

COURSE 2: HYPERPARAMETER TUNING

Bias and Variance

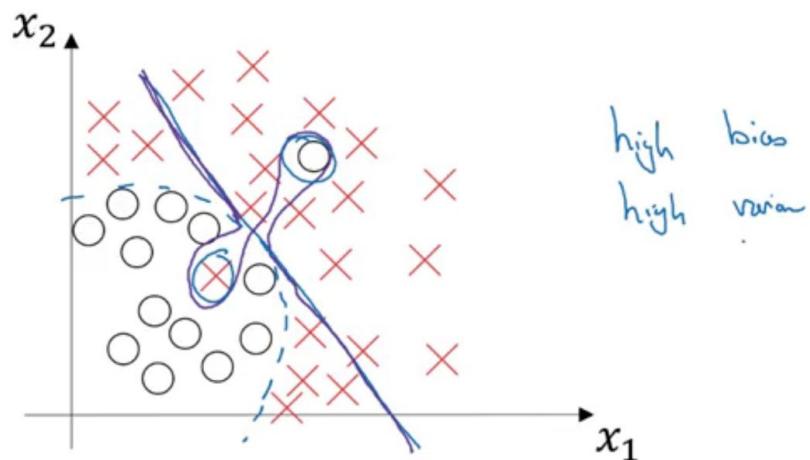
Cat classification



Train set error:	1%	15% ↗	15%	0.5%
Dev set error:	11%	16% ↗	30%	1%
	high variance	high bias	high bias & high varan	low bias low variance
<u>Human: ~0%</u>				
Optimal (Bayes) error: ~0 to 15%				

