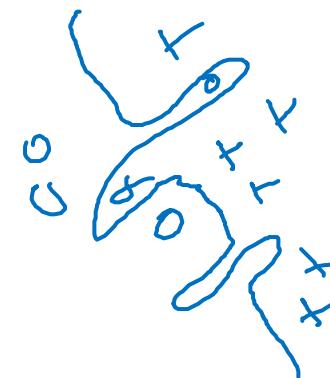
$$\lambda \uparrow$$

$$\underline{w^{[l]}} \downarrow$$

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

Every layer \approx linear.

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



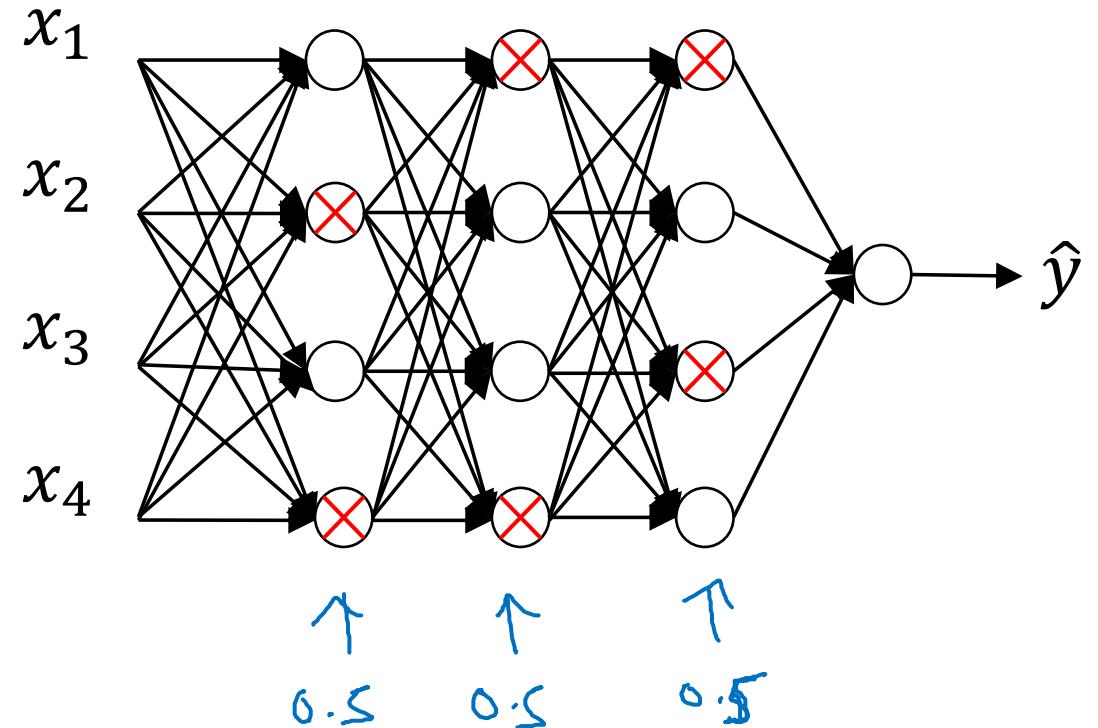
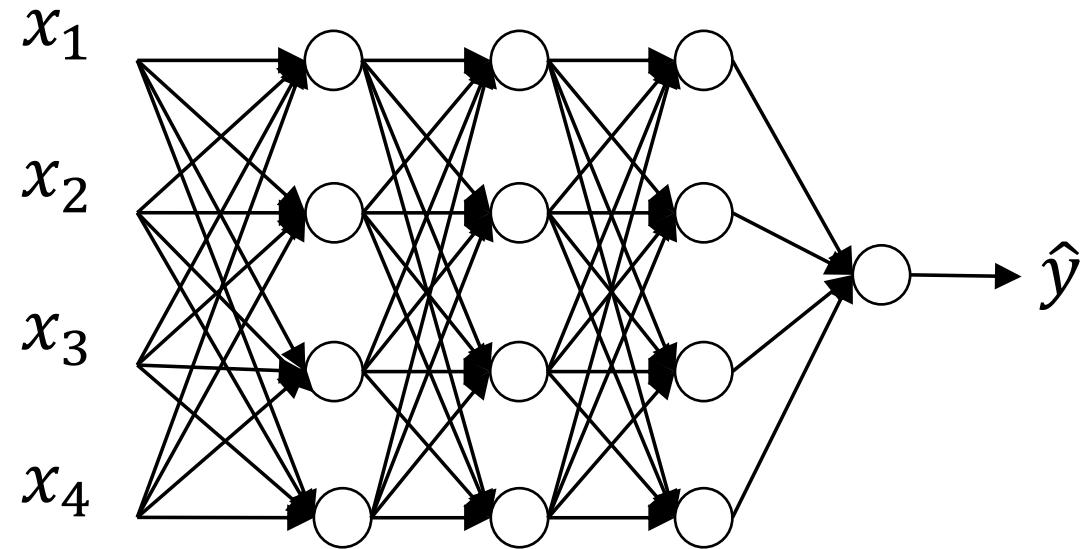


deeplearning.ai

Regularizing your
neural network

Dropout
regularization

Dropout regularization



Implementing dropout (“Inverted dropout”)

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

$$\rightarrow d_3 = \underbrace{\text{np.random.rand}(a_3.shape[0], a_3.shape[1]) < \text{keep-prob}}_{\text{d3}}$$

$$\underbrace{a_3}_{\text{a3}} = \text{np.multiply}(a_3, d_3) \quad \# a_3 * d_3.$$

$$\rightarrow \underbrace{a_3 /=\cancel{0.8} \text{ keep-prob}}_{\text{a3}} \leftarrow$$

50 units. \rightsquigarrow 10 units shut off

$$z^{(4)} = w^{(4)} \cdot \underbrace{\frac{a^{(3)}}{x}}_{\text{reduced by } 20\%} + b^{(4)}$$

Test

$$x = \underline{\underline{0.8}}$$

Making predictions at test time

$$a^{(0)} = X$$

No drop out.

$$\uparrow z^{(1)} = w^{(1)} \underline{a^{(0)}} + b^{(1)}$$

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

$$z^{(2)} = w^{(2)} \underline{a^{(1)}} + b^{(2)}$$

$$a^{(2)} = \dots$$

$$\downarrow \hat{y}$$

λ = keep-prob



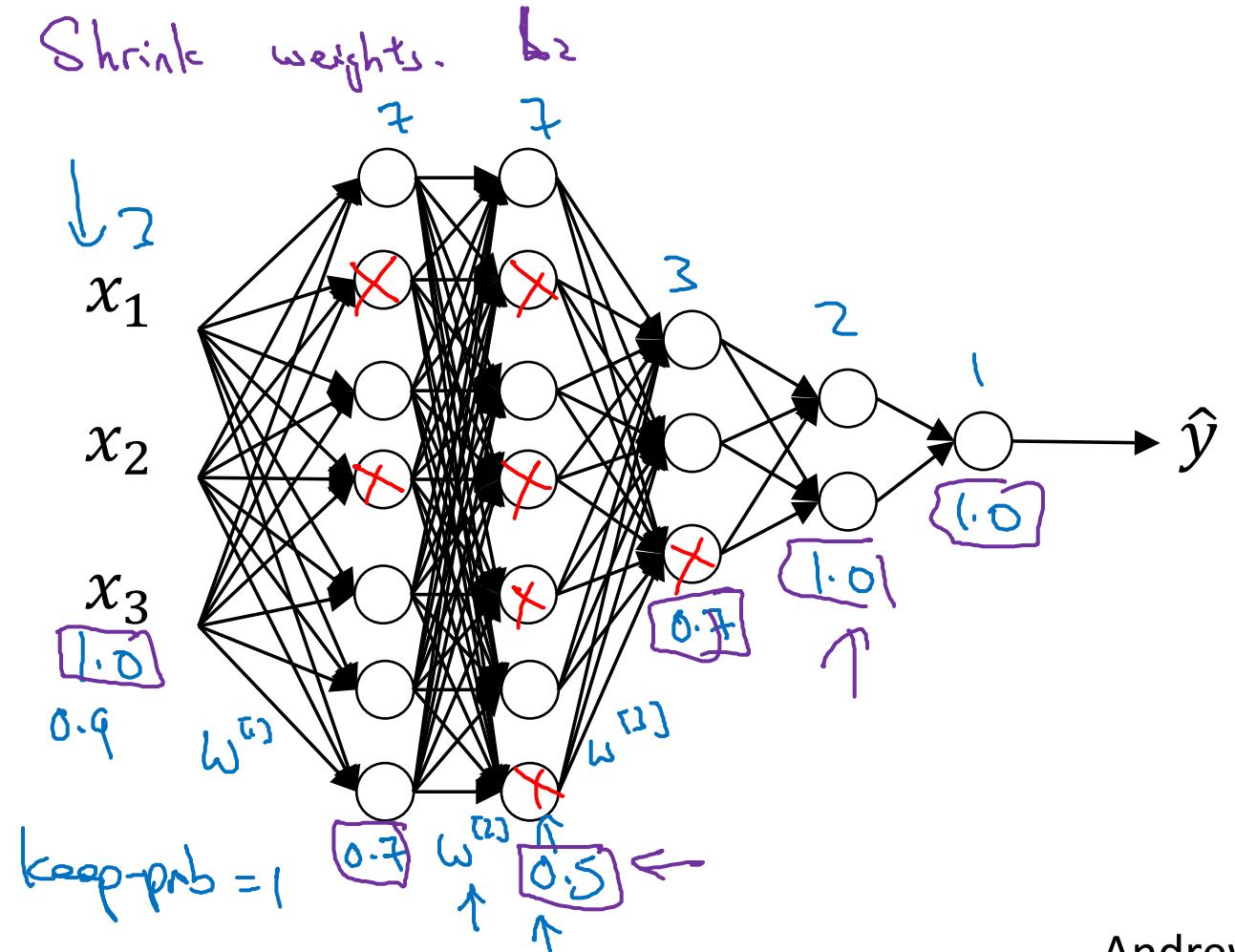
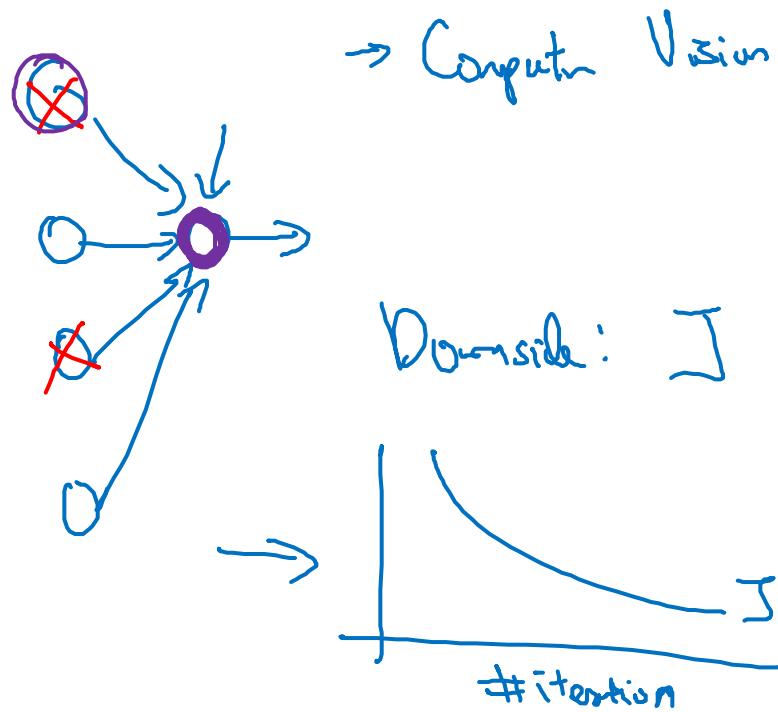
deeplearning.ai

Regularizing your
neural network

Understanding
dropout

Why does drop-out work?

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





deeplearning.ai

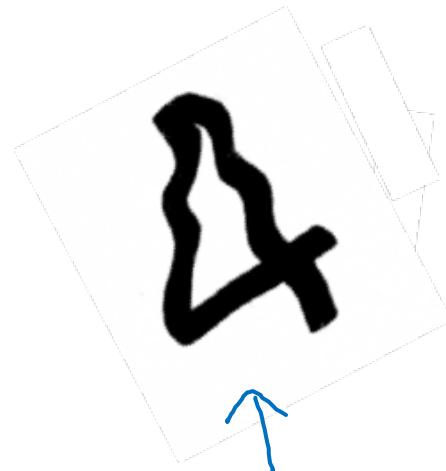
Regularizing your neural network

Other regularization methods

Data augmentation



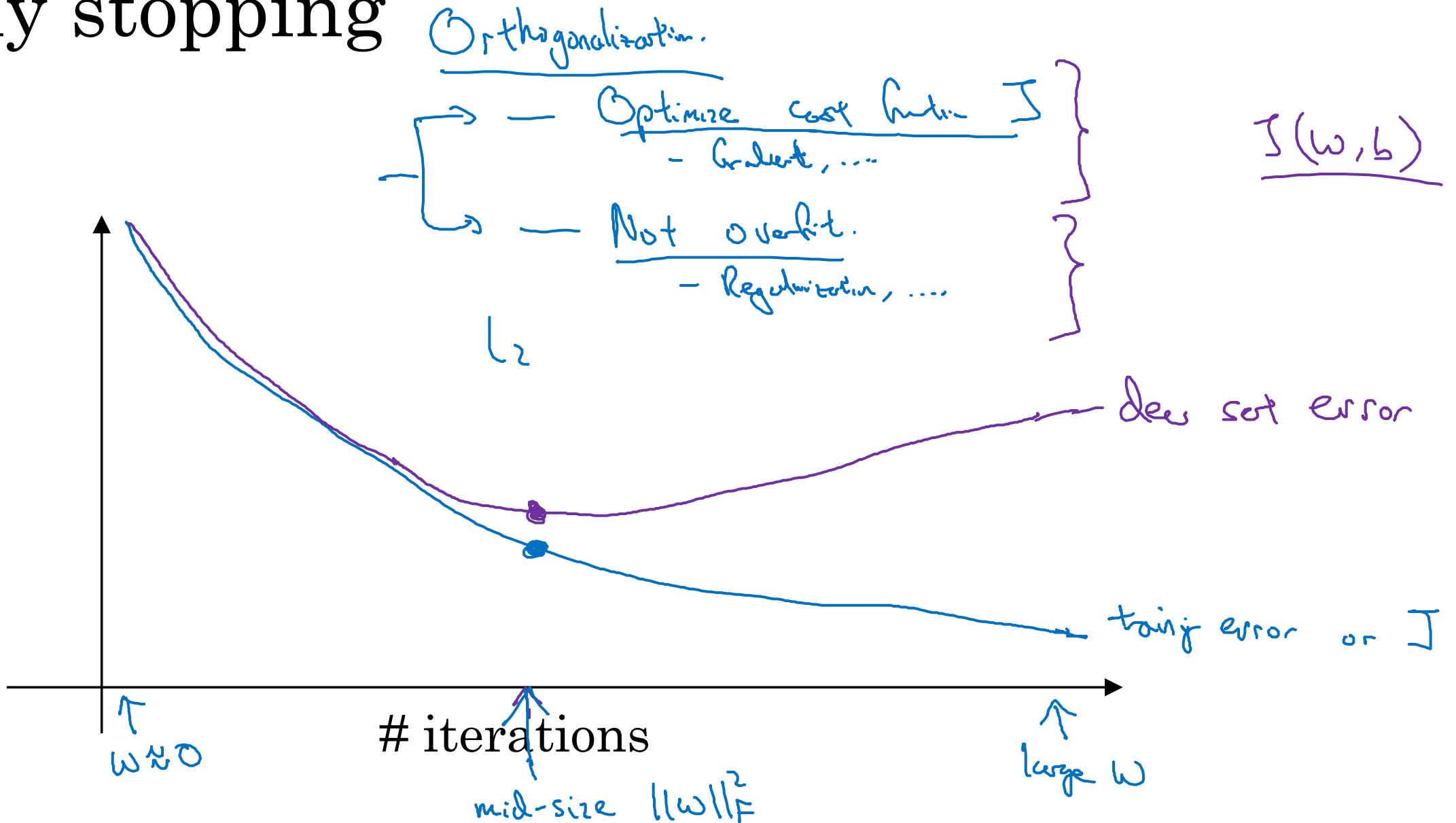
4



4



Early stopping





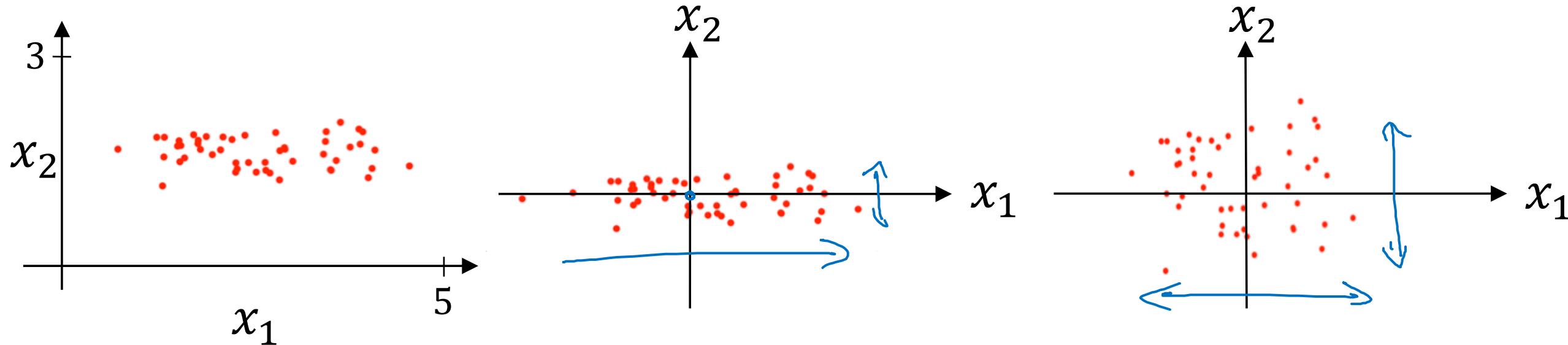
deeplearning.ai

Setting up your
optimization problem

Normalizing inputs

Normalizing training sets

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



Subtract mean:

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

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

Normalize variance

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

~ element-wise

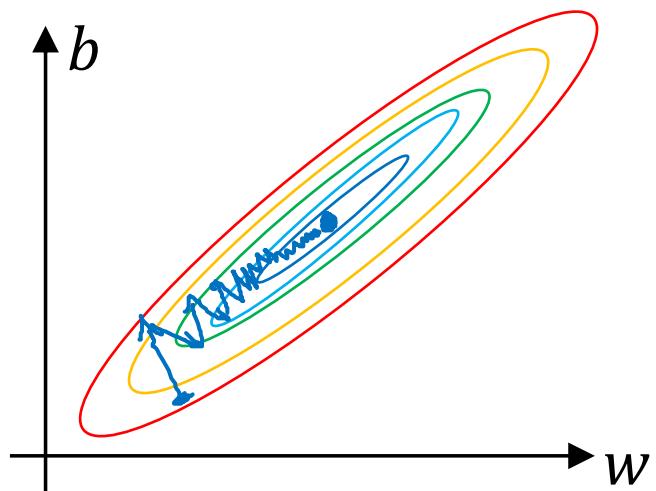
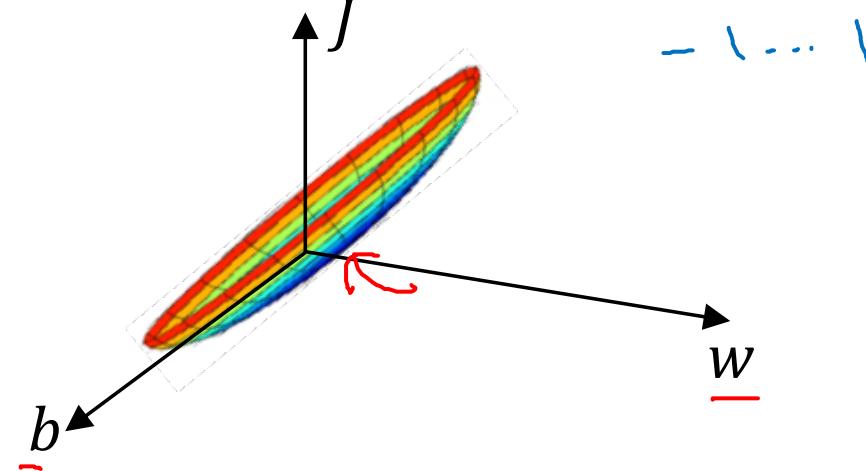
$$\underline{x / = \sigma^{-2}}$$

Use same μ, σ^2 to normalize test set.

Why normalize inputs?

$\omega_1 \quad x_1: \frac{1 \dots 1000}{0 \dots 1} \leftarrow$
 $\omega_2 \quad x_2: \frac{0 \dots 1}{-1 \dots 1} \leftarrow$

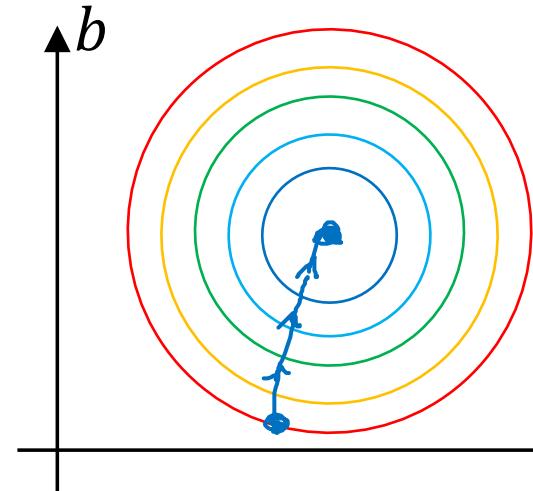
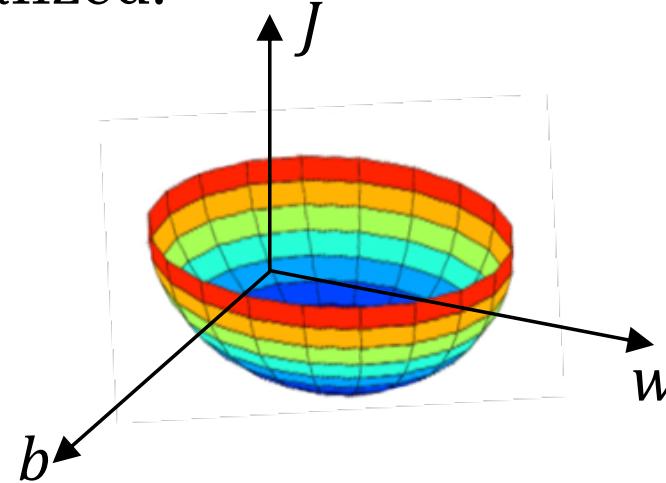
Unnormalized:



$x_1: 0 \dots 1$
 $x_2: -1 \dots 1$
 $x_3: 1 \dots 2$

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

Normalized:



w Andrew Ng



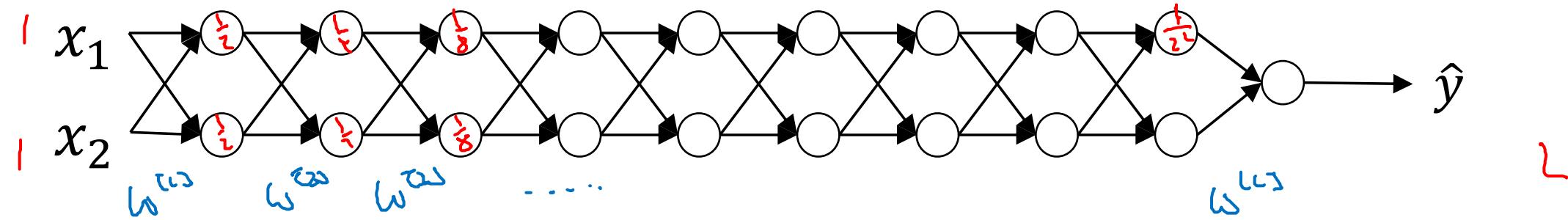
deeplearning.ai

Setting up your
optimization problem

Vanishing/exploding
gradients

Vanishing/exploding gradients

$L=150$



$$\underline{g(z) = z} . \quad b^{[L]} = 0 .$$



$$w^{[L]} > I$$

$$w^{[L-1]} < I \quad \begin{bmatrix} 0.9 & \\ & 0.9 \end{bmatrix}$$

$$w^{[L]} = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 1.5 & 0 \\ 0 & 0 & 6.5 \end{bmatrix}$$

$$1.5^{L-1} \times \\ 6.5^{L-1} \times$$

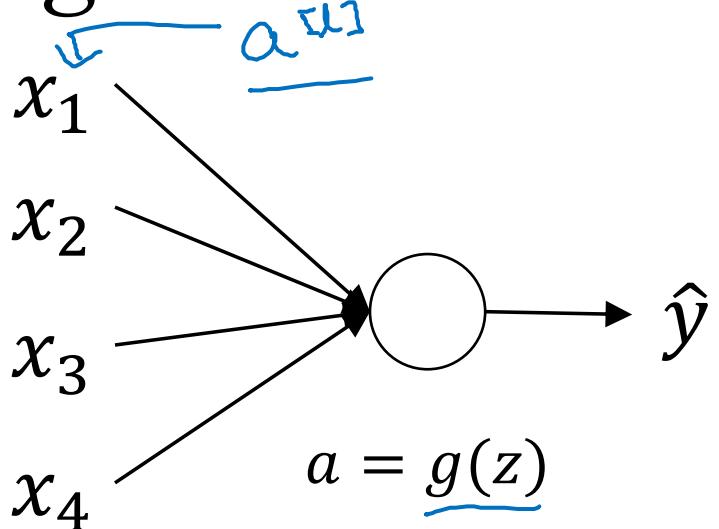
$$z^{[L]} = \underline{w^{[L]} x} \\ a^{[L]} = g(z^{[L]}) = g(w^{[L]} x)$$

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

$$\hat{y} = w^{[L]} \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 1.5 & 0 \\ 0 & 0 & 6.5 \end{bmatrix}^{L-1} x$$

$$1.5^L \\ 6.5^L$$

Single neuron example



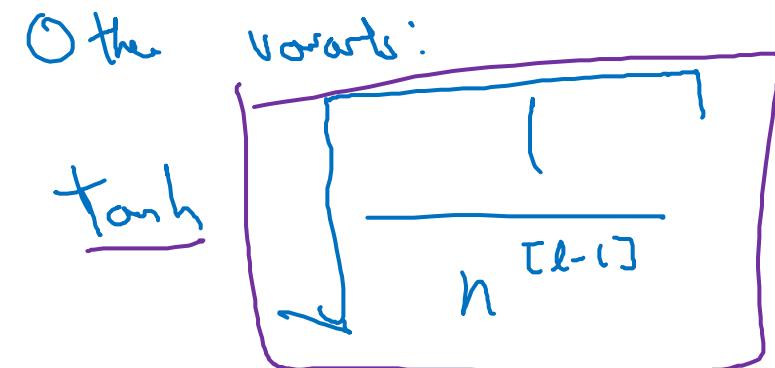
$$z = \underline{w_1}x_1 + \underline{w_2}x_2 + \cdots + \underline{w_n}x_n \quad \cancel{\text{if } n \text{ is large}}$$

\downarrow
Large $n \rightarrow$ Smaller w_i

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

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

\downarrow
 ReLU $\underline{g^{[l]}}(z) = \text{ReLU}(z)$



Xavier initialization \uparrow

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

\uparrow



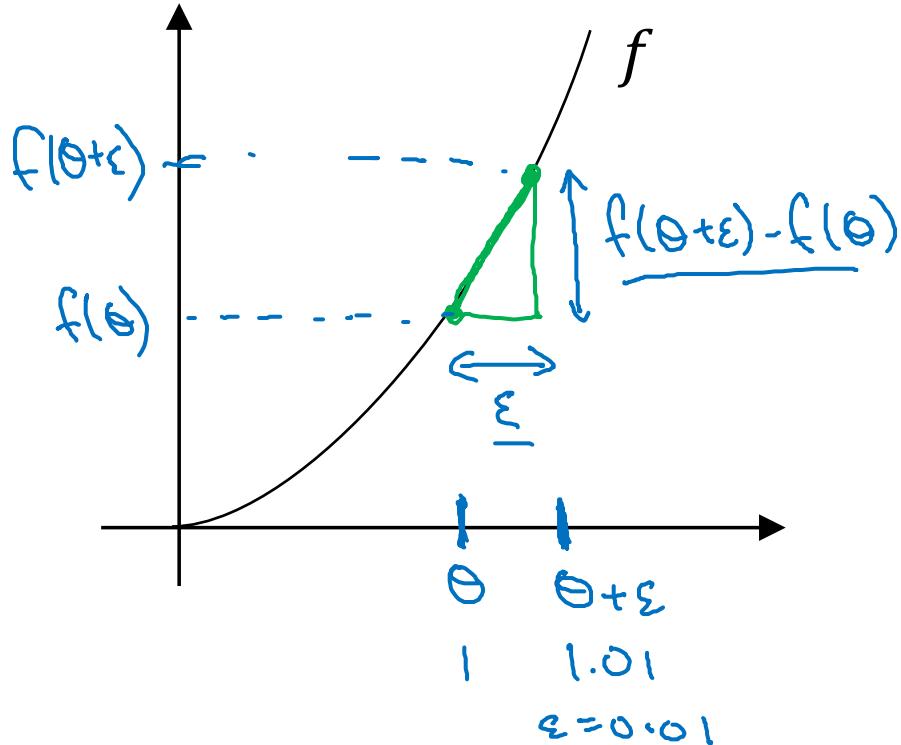
deeplearning.ai

Setting up your optimization problem

Numerical approximation of gradients

Checking your derivative computation

$$\begin{aligned} f(\theta) &= \underline{\theta^3} \\ \theta &\in \mathbb{R}. \\ \text{I} \end{aligned}$$



$$\begin{aligned} g(\theta) &= \frac{d}{d\theta} f(\theta) = f'(\theta) \\ g(\theta) &= 3\theta^2. \\ g(1) &= 3 \cdot (1)^2 = 3 \\ \text{when } \theta &= 1 \\ \frac{dw}{db} \end{aligned}$$

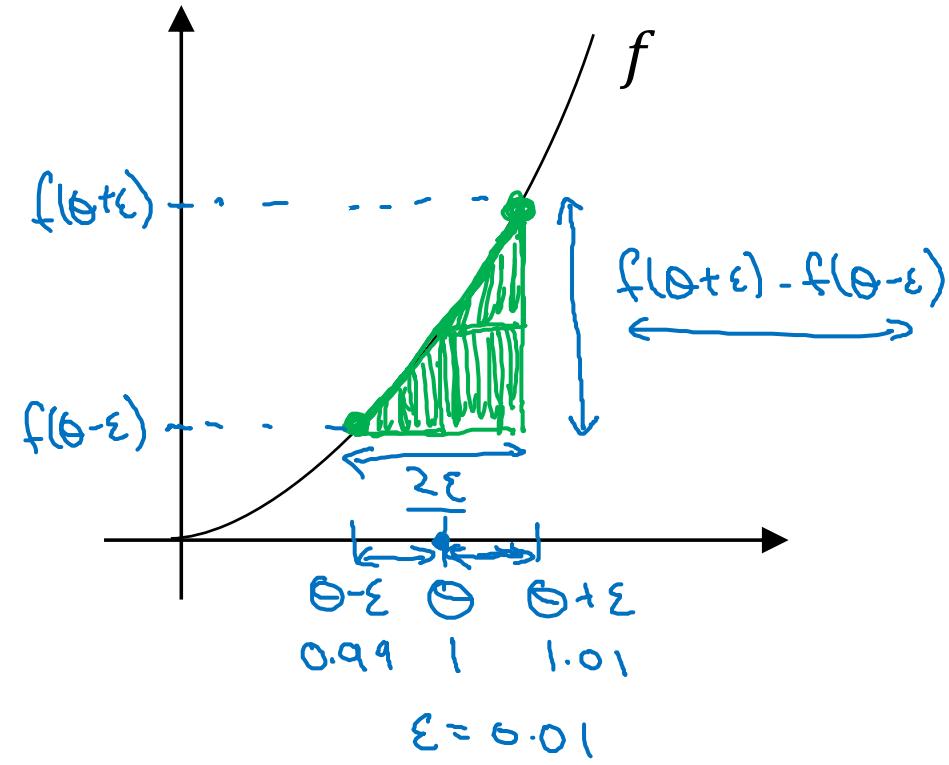
$$\begin{aligned} \frac{f(\theta + \epsilon) - f(\theta)}{\epsilon} &\approx g(\theta) \\ \frac{(1.01)^3 - 1^3}{0.01} &= \frac{3.0301}{0.01} \\ &\approx 3 \end{aligned}$$

$$\begin{aligned} \theta &= 1 \\ \theta + \epsilon &= 1.01 \end{aligned}$$

$$\begin{aligned} 3.1 \\ 3.2 \\ 0.0301 \end{aligned}$$

Checking your derivative computation

$$\underline{f(\theta) = \theta^3}$$



$$\left[\frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon} \right] \approx g(\theta)$$

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

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

approx error: 0.0001

(prev slide: 3.0301. error: 0.03)

$f'(\theta) = \lim_{\epsilon \rightarrow 0} \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$	$\frac{\mathcal{O}(\epsilon^2)}{\epsilon} = \frac{0.01}{0.0001}$	$\frac{f(\theta + \epsilon) - f(\theta)}{\epsilon}$ $\uparrow \quad \uparrow$ $\text{error: } \mathcal{O}(\epsilon) = 0.01$
--	--	---



deeplearning.ai

Setting up your
optimization problem

Gradient Checking

Gradient check for a neural network

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

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

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 $J(\theta)$?

Gradient checking (Grad check)

$$J(\theta) = J(\theta_0, \theta_1, \theta_2, \dots)$$

for each i :

$$\rightarrow \underline{d\theta_{\text{approx}}[i]} = \frac{J(\theta_0, \theta_1, \dots, \theta_i + \varepsilon, \dots) - J(\theta_0, \theta_1, \dots, \theta_i - \varepsilon, \dots)}{\varepsilon}$$

$$\approx \underline{d\theta[i]} = \frac{\partial J}{\partial \theta_i}$$

$$d\theta_{\text{approx}} \stackrel{?}{\approx} d\theta$$

Check

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

$$\varepsilon = 10^{-7}$$

$$\approx \boxed{10^{-7} - \text{great!}} \leftarrow 10^{-5}$$

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



deeplearning.ai

Setting up your
optimization problem

Gradient Checking
implementation notes

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.

$$\frac{\partial b^{[l]}}{\uparrow} \quad \frac{\partial w^{[l]}}{\uparrow}$$

- Remember regularization.

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

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

- Doesn't work with dropout.

$$J \quad \underline{\text{keep-prob} = 1.0}$$

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

$$\underline{w, b \text{ no}}$$

Copyright Notice

These slides are distributed under the Creative Commons License.