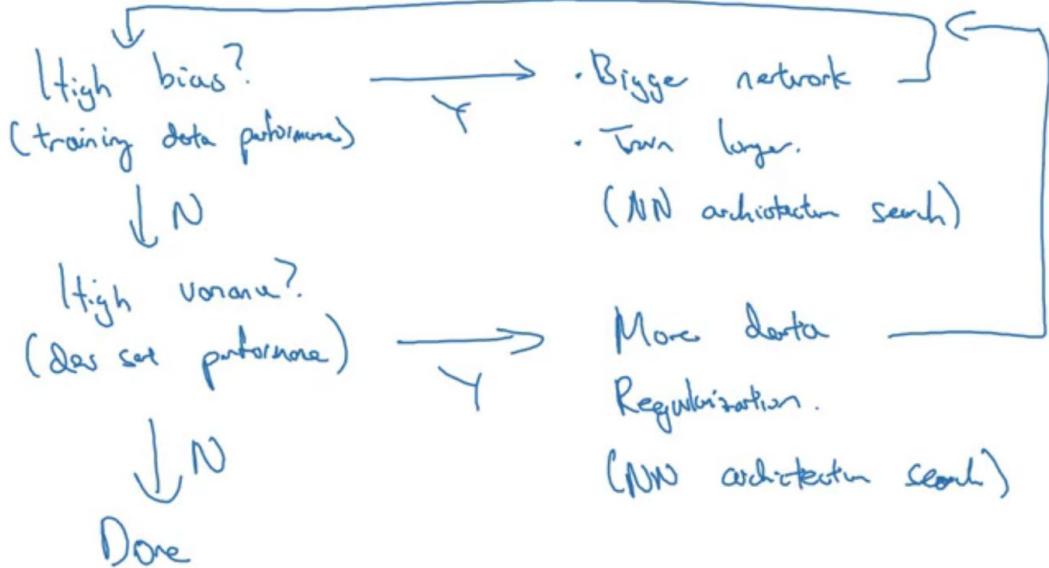
Andrew Ng

High bias and high variance

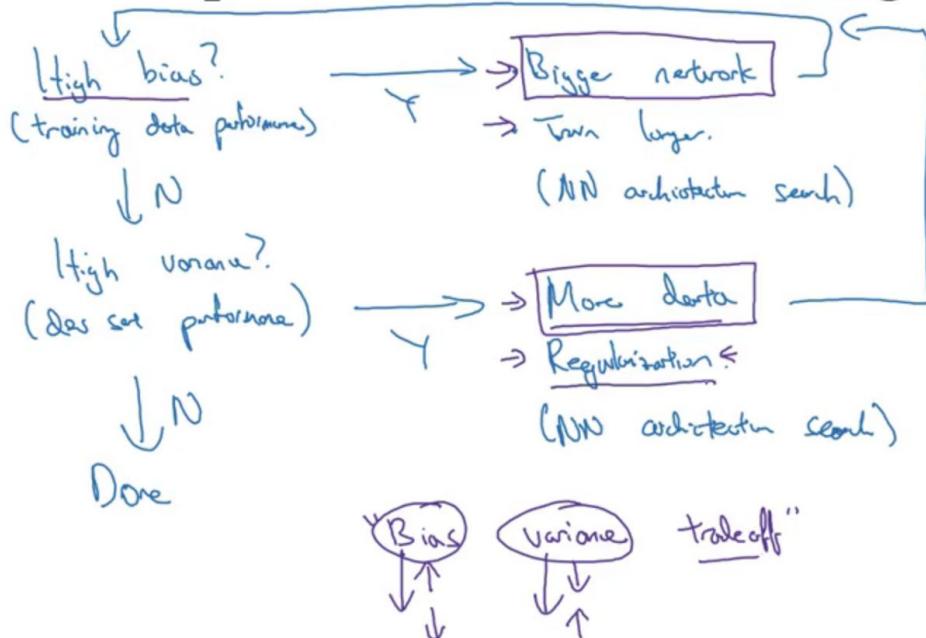


Mostly linear and sudden curves to overfit at some points

Basic recipe for machine learning



Basic recipe for machine learning



Training a bigger network and more data REDUCE both bias and variance

Logistic regression

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

lambda lambda

$$J(w, b) = \underbrace{\frac{1}{m} \sum_{i=1}^m \ell(\hat{y}^{(i)}, y^{(i)})}_{\text{L2 regularization}} + \underbrace{\frac{\lambda}{2m} \|w\|_2^2}_{\text{L1 regularization}}$$

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

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

L2 norm is also called weight decay

Neural network

$$J(w^{(0)}, b^{(0)}, \dots, w^{(L)}, b^{(L)}) = \underbrace{\frac{1}{m} \sum_{i=1}^m \ell(\hat{y}^{(i)}, y^{(i)})}_{\text{"Frobenius norm"} \quad \text{from backprop}} + \underbrace{\frac{\lambda}{2m} \sum_{l=1}^L \|w^{(l)}\|_F^2}_{\|w^{(l)}\|_F^2 = \sum_{i=1}^{n^{(l+1)}} \sum_{j=1}^{n^{(l)}} (w_{ij}^{(l)})^2 \quad w: (n^{(0)} \ n^{(1)} \ \dots)}$$

$$\|w^{(l)}\|_F^2 = \sum_{i=1}^{n^{(l+1)}} \sum_{j=1}^{n^{(l)}} (w_{ij}^{(l)})^2 \quad \|\cdot\|_2^2 \quad \|\cdot\|_F^2$$

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

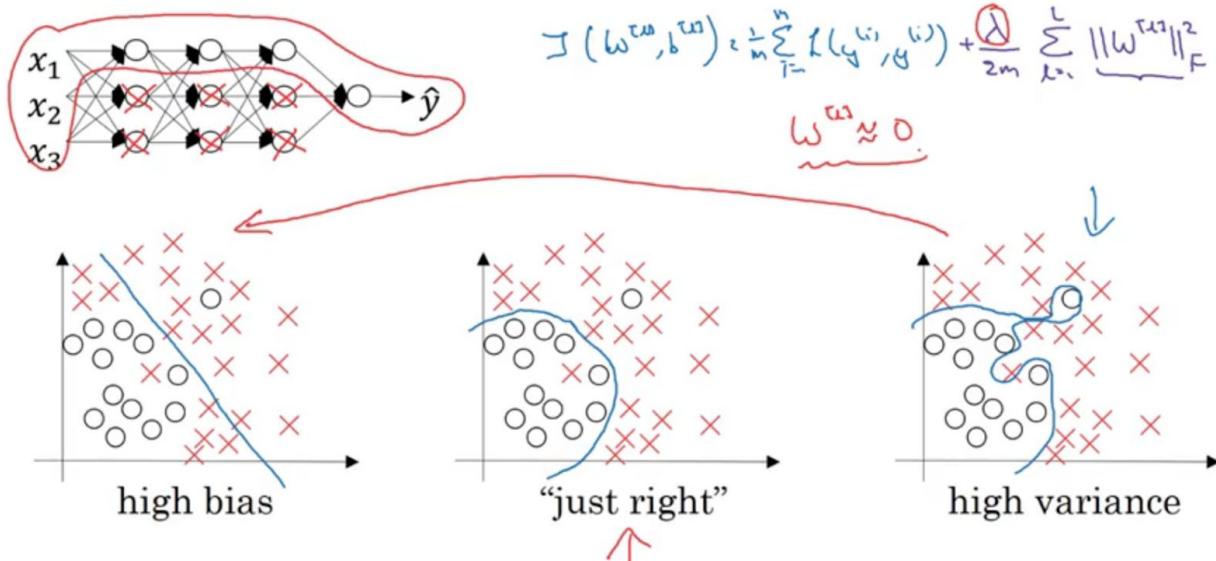
$$\partial w^{(l)} = \boxed{(\text{from backprop}) + \frac{\lambda}{m} w^{(l)}} \quad \rightarrow w^{(l)} := w^{(l)} - \alpha \partial w^{(l)}$$

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

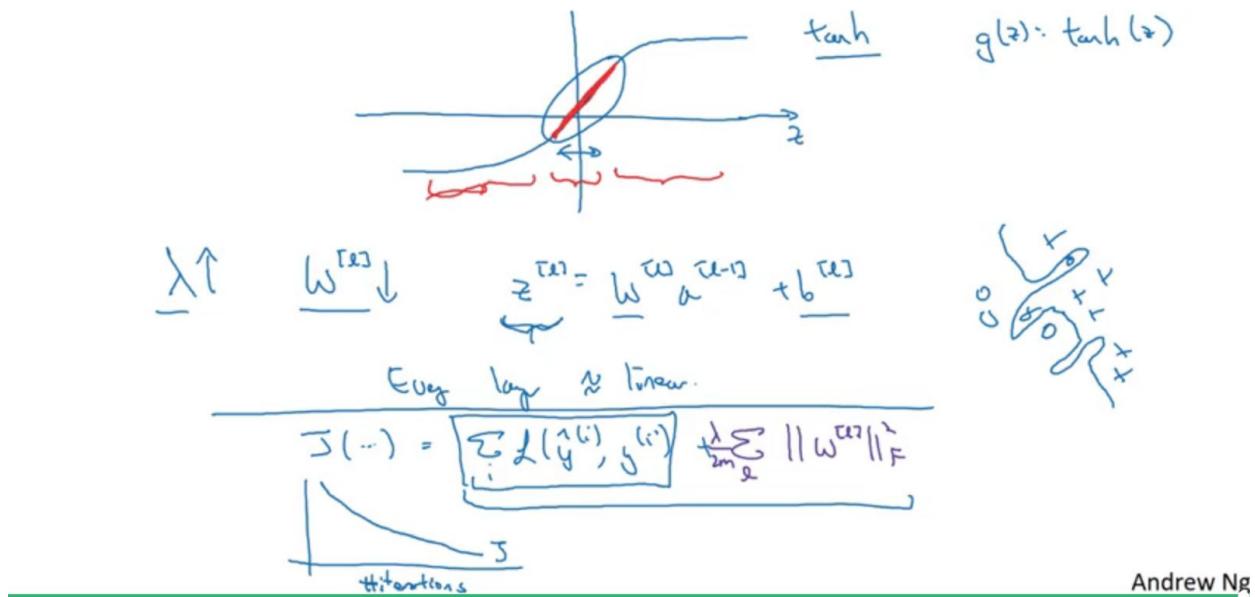
$$(1 - \frac{\alpha \lambda}{m}) w^{(l)} = \underbrace{w^{(l)} - \frac{\alpha \lambda}{m} w^{(l)}}_{- \alpha (\text{from backprop})}$$

Andrew

How does regularization prevent overfitting?



How does regularization prevent overfitting?