[DeepLearning.AI](#) makes these slides available for educational purposes. You may not use or distribute these slides for commercial purposes. You may make copies of these slides and use or distribute them for educational purposes as long as you cite [DeepLearning.AI](#) as the source of the slides.

For the rest of the details of the license, see <https://creativecommons.org/licenses/by-sa/2.0/legalcode>



deeplearning.ai

Optimization Algorithms

Mini-batch gradient descent

Batch vs. mini-batch gradient descent

X, Y

$X^{\{t\}}, Y^{\{t\}}$

Vectorization allows you to efficiently compute on m examples.

$$X = [x^{(1)} \ x^{(2)} \ x^{(3)} \ \dots \ x^{(500)} \ | \ x^{(1001)} \ \dots \ x^{(2000)} \ | \ \dots \ | \ \dots \ x^{(m)}]$$

$\underbrace{x^{(1)} \ \dots \ x^{(1000)}}_{(n_x, m)}$ $\underbrace{x^{(2)} \ \dots \ x^{(1000)}}_{(n_x, 1000)}$ \dots $\underbrace{x^{(5,000)} \ \dots \ x^{(m)}}_{(n_x, 1000)}$

$$Y = [y^{(1)} \ y^{(2)} \ y^{(3)} \ \dots \ y^{(1000)} \ | \ y^{(1001)} \ \dots \ y^{(2000)} \ | \ \dots \ | \ \dots \ y^{(m)}]$$

$\underbrace{y^{(1)} \ \dots \ y^{(1000)}}_{(1, m)}$ $\underbrace{y^{(2)} \ \dots \ y^{(1000)}}_{(1, 1000)}$ \dots $\underbrace{y^{(5,000)} \ \dots \ y^{(m)}}_{(1, 1000)}$

What if $m = 5,000,000$?

5,000 mini-batches of 1,000 each

Mini-batch t : $X^{\{t\}}, Y^{\{t\}}$

$x^{(i)}$
 $z^{[l]}$
 $X^{\{t\}}, Y^{\{t\}}$

Mini-batch gradient descent

repeat {
for $t = 1, \dots, 5000$ {

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

$$Z^{(l)} = W^{(l)} X^{\{t\}} + b^{(l)}$$

$$A^{(l)} = g^{(l)}(Z^{(l)})$$

:

$$A^{(L)} = g^{(L)}(Z^{(L)})$$

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

Backprop to compute gradients wrt $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)

$$W^{(l)} := W^{(l)} - \alpha \delta W^{(l)}, \quad b^{(l)} := b^{(l)} - \alpha \delta b^{(l)}$$

3 } 3 }

"1 epoch"
└ pass through training set.

1 step of gradient descent
using $\frac{X^{\{t+1\}}}{Y^{\{t+1\}}}$
(as if $t=5000$)

X, Y



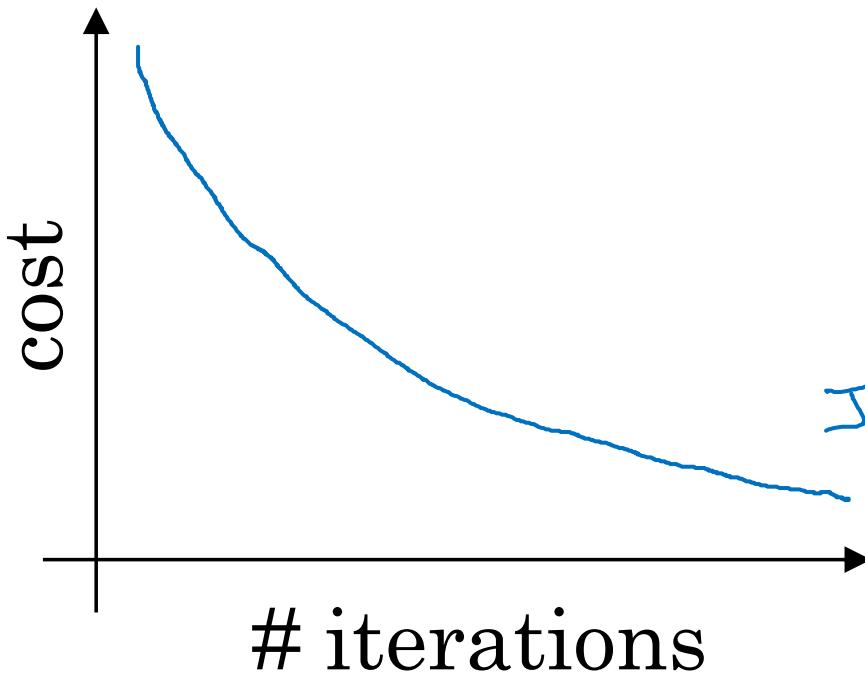
deeplearning.ai

Optimization Algorithms

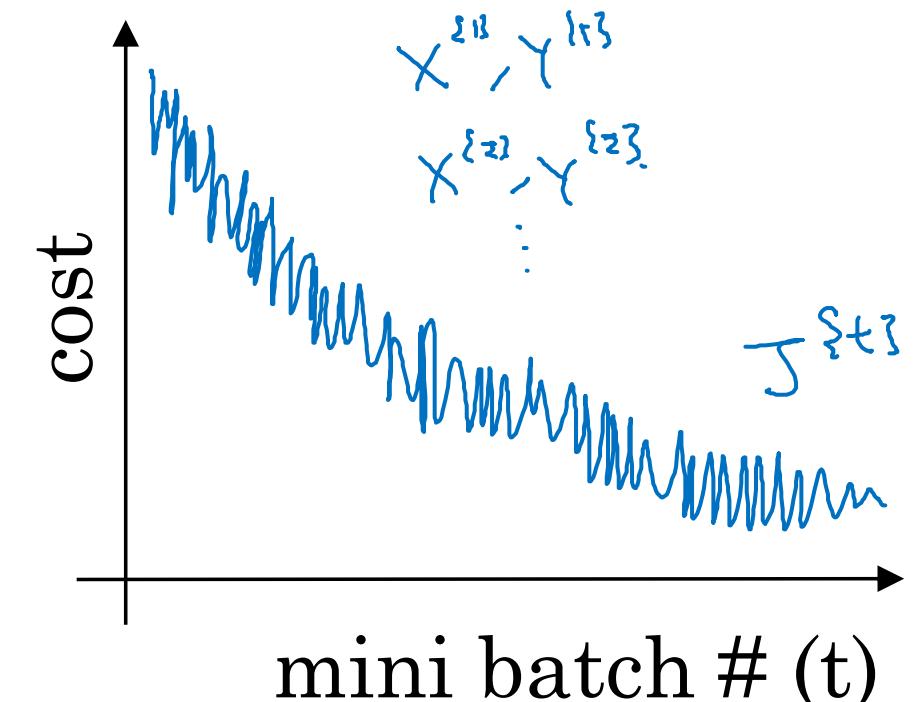
Understanding mini-batch gradient descent

Training with mini batch gradient descent

Batch gradient descent



Mini-batch gradient descent



Plot J^{st} computed using $X^{\{t\}}, Y^{\{t\}}$

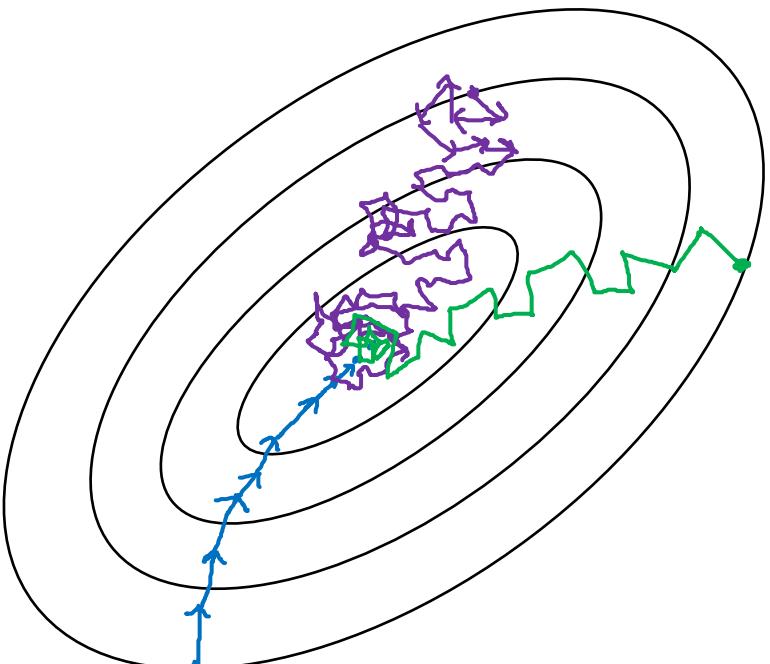
Choosing your mini-batch size

→ If mini-batch size = m : Batch gradient descent.

$$(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$$

→ If mini-batch size = 1 : Stochastic gradient descent. Every example is its own
 $(X^{(1)}, Y^{(1)}) = (x^{(1)}, y^{(1)}) \dots (x^{(n)}, y^{(n)})$ mini-batch.

In practice: Somehw in-between 1 and m



Stochastic
gradient
descent

{
Use speedup
from vectorization

In-between
(mini-batch size
not too big/small)

↓
Fastest learning.

- Vectorization.
($n \approx 1000$)
- Make passes without
processing entire training set.

Batch
gradient descent
(mini-batch size = m)

↓
Two long
per iteration

Choosing your mini-batch size

If small training set : Use batch gradient descent.
 $(m \leq 2000)$

Typical mini-batch sizes:

$$\rightarrow 64, 128, 256, 512 \quad \frac{1024}{2^{10}}$$

$2^6 \quad 2^7 \quad 2^8 \quad 2^9$



Make sure mini-batch fits in CPU/GPU memory.

$$X^{\{t\}}, Y^{\{t\}}$$



deeplearning.ai

Optimization Algorithms

Exponentially weighted averages

Temperature in London

$$\theta_1 = 40^{\circ}\text{F} \quad 4^{\circ}\text{C} \quad \leftarrow$$

$$\theta_2 = 49^{\circ}\text{F} \quad 9^{\circ}\text{C}$$

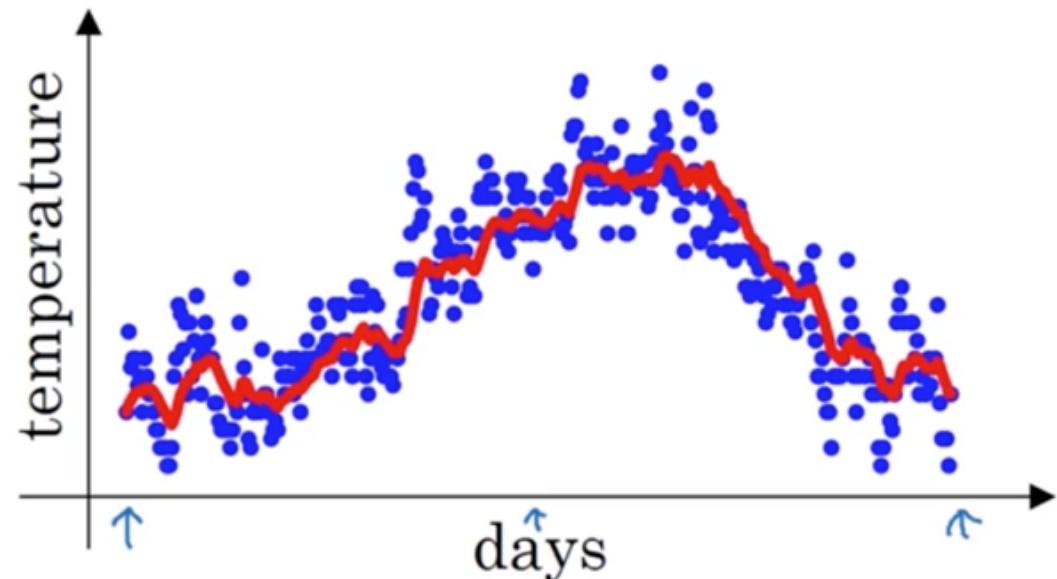
$$\theta_3 = 45^{\circ}\text{F} \quad \vdots$$

\vdots

$$\theta_{180} = 60^{\circ}\text{F} \quad 15^{\circ}\text{C}$$

$$\theta_{181} = 56^{\circ}\text{F} \quad \vdots$$

\vdots



$$V_0 = 0$$

$$V_1 = 0.9 V_0 + 0.1 \theta_1$$

$$V_2 = 0.9 V_1 + 0.1 \theta_2$$

$$V_3 = 0.9 V_2 + 0.1 \theta_3$$

\vdots

$$V_t = 0.9 V_{t-1} + 0.1 \theta_t$$

Exponentially weighted averages

$$V_t = \beta V_{t-1} + (1-\beta) \theta_t \leftarrow$$

$\beta = 0.9$: ≈ 10 days' temper.

$\beta = 0.98$: ≈ 50 days

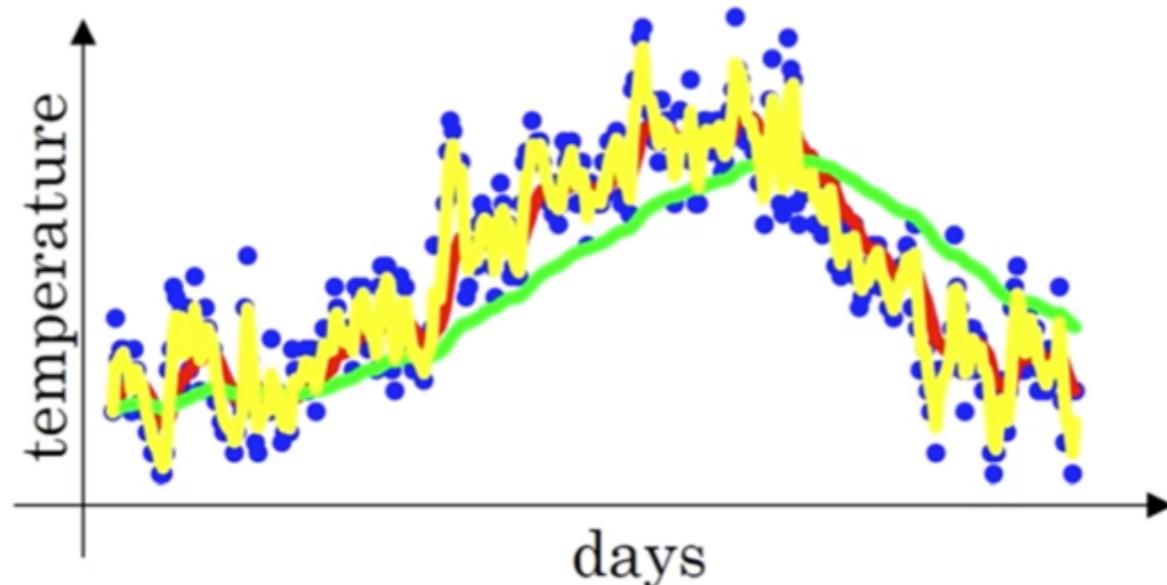
$\beta = 0.5$: ≈ 2 days

V_t is approximately

average over

$\rightarrow \approx \frac{1}{1-\beta}$ days' temperature.