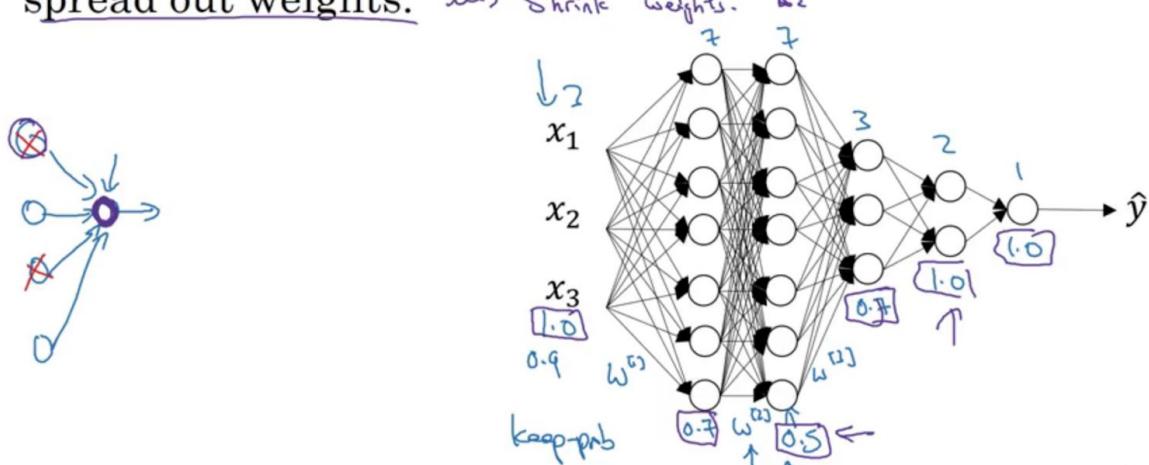
Regularization actually reduces the weights and thus reduces the Z and thus keeps the activation of Z in the linear region, thus preventing VERY complex fitting to data like the one shown above on right.

Dropout has the same affect as L2 regularization (Somewhat) -- dropout you are spreading the weights and not put all the bets on one input, in L2 regularization you are penalizing large weights. So the effect is similar.

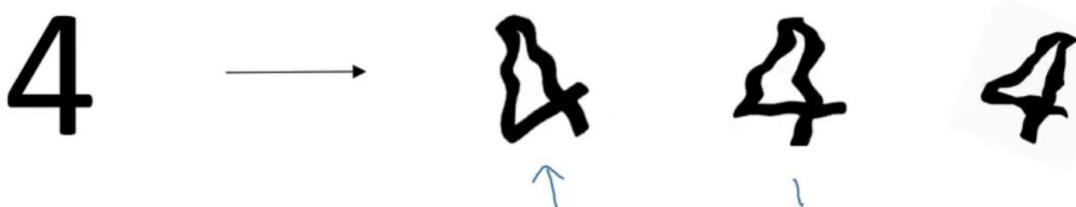
Some layers might have more keep_prob or less keep_prob -- ones that you are more scared about for overfitting you can keep lower keep prob.

Why does drop-out work?

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

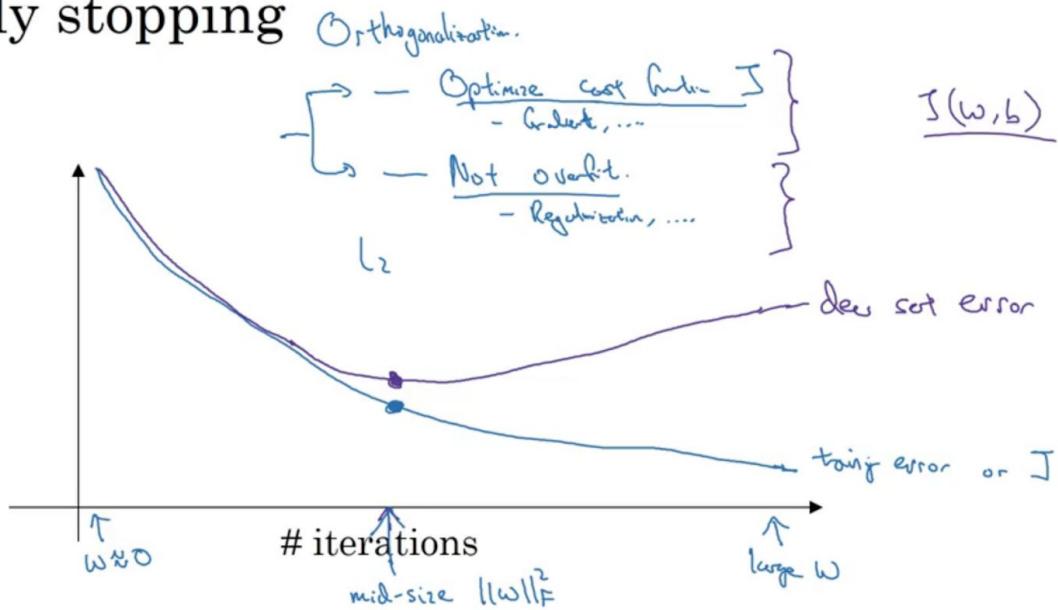


Data augmentation



This is not as good as getting more training examples, but it is inexpensive way of doing regularization to the data.