$$\frac{1}{1-0.98} = 50$$





deeplearning.ai

Optimization Algorithms

Understanding exponentially weighted averages

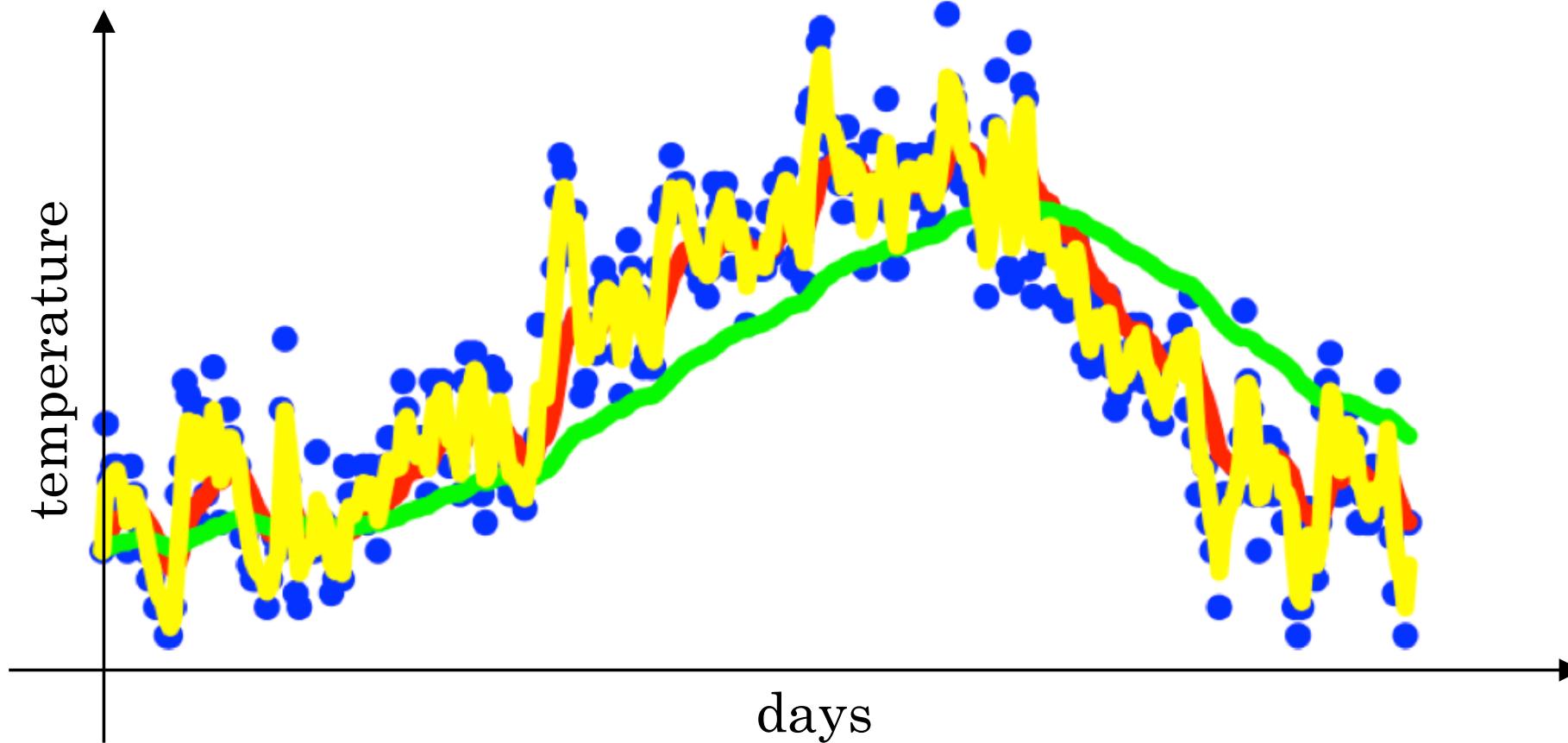
Exponentially weighted averages

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

$$\beta = 0.9$$

$$0.98$$

$$0.5$$



Exponentially weighted averages

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

$$v_{100} = 0.9v_{99} + 0.1\theta_{100}$$

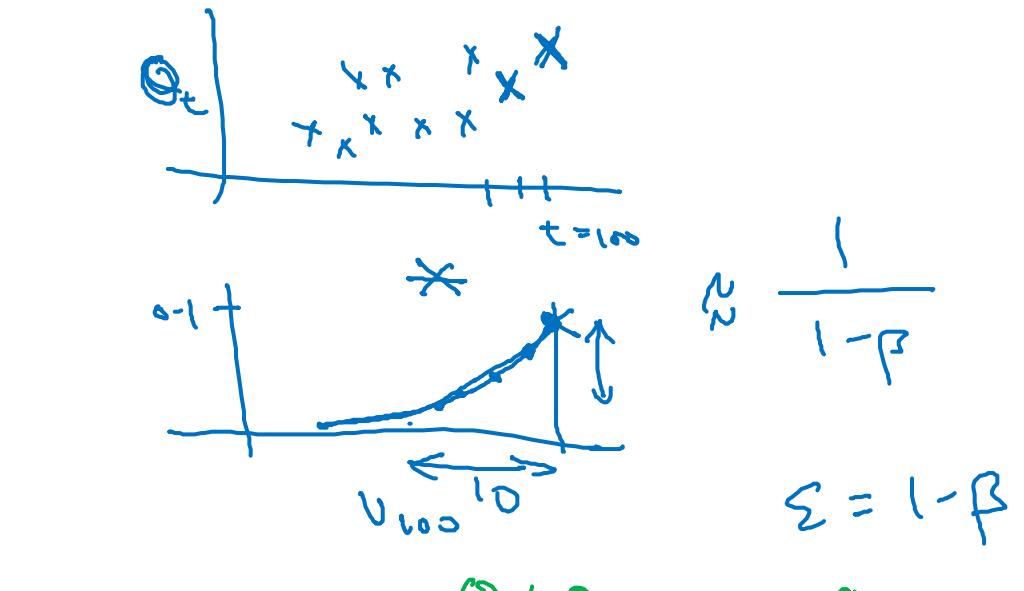
$$v_{99} = 0.9v_{98} + 0.1\theta_{99}$$

$$v_{98} = 0.9v_{97} + 0.1\theta_{98}$$

...

$$\begin{aligned} \underline{v_{100}} &= 0.1 \underline{\theta_{100}} + 0.9 \cancel{(0.1 \underline{\theta_{99}})} + 0.9 \cancel{(0.1 \underline{\theta_{98}})} + 0.9 \cancel{(0.1 \underline{\theta_{97}})} + 0.9 \cancel{(0.1 \underline{\theta_{96}})} \\ &= 0.1 \underline{\theta_{100}} + \underline{0.1 \times 0.9 \cdot \theta_{99}} + \underline{0.1 (0.9)^2 \theta_{98}} + \underline{0.1 (0.9)^3 \theta_{97}} + \underline{0.1 (0.9)^4 \theta_{96}} + \dots \end{aligned}$$

$$\underline{0.9^{10}} \approx \underline{0.35} \approx \frac{1}{e}$$



$$\frac{1}{1-\beta}$$

$$\Sigma = 1 - \beta$$

$$0.1 \underline{\theta_{98}} + 0.9 \underline{v_{97}}$$

$$\frac{(1-\epsilon)^{1/\epsilon}}{0.9} = \frac{1}{e}$$

$$0.98 ?$$

$$\epsilon = 0.02 \rightarrow \underline{0.98^{50}} \approx \frac{1}{e}$$

Implementing exponentially weighted averages

$$v_0 = 0$$

$$v_1 = \beta v_0 + (1 - \beta) \theta_1$$

$$v_2 = \beta v_1 + (1 - \beta) \theta_2$$

$$v_3 = \beta v_2 + (1 - \beta) \theta_3$$

...

$$V_0 := 0$$

$$V_0 := \beta V + (1-\beta) \theta_1$$

$$V_0 := \beta V + (1-\beta) \theta_2$$

:

$$\rightarrow V_0 = 0$$

Repeat {

Get next θ_t

$$V_0 := \beta V_0 + (1-\beta) \theta_t \leftarrow$$

}

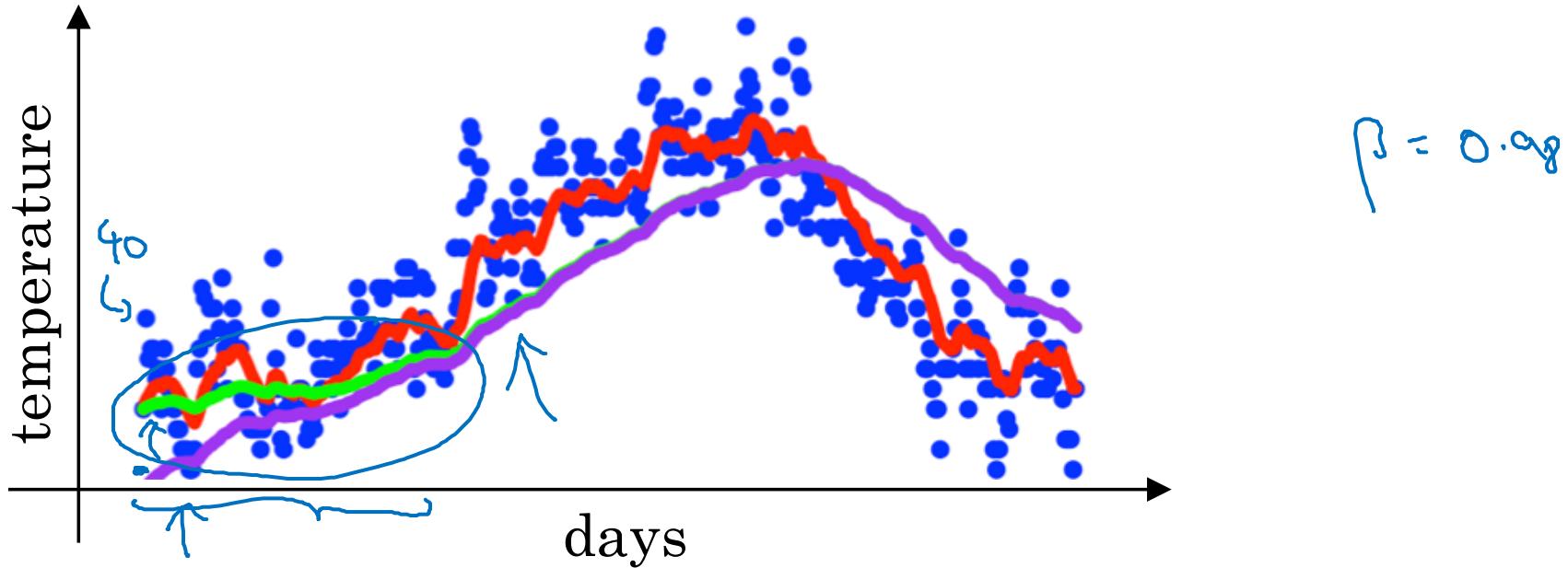


deeplearning.ai

Optimization Algorithms

Bias correction
in exponentially
weighted average

Bias correction



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

$$v_0 = 0$$

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

$$\begin{aligned} v_2 &= 0.98 v_1 + 0.02 \theta_2 \\ &= 0.98 \times 0.02 \times \theta_1 + 0.02 \theta_2 \\ &= \underline{0.0196\theta_1} + \underline{0.02\theta_2} \end{aligned}$$

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

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

$$\frac{v_2}{0.0396} =$$

$$\frac{\underline{0.0196\theta_1} + \underline{0.02\theta_2}}{0.0396}$$

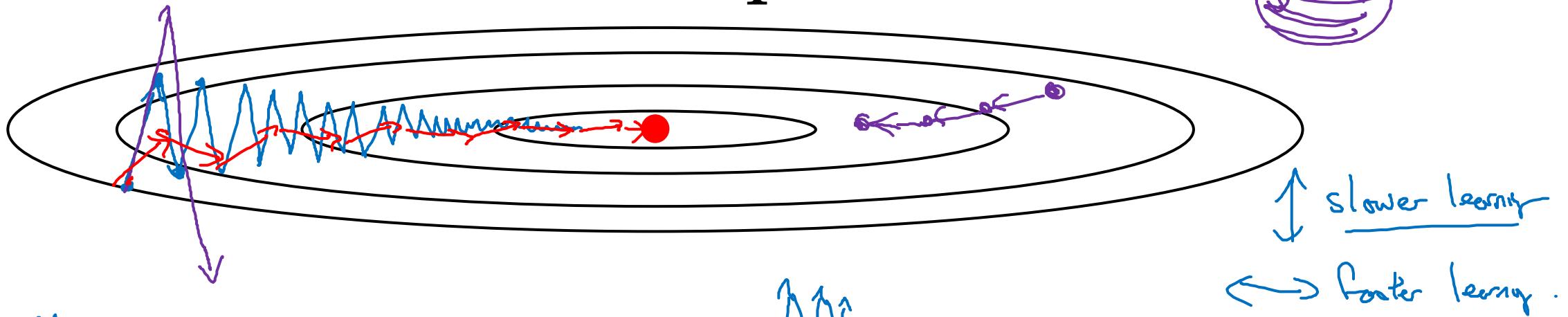


deeplearning.ai

Optimization Algorithms

Gradient descent with momentum

Gradient descent example



Momentum:

On iteration t :

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

$$v_{dw} = \beta v_{dw} + (1-\beta) \frac{\Delta w}{\text{acceleration}}$$

$$v_{db} = \beta v_{db} + (1-\beta) \frac{\Delta b}{\text{acceleration}}$$

Friction ↑ velocity

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

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

Implementation details

$$v_{dw} = 0, v_{db} = 0$$

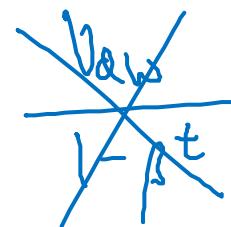
On iteration t :

Compute dW, db on the current mini-batch

$$\begin{aligned}\rightarrow v_{dw} &= \beta v_{dw} + (1 - \beta) dW \\ \rightarrow v_{db} &= \beta v_{db} + (1 - \beta) db\end{aligned}$$

$v_{dw} = \beta v_{dw} + dW \leftarrow$

$$W = W - \underbrace{\alpha v_{dw}}, b = \underbrace{b - \alpha v_{db}}$$



Hyperparameters: α, β

$$\beta = 0.9$$

average over last ≈ 10 gradients

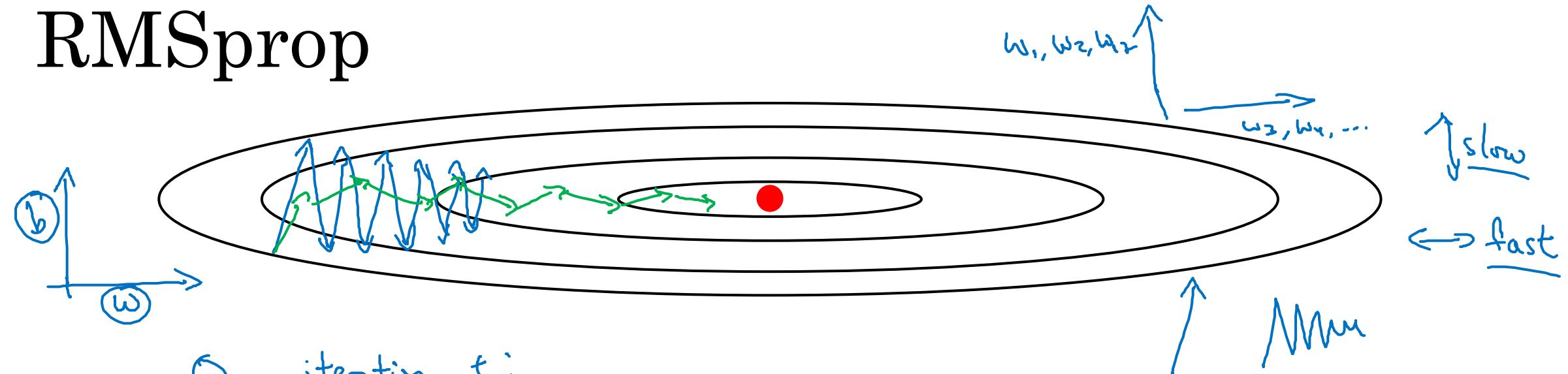


deeplearning.ai

Optimization Algorithms

RMSprop

RMSprop



On iteration t :

Compute dW, db on current mini-batch

$$\underline{S_{dw}} = \beta_2 \underline{S_{dw}} + (1-\beta_2) \underline{dW^2} \leftarrow \text{element-wise}$$

$$\rightarrow \underline{S_{db}} = \beta_2 \underline{S_{db}} + (1-\beta_2) \underline{d b^2} \leftarrow \text{large}$$

$$w := w - \frac{\alpha dW}{\sqrt{\underline{S_{dw}} + \epsilon}} \leftarrow$$

$$b := b - \frac{\alpha db}{\sqrt{\underline{S_{db}} + \epsilon}} \leftarrow$$

$$\epsilon = 10^{-8}$$



deeplearning.ai

Optimization Algorithms

Adam optimization algorithm

Adam optimization algorithm

$$V_{dw} = 0, S_{dw} = 0. \quad V_{db} = 0, S_{db} = 0$$

On iteration t :

Compute $\delta w, \delta b$ using current mini-batch

$$V_{dw} = \beta_1 V_{dw} + (1 - \beta_1) \delta w, \quad V_{db} = \beta_1 V_{db} + (1 - \beta_1) \delta b \quad \leftarrow \text{"moment"} \beta_1$$

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

$yhat = np.array([.9, 0.2, 0.1, .4, .9])$

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

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

$$w := w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected} + \epsilon}}$$

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

Hyperparameters choice:

- α : needs to be tune
- β_1 : 0.9 $\rightarrow (\underline{dw})$
- β_2 : 0.999 $\rightarrow (\underline{dw^2})$
- ϵ : 10^{-8}

Adam: Adaptive moment estimation



Adam Coates



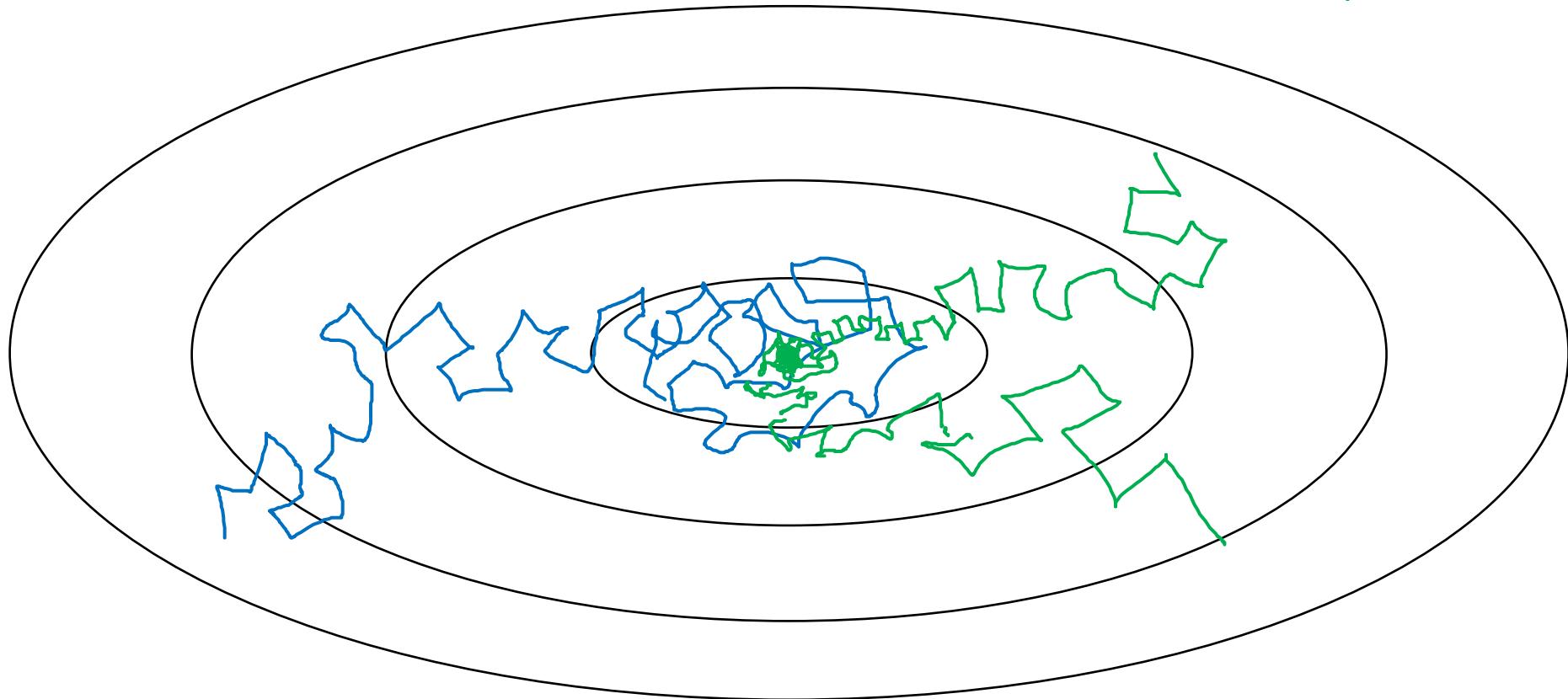
deeplearning.ai

Optimization Algorithms

Learning rate decay

Learning rate decay

Slowly reduce λ

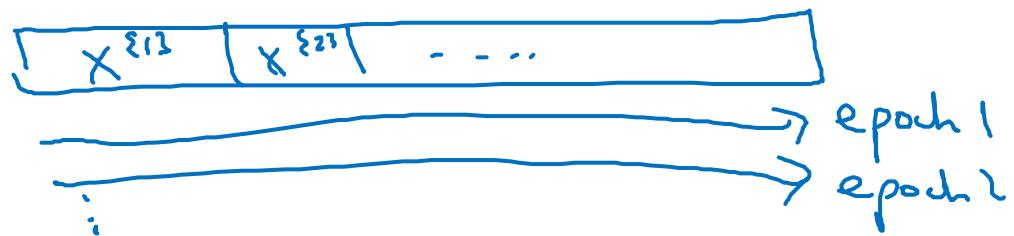


Learning rate decay

1 epoch = 1 pass through data.

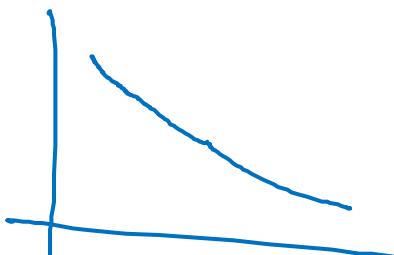
$$\alpha = \frac{\alpha_0}{1 + \text{decay-rate} * \text{epoch-num}}$$

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
:	:

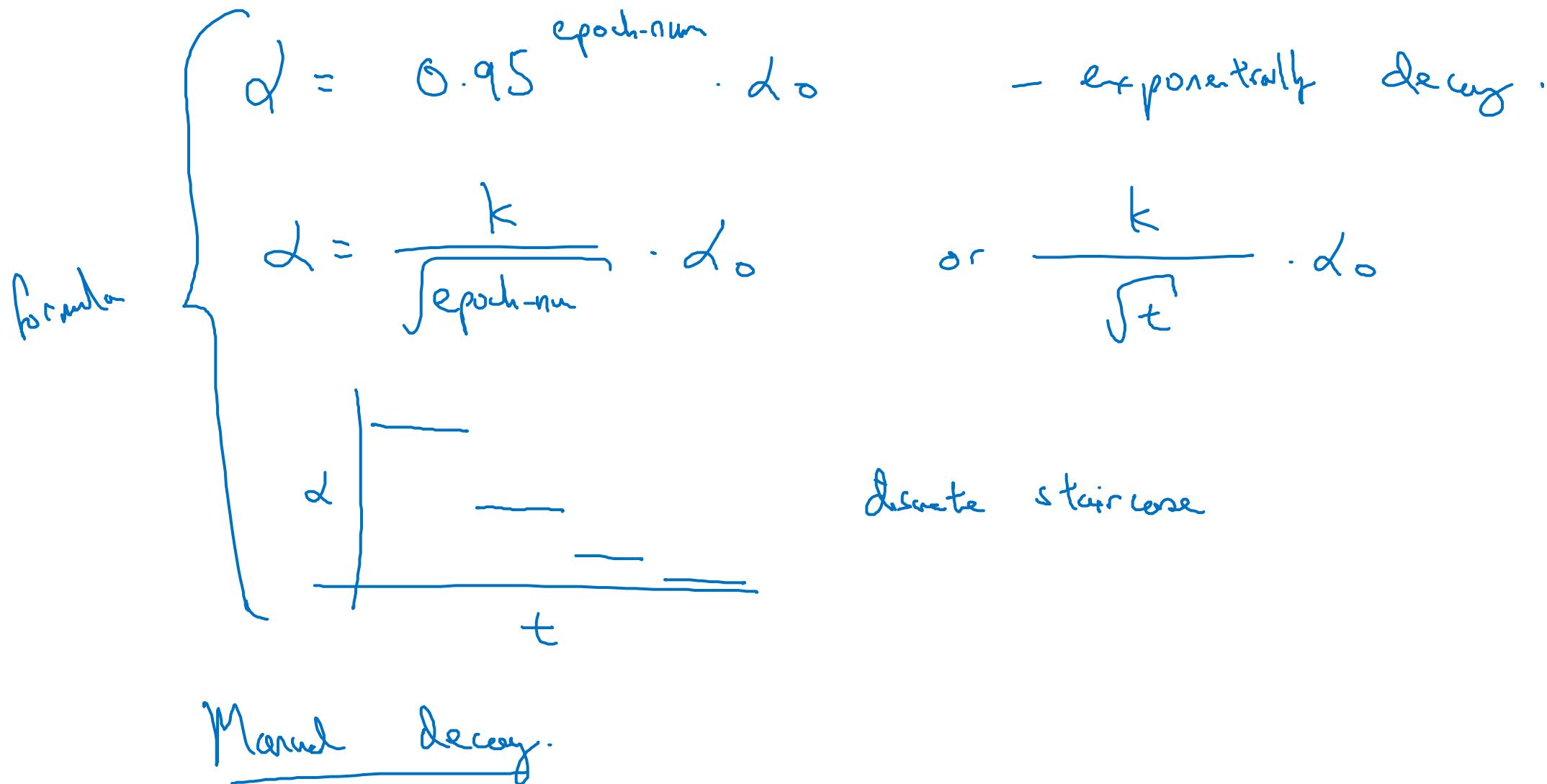


$$\alpha_0 = 0.2$$

$$\text{decay-rate} = 1$$



Other learning rate decay methods



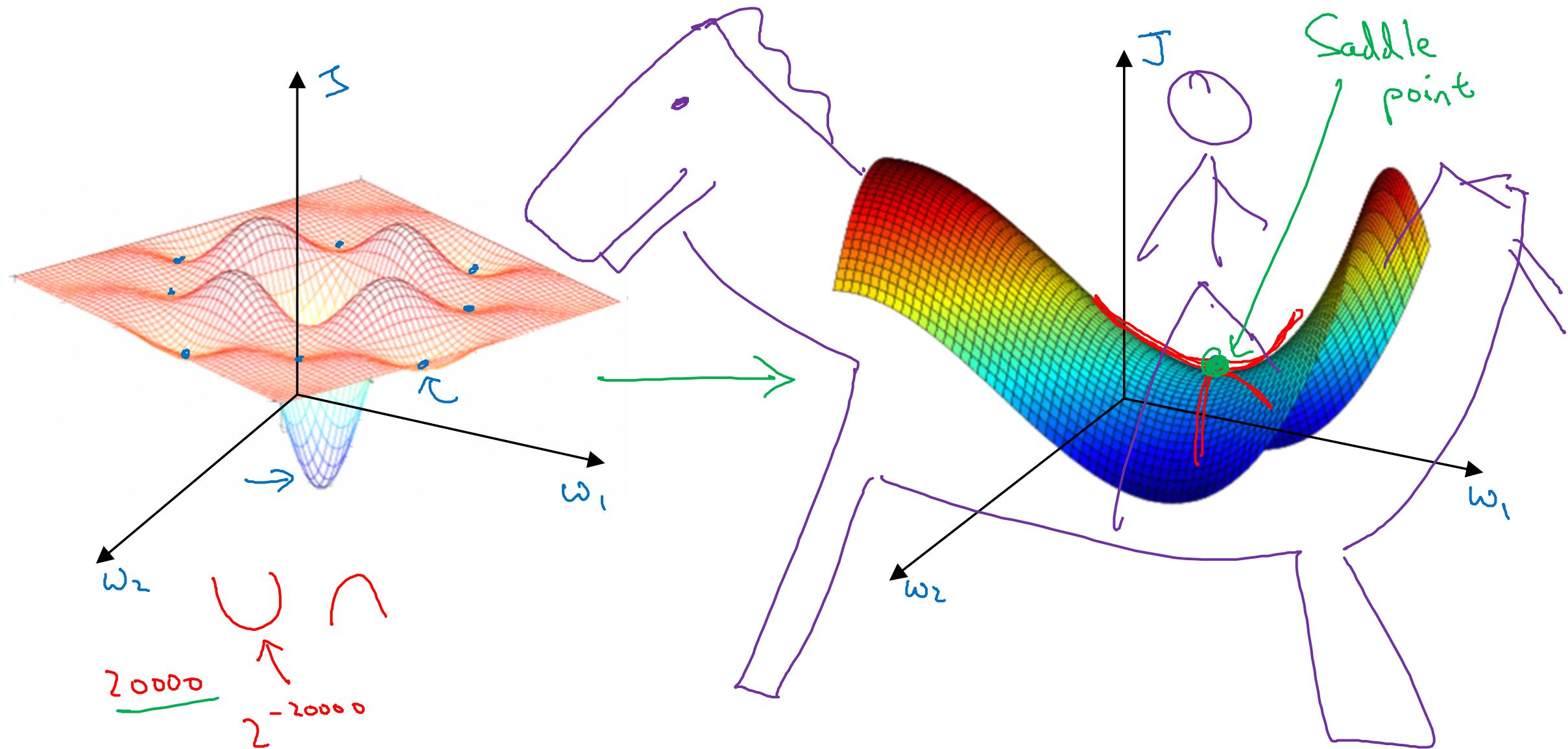


deeplearning.ai

Optimization Algorithms

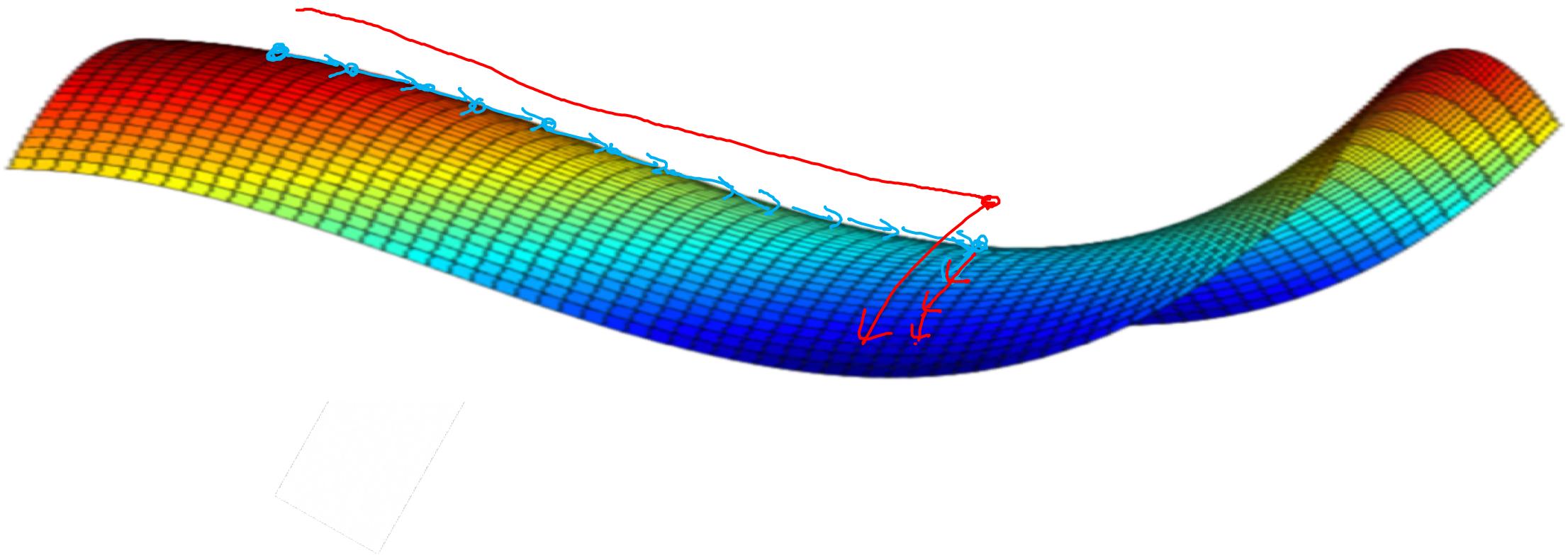
The problem of local optima

Local optima in neural networks



Andrew Ng

Problem of plateaus



- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow

Copyright Notice

These slides are distributed under the Creative Commons License.

[DeepLearning.AI](#) makes these slides available for educational purposes. You may not use or distribute these slides for commercial purposes. You may make copies of these slides and use or distribute them for educational purposes as long as you cite [DeepLearning.AI](#) as the source of the slides.

For the rest of the details of the license, see <https://creativecommons.org/licenses/by-sa/2.0/legalcode>

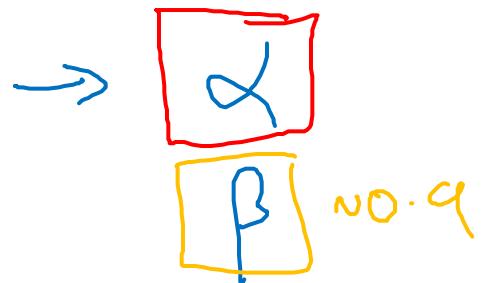


deeplearning.ai

Hyperparameter tuning

Tuning process

Hyperparameters



$\beta_1, \beta_2, \epsilon$
 $0.9, 0.99, 10^{-8}$

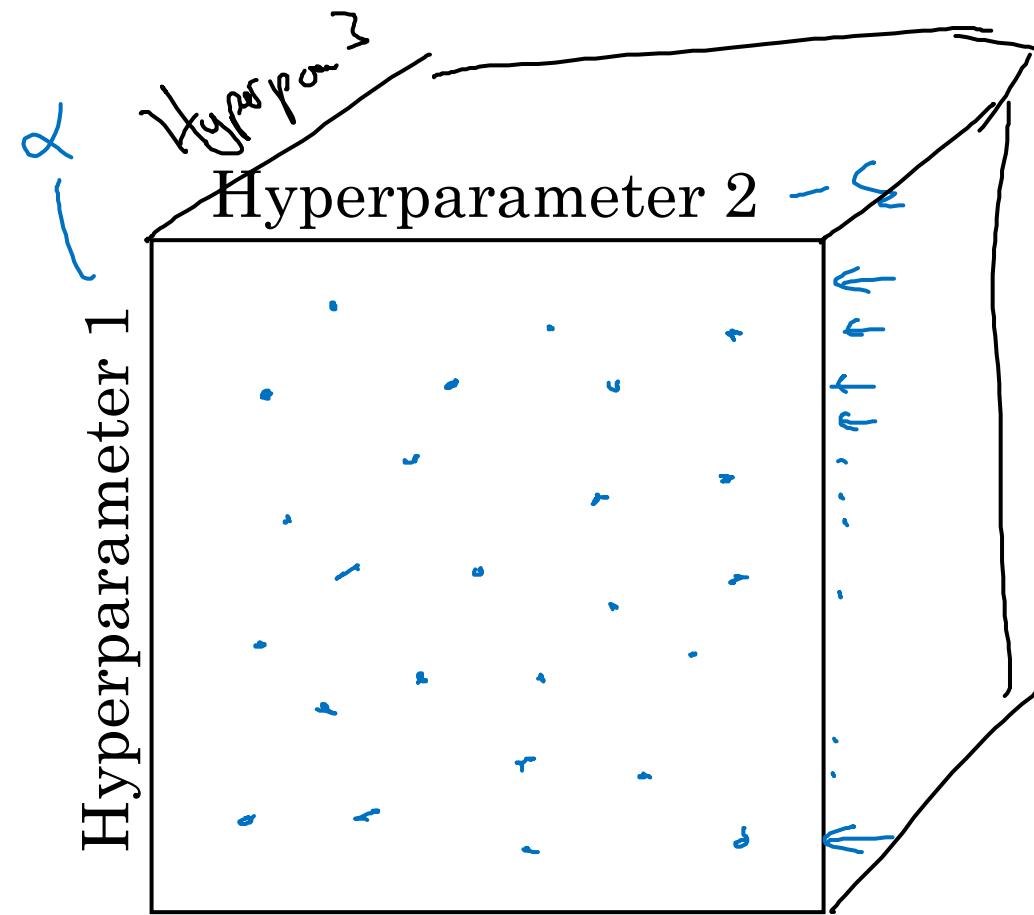
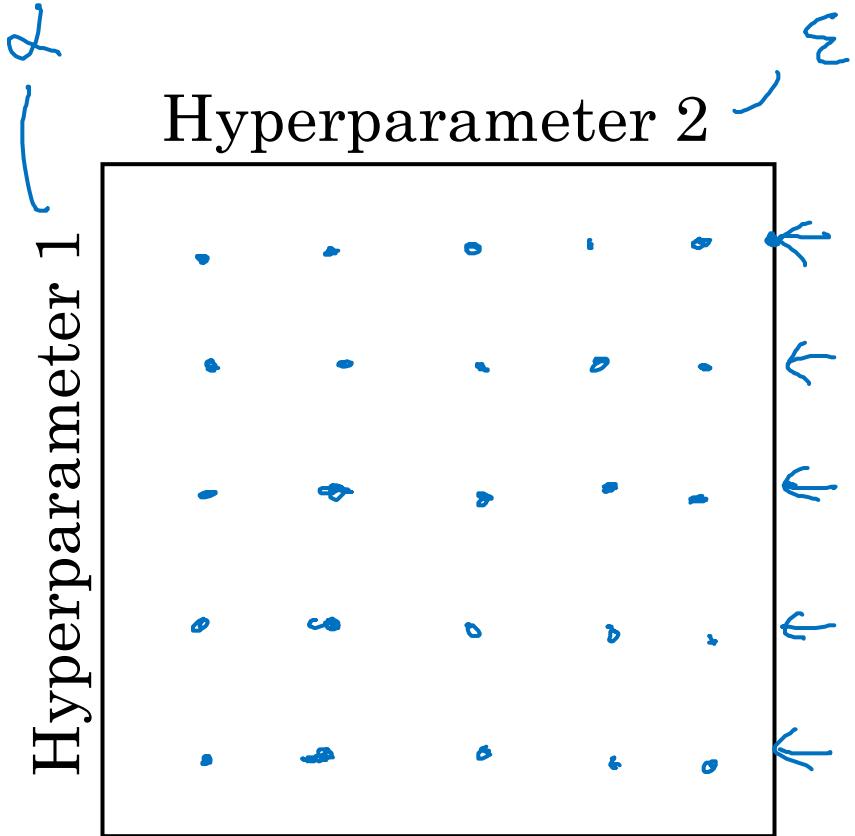
layers