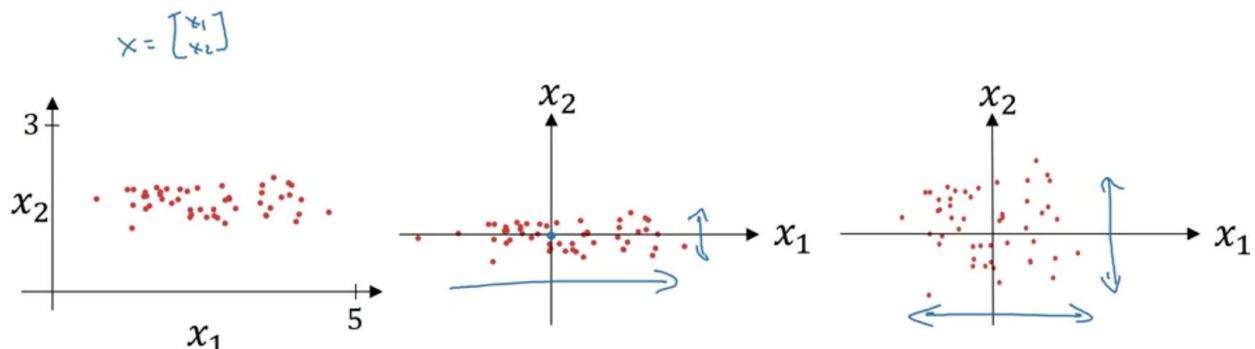
Early stopping



L_2 is better for orthogonalization -- separating task for first getting minimum cost and then not overfit. Early stopping kind of combines both at the same time -- so its little more complex. Disadvantage of L_2 is that you need to try out many different lambdas before getting a good one so its more computationally expensive.

NORMALIZATION:

Normalizing training sets



Subtract mean:

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

$$x := x - \bar{x}$$

Normalize variance

$$\frac{1}{m} \sum_{i=1}^m (x^{(i)} - \bar{x})^2$$

σ^2 elmt-wise

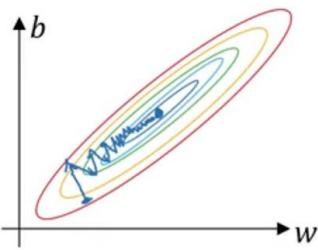
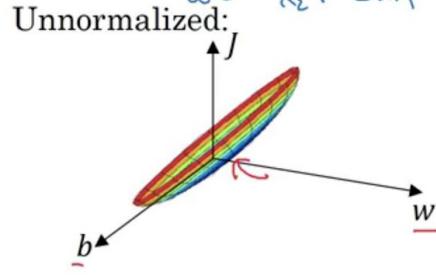
$$\frac{x}{\sigma}$$

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

Andrew Ng

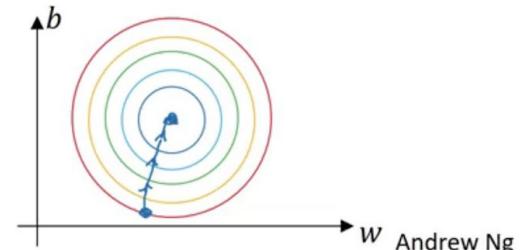
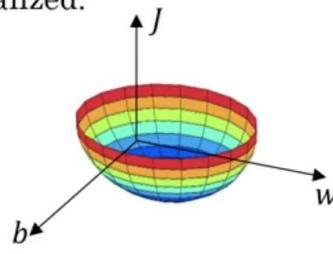
Why normalize inputs?