hidden units

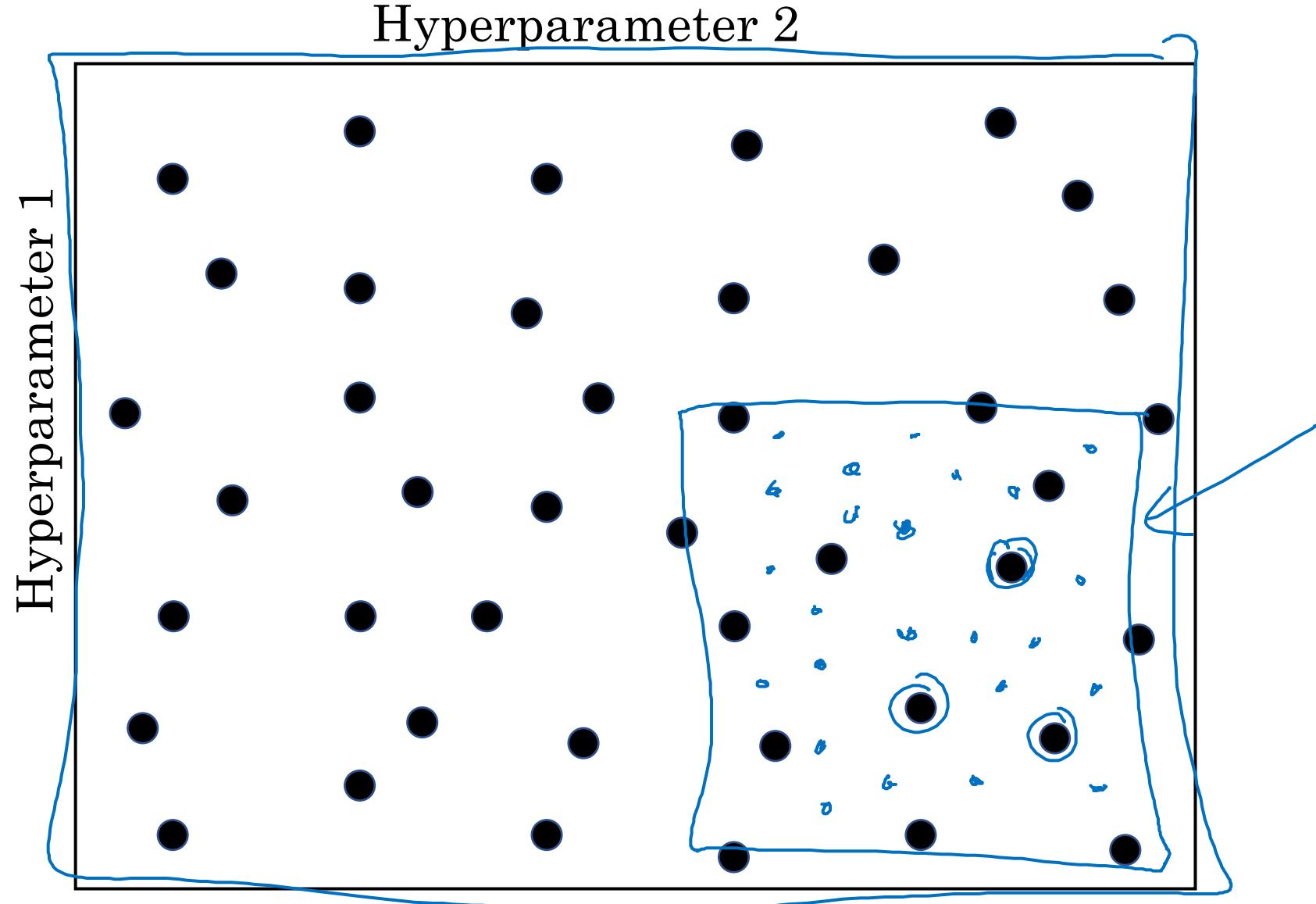
learning rate decay

mini-batch size

Try random values: Don't use a grid



Coarse to fine





deeplearning.ai

Hyperparameter tuning

Using an appropriate
scale to pick
hyperparameters

Picking hyperparameters at random

→ $n^{[l]} = 50, \dots, 100$

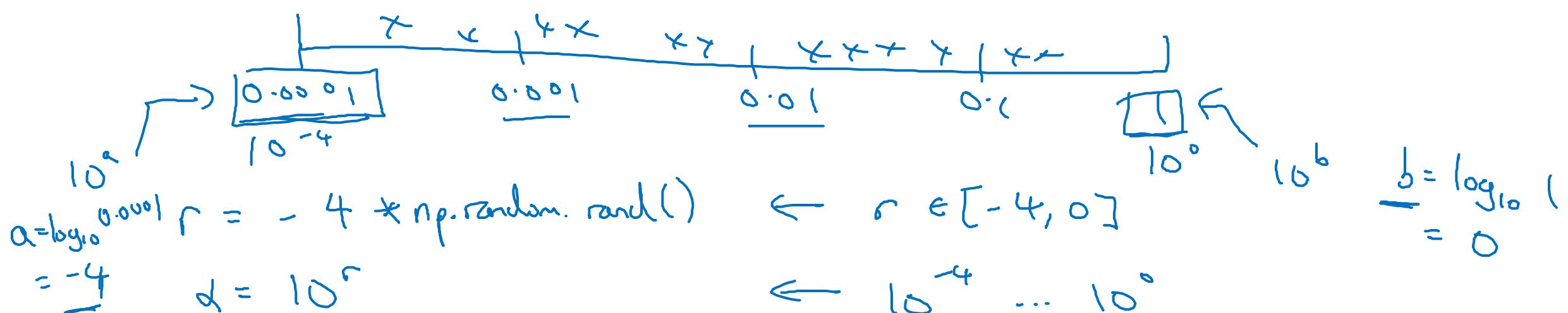
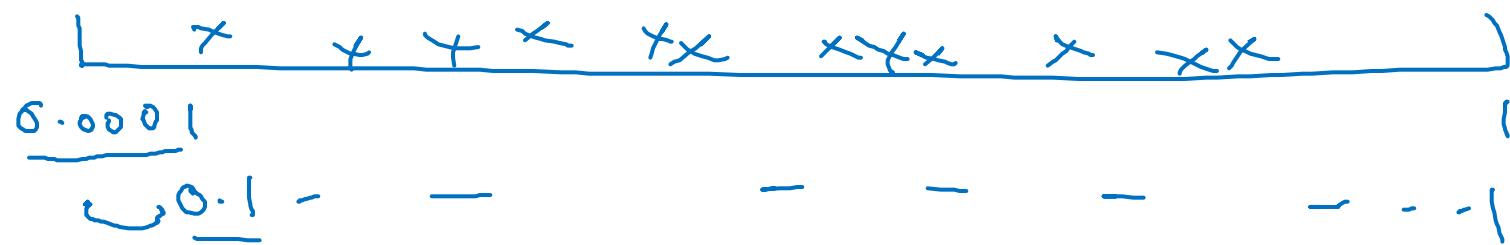


→ #layers $L : 2 - 4$

2, 3, 4

Appropriate scale for hyperparameters

$$\lambda = 0.0001, \dots, 1$$



$$10^a \dots 10^b$$

$$\frac{r \in [a, b]}{[-4, 0]}$$

$$\lambda = 10^r$$

Hyperparameters for exponentially weighted averages

$$\beta = 0.9 \dots 0.999$$

\downarrow \downarrow
 10 1000

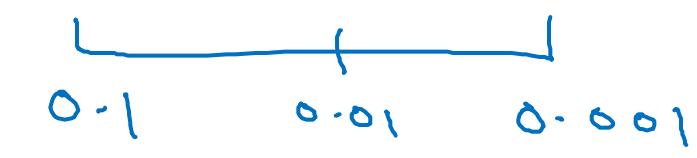
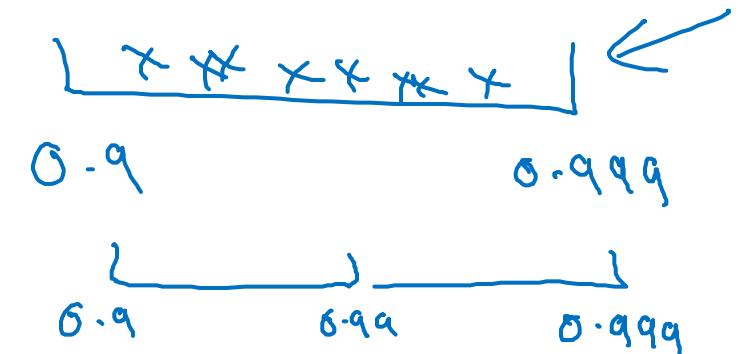
$$1-\beta = 0.1 \dots 0.001$$

$$\beta: 0.900 \rightarrow 0.9005 \quad \} \sim 10$$

$$\beta: 0.999 \rightarrow 0.9995$$

~ 1000 ~ 2000

$$\frac{1}{1-\beta}$$



$$\frac{10^{-1}}{1-\beta} \quad \frac{10^{-3}}{\beta}$$
$$r \in [-3, -1]$$

$$1-\beta = 10^r$$
$$\beta = 1 - 10^r$$

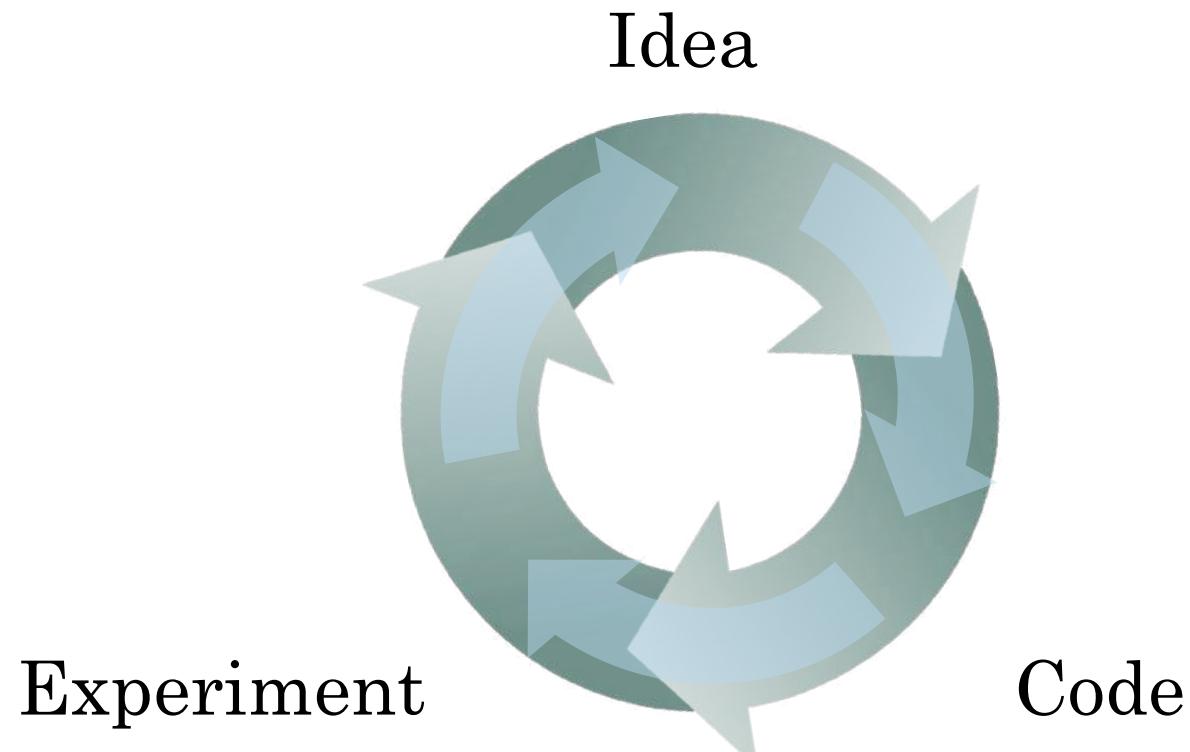


deeplearning.ai

Hyperparameters tuning

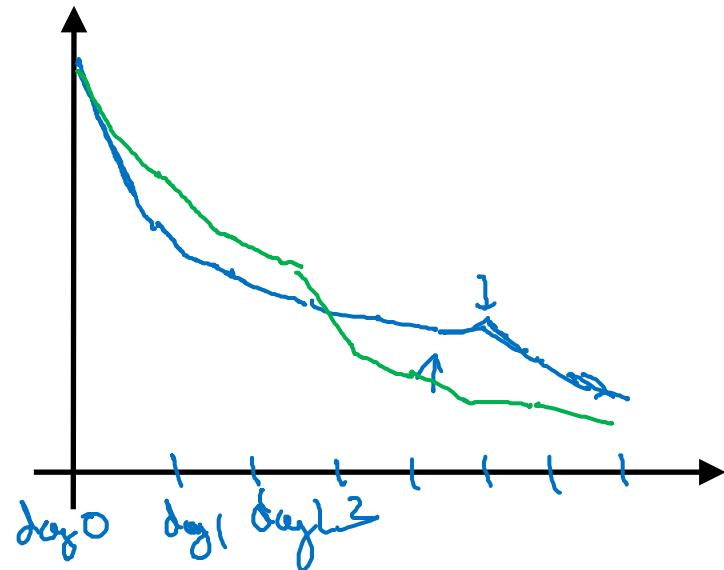
Hyperparameters tuning in practice: Pandas vs. Caviar

Re-test hyperparameters occasionally



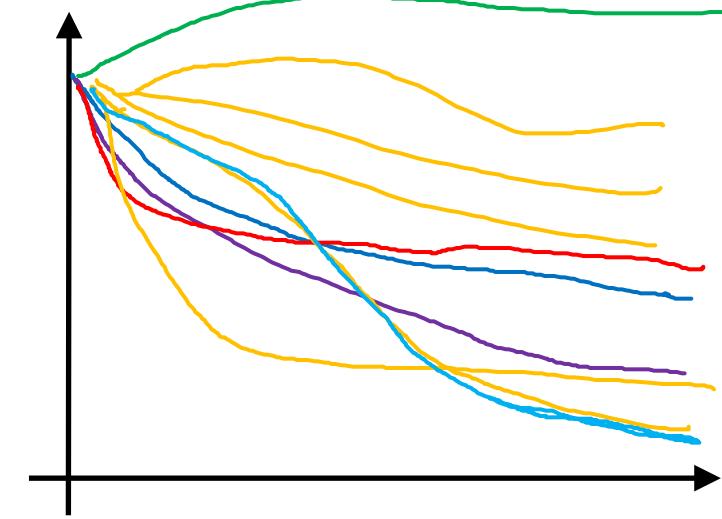
- NLP, Vision, Speech,
Ads, logistics,
- Intuitions do get stale.
Re-evaluate occasionally.

Babysitting one model



Panda ↵

Training many models in parallel



Caviar ↵

Andrew Ng

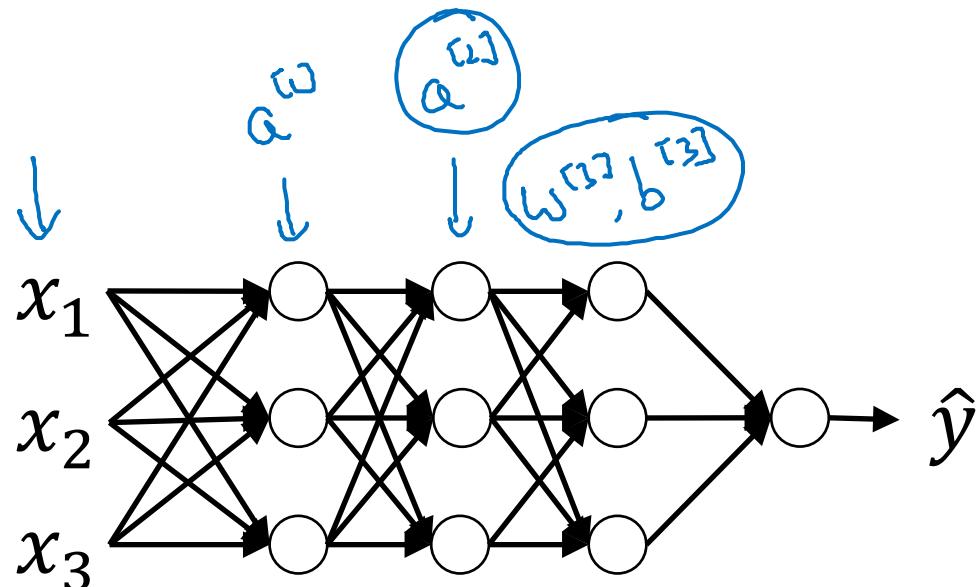
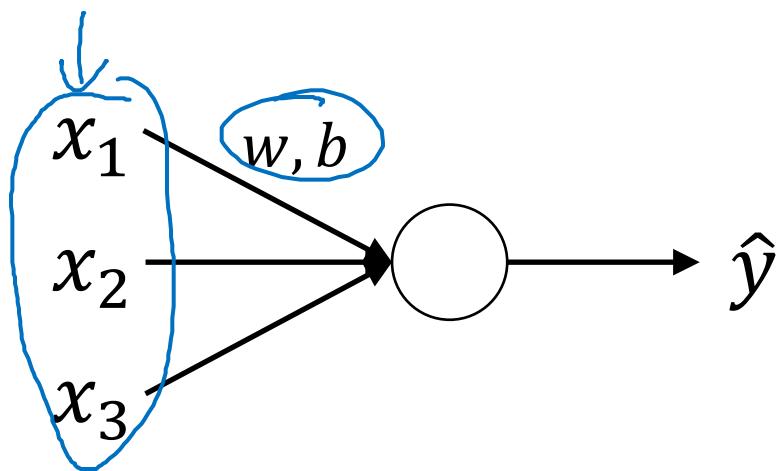


deeplearning.ai

Batch Normalization

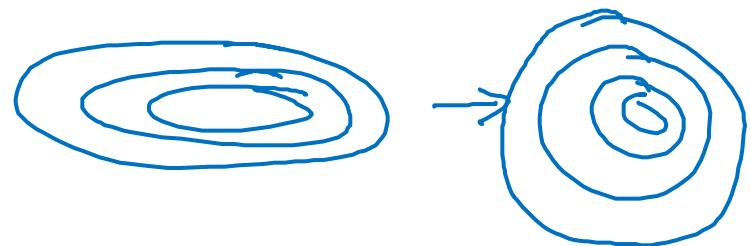
Normalizing activations in a network

Normalizing inputs to speed up learning



$$\mu = \frac{1}{m} \sum_i x^{(i)}$$
$$X = X - \mu$$
$$\sigma^2 = \frac{1}{m} \sum_i (x^{(i)})^2$$
$$X = X / \sigma^2$$

element-wise



Can we normalize $\frac{a^{[2]}}{w^{[2]}, b^{[2]}}$ so
as to train $w^{[2]}, b^{[2]}$ faster

Normalize $\frac{z^{[2]}}{\uparrow}$

Implementing Batch Norm

Given some intermediate values in NN

$$\mu = \frac{1}{m} \sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m} \sum_i (z^{(i)} - \mu)^2$$

$$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\hat{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta$$

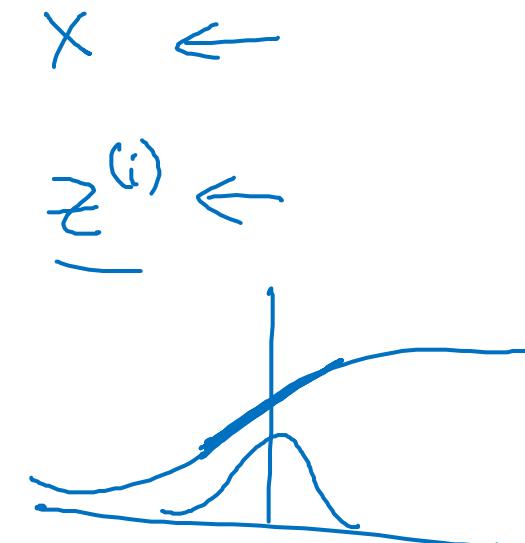
Use $\hat{z}^{[l](i)}$ instead of $z^{[l](i)}$.

If $\gamma = \sqrt{\sigma^2 + \epsilon}$ ←
then $\beta = \mu$ ←

learnable parameters
of model.

$$z^{(1)}, \dots, z^{(m)}$$

$$z^{[l](i)}$$



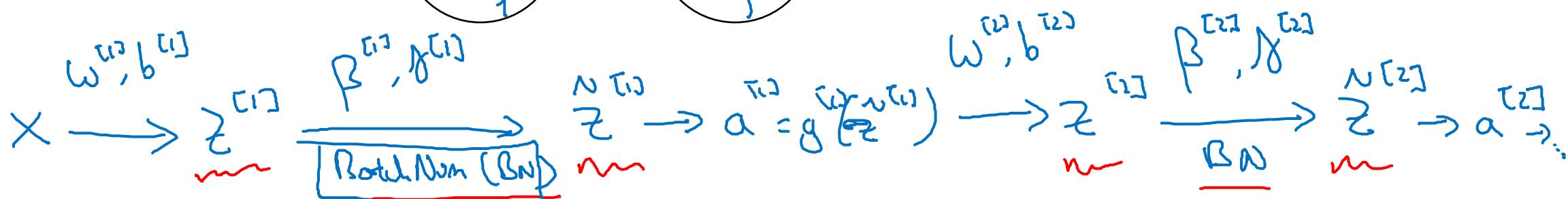
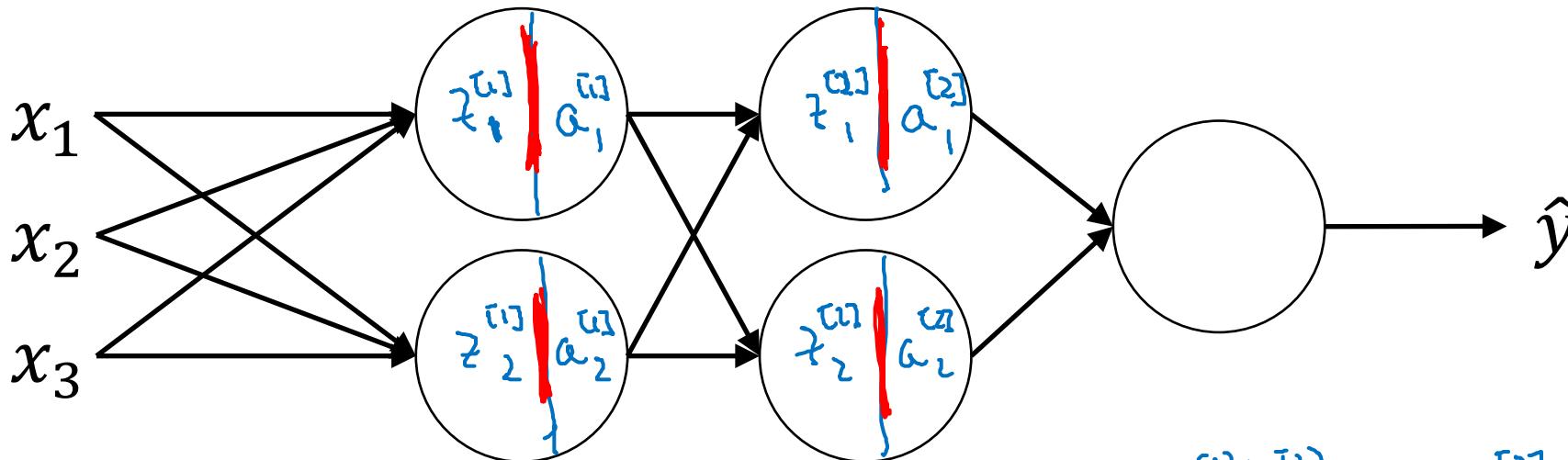


deeplearning.ai

Batch Normalization

Fitting Batch Norm into a neural network

Adding Batch Norm to a network



Parameters:

$$\left\{ w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots, w^{[L]}, b^{[L]}, \right.$$

$$\left. \rightarrow \beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[L]}, \gamma^{[L]} \right\}$$

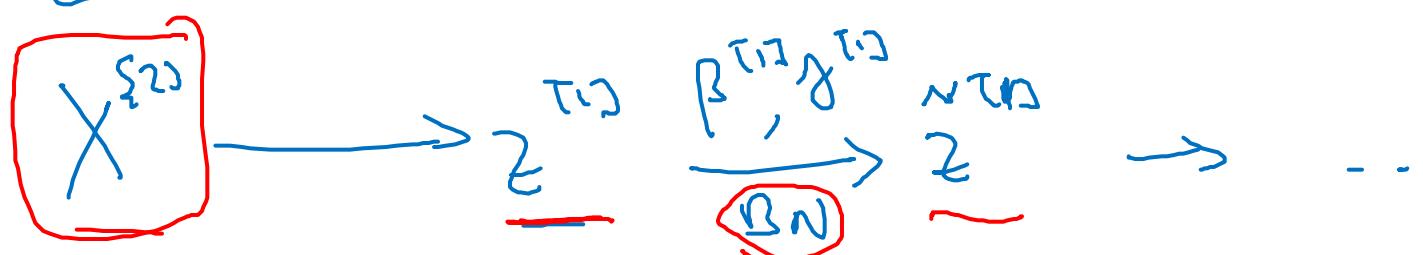
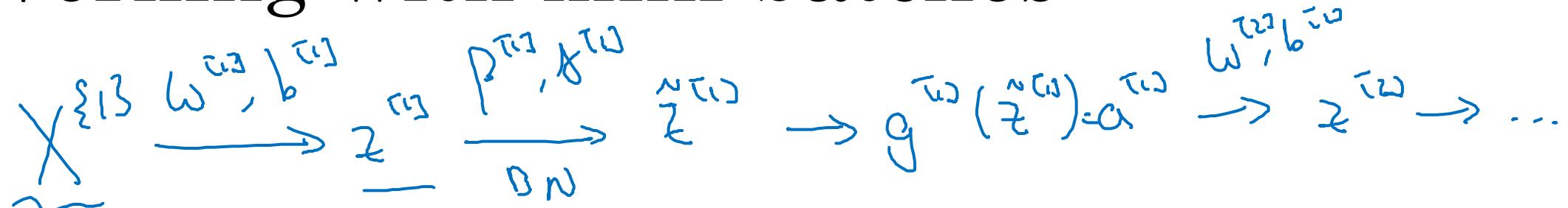
$$\rightarrow \beta$$

$$\frac{d\beta^{[l]}}{d\beta} = \frac{\tau^{[l]}}{\tau^{[l]}}$$

$$\beta = \beta - \frac{1}{n} \sum \frac{\tau^{[l]}}{\tau^{[l]}}$$

`tf.nn.batch_normalization` ←

Working with mini-batches



$X^2 \xrightarrow{\dots}$

Parameters: $\{W^{[l]}, \cancel{b^{[l]}}, \beta^{[l]}, \gamma^{[l]}\}$.

$$z^{[l]} \\ (n^{[l]}, 1)$$

$$\begin{matrix} | & | & | \\ (n^{[l]}, 1) & (n^{[l]}, 1) & (n^{[l]}, 1) \end{matrix}$$

$$\begin{aligned} \rightarrow z^{[l]} &= W^{[l]} a^{[l-1]} + \cancel{b^{[l]}} \\ z^{[l]} &= W^{[l]} a^{[l-1]} \\ z^{[l]}_{norm} &= \gamma^{[l]} z^{[l]}_{norm} + \beta^{[l]} \end{aligned}$$

Andrew Ng

Implementing gradient descent

for $t = 1 \dots \text{num MiniBatches}$
Compute forward prop on $X^{[t]}$.

In each hidden layer, use BN to replace $\underline{z}^{[l]}$ with $\hat{\underline{z}}^{[l]}$.

Use backprop to compute $\underline{dw}^{[l]}$, ~~$\underline{db}^{[l]}$~~ , $\underline{d\beta}^{[l]}$, $\underline{dg}^{[l]}$

Update parameters $\left. \begin{array}{l} w^{[l]} := w^{[l]} - \alpha \underline{dw}^{[l]} \\ \beta^{[l]} := \beta^{[l]} - \alpha \underline{d\beta}^{[l]} \\ g^{[l]} := \dots \end{array} \right\} \leftarrow$

Works w/ momentum, RMSprop, Adam.

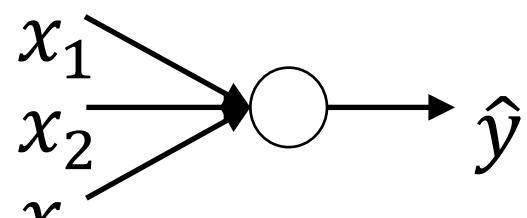


deeplearning.ai

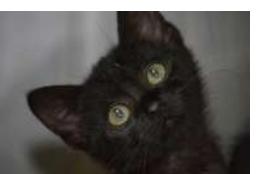
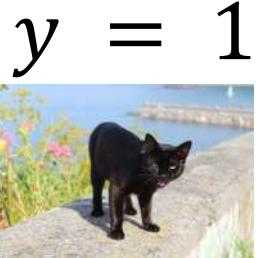
Batch Normalization

Why does
Batch Norm work?

Learning on shifting input distribution



Cat



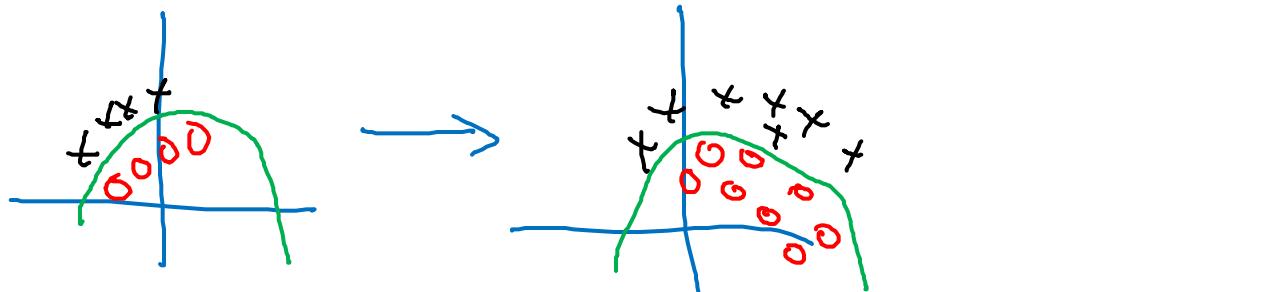
Non-Cat



$$y = 1$$

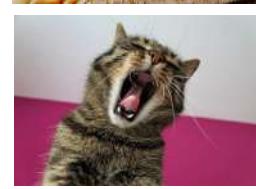


$$y = 0$$

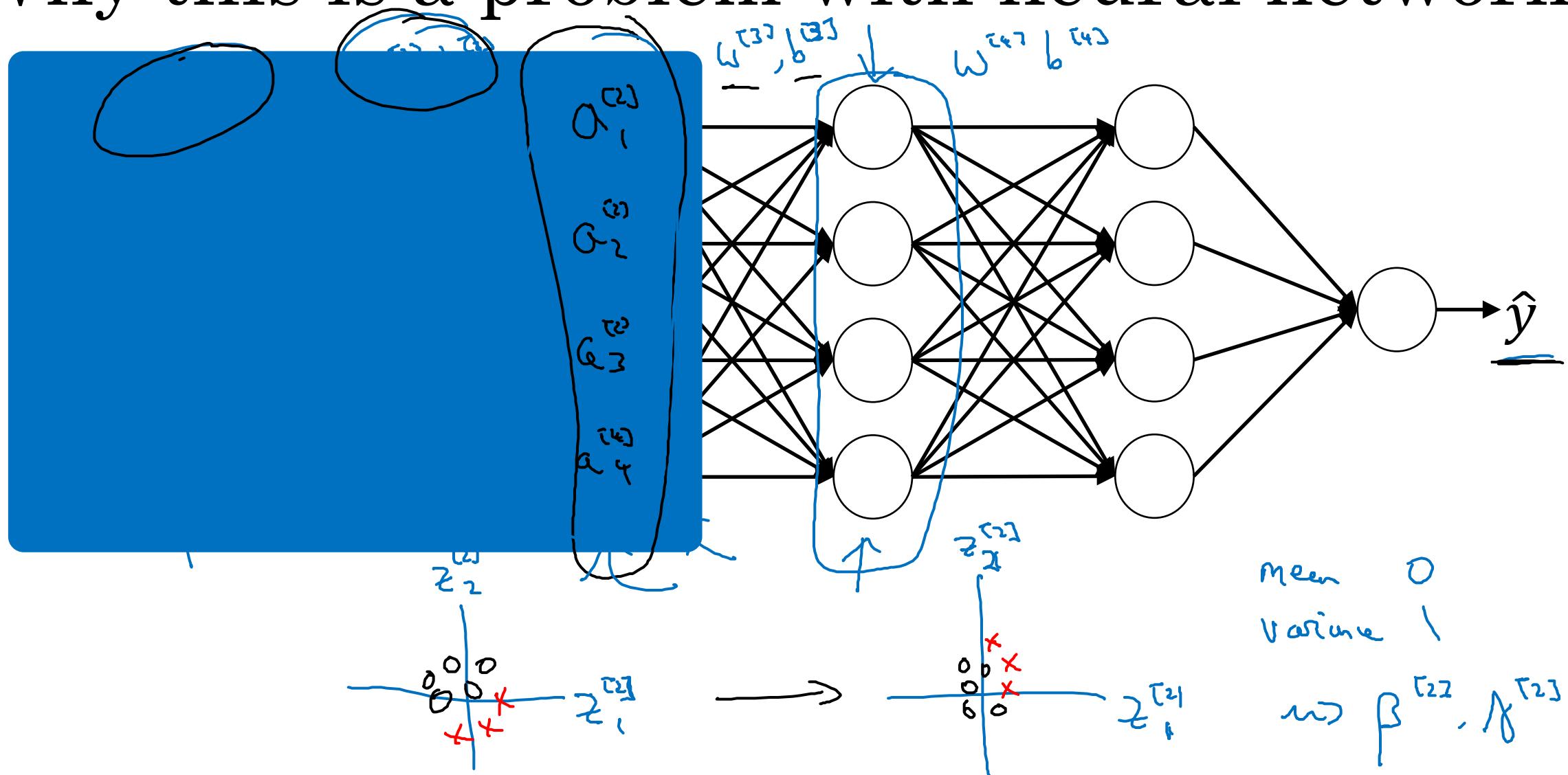


"Covariate shift"

$$\underline{x} \rightarrow y$$



Why this is a problem with neural networks?