$w_1 \quad x_1: 1 \dots 1000$
 $w_2 \quad x_2: 0 \dots 1$



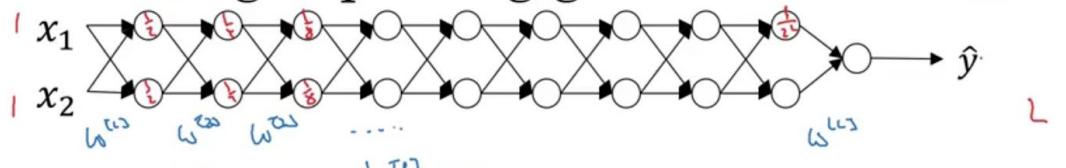
$$J(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

Normalized:



You need very small learning rate for the left -- unnormalized data

Vanishing/exploding gradients



$$\hat{y} = w^{[L]} \underbrace{\left(\underbrace{w^{[L-1]} \cdots w^{[1]}}_{\text{Input } x} \right)}_{z^{[L-1]} = w^{[L-1]} x} \cdots \underbrace{\left(\underbrace{w^{[2]} \cdots w^{[1]}}_{\text{Input } x} \right)}_{z^{[1]} = w^{[1]} x} \underbrace{g(z^{[1]})}_{a^{[1]}} = g(w^{[1]} a^{[1]})$$

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

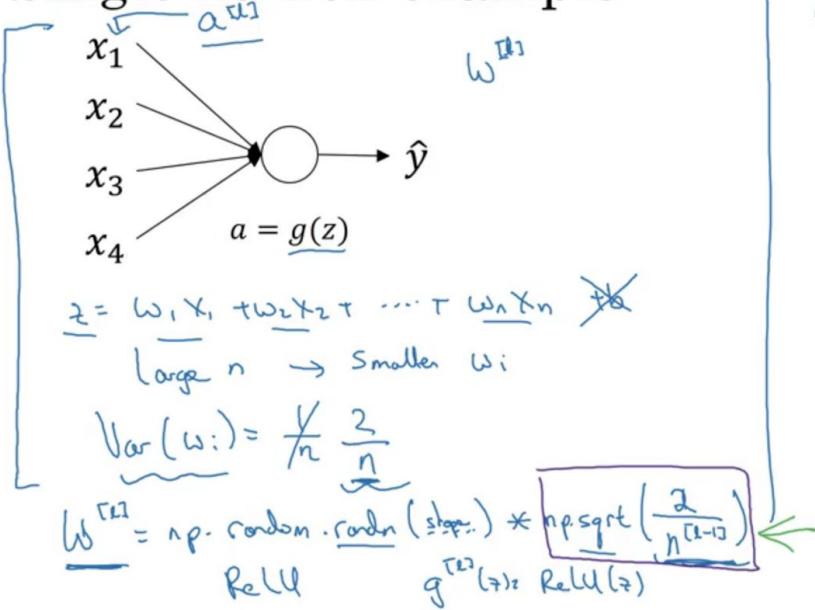
$$w^{[L]} < I \quad [0.9 \quad 0.9]$$

$$w^{[L]} = \begin{bmatrix} 0.5 & 0 \\ 0 & +5 \\ 0 & -5 \end{bmatrix}$$

$$\hat{y} = \underbrace{w^{[L]} \left[\underbrace{\begin{bmatrix} 0.5 & 0 \\ 0 & +5 \\ 0 & -5 \end{bmatrix}^{L-1} \times}_{a^{[L-1]} = g(z^{[L-1]}) = g(w^{[L-1]} a^{[L-1]})} \cdots \underbrace{\begin{bmatrix} 0.5 & 0 \\ 0 & +5 \\ 0 & -5 \end{bmatrix}^{1-1} \times}_{a^{[1]} = g(z^{[1]}) = g(w^{[1]} a^{[1]})} \underbrace{w^{[1]} x}_{z^{[1]} = w^{[1]} x} \right]}_{1.5^{L-1} \times 0.5^{L-1} \times}$$

Andrew Ng

Single neuron example



Other variants:
Tanh

Xavier initialization

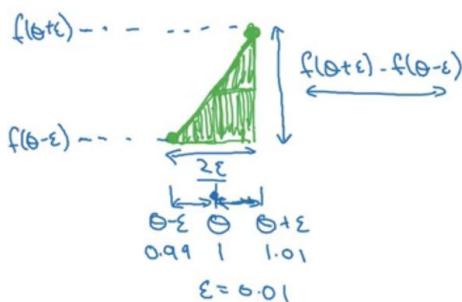
$$\frac{2}{n^{[l-1]} + n^m}$$

Andrew Ng

$2/\sqrt{n}$ for relu for random variable variance. Reason is more number of layers, you want weights to be smaller

Checking your derivative computation

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



$$\frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon} \