Batch Norm as regularization

X

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
 $\xrightarrow{\hat{z}^{[l]}}$ μ, σ^2 $\{z^{[l]}\}$
 $\hat{z}^{[l]}$ $\underline{64}, \underline{128}$
- This adds some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
 μ, σ^2
- This has a slight regularization effect.

mini-batch : 64 \longrightarrow 512



deeplearning.ai

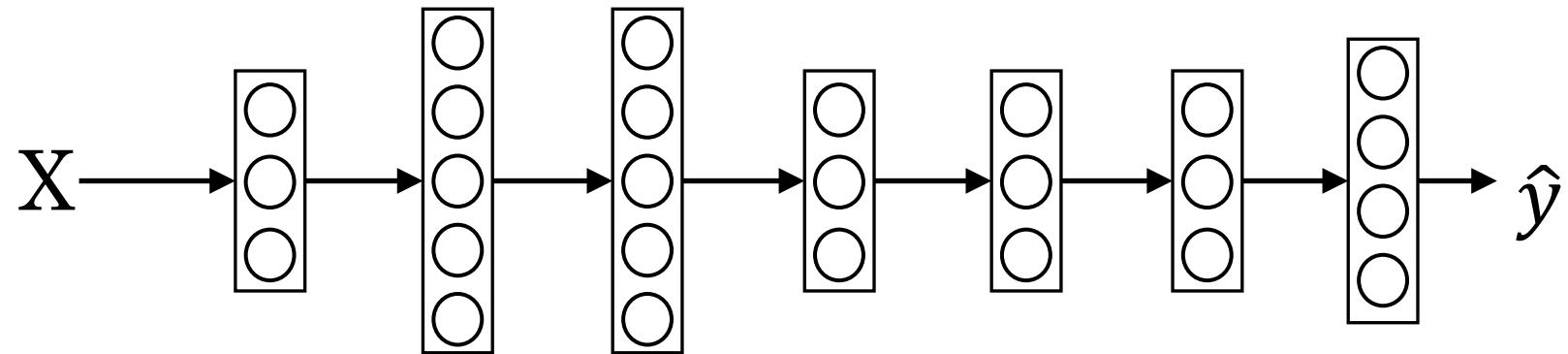
Multi-class
classification

Softmax regression

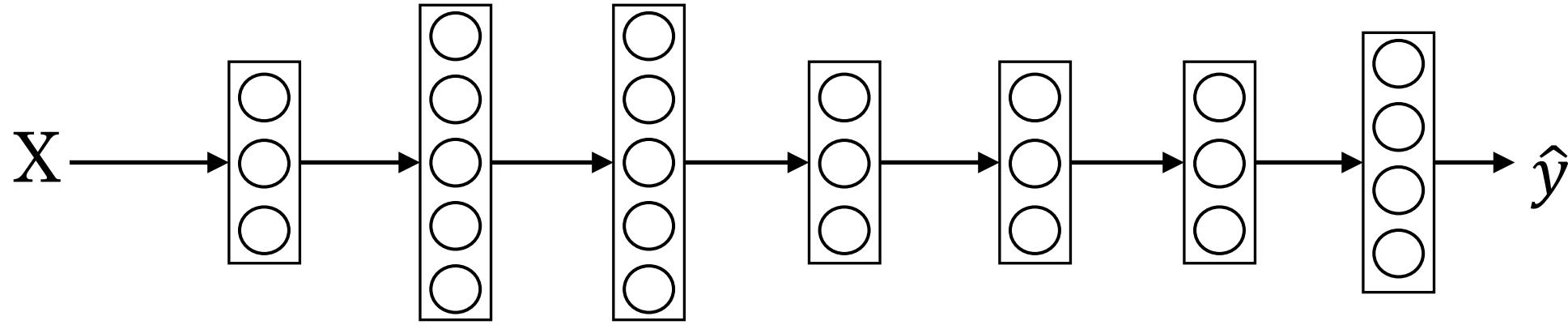
Recognizing cats, dogs, and baby chicks



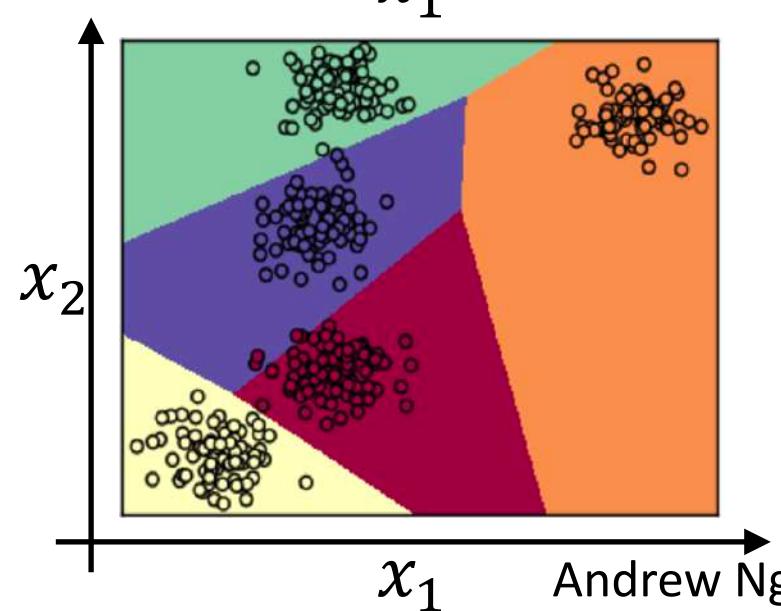
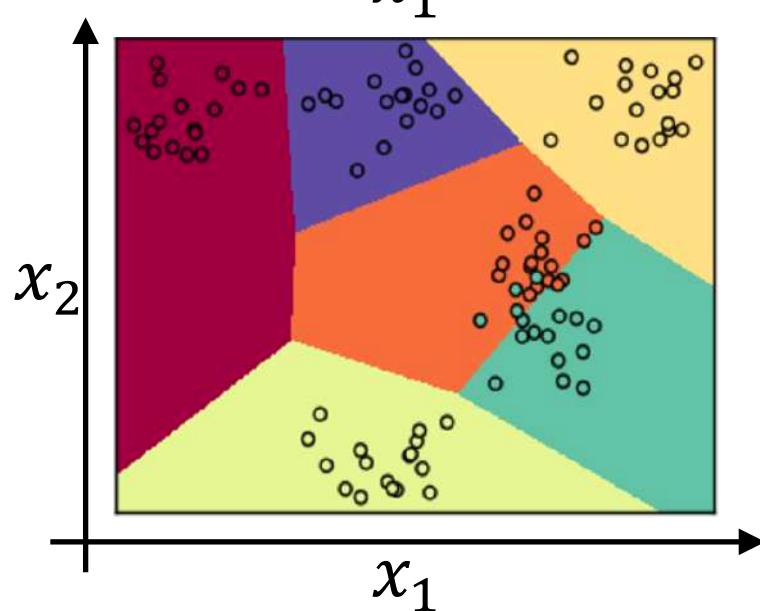
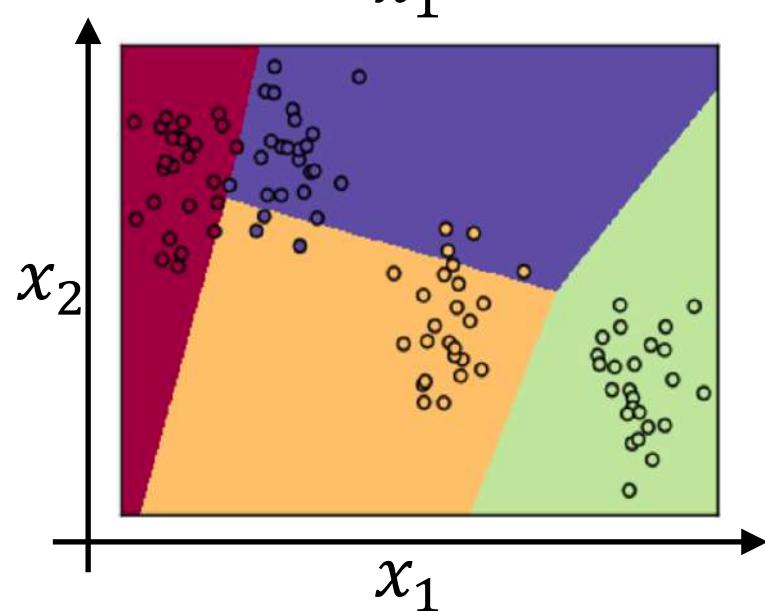
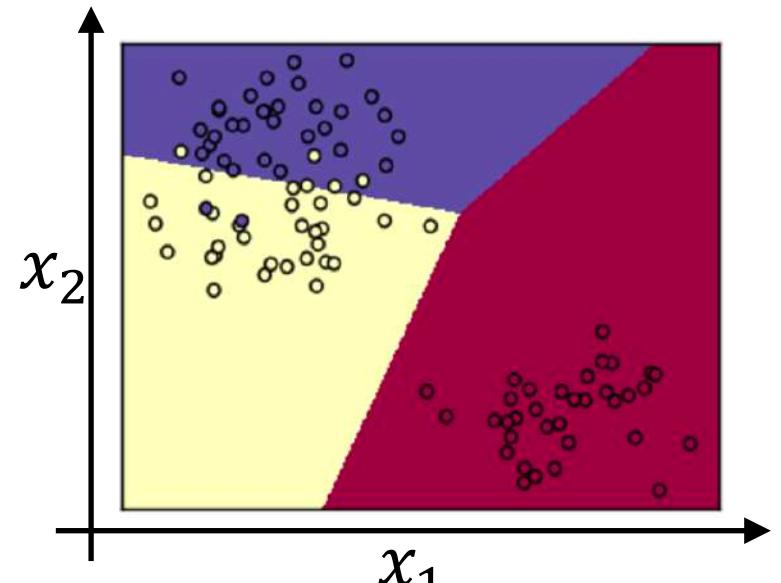
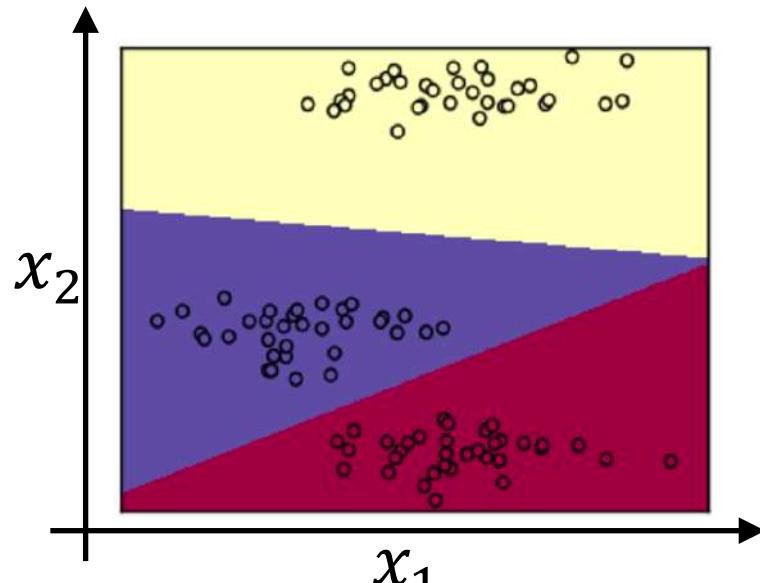
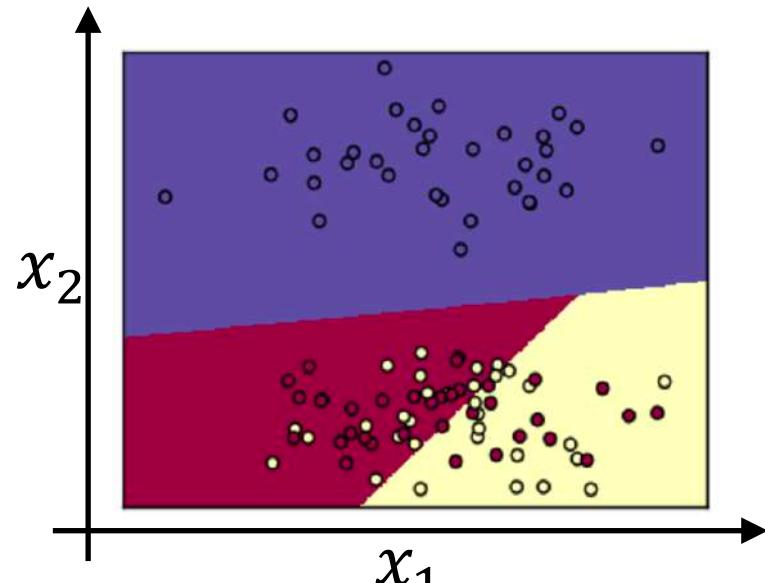
3 1 2 0 3 2 0 1



Softmax layer



Softmax examples



Andrew Ng



deeplearning.ai

Programming Frameworks

Deep Learning frameworks

Deep learning frameworks

- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Choosing deep learning frameworks

- Ease of programming (development and deployment)
 - Running speed
- - Truly open (open source with good governance)



deeplearning.ai

Programming Frameworks

TensorFlow

Motivating problem

$$J(\omega) = \frac{(\omega^2 - 10\omega + 25)}{(\omega - 5)^2}$$

$\omega = 5$

$J(\omega, b)$

↑ ↑

Code example

```
import numpy as np  
import tensorflow as tf  
  
coefficients = np.array([[1], [-20], [25]])
```

```
w = tf.Variable([0], dtype=tf.float32)
```

```
x = tf.placeholder(tf.float32, [3,1])
```

```
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0] # (w-5)**2
```

```
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
```

```
init = tf.global_variables_initializer()
```

```
session = tf.Session()
```

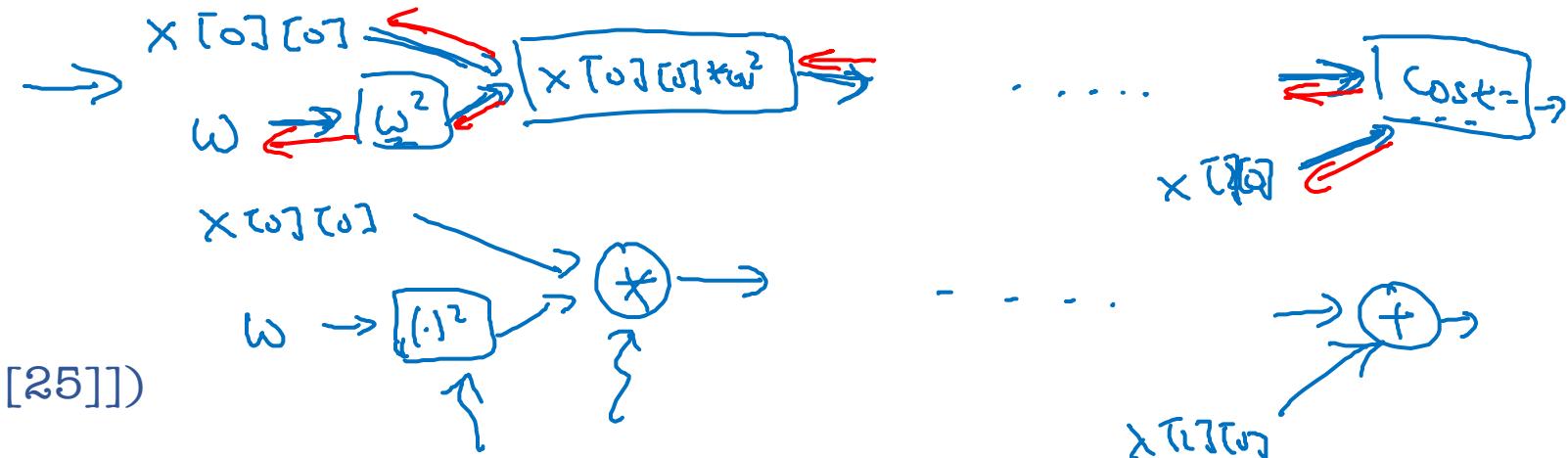
```
session.run(init)
```

```
print(session.run(w))
```

```
for i in range(1000):
```

```
    session.run(train, feed_dict={x:coefficients})
```

```
print(session.run(w))
```



```
with tf.Session() as session:
```

```
    session.run(init)
```

```
    print(session.run(w))
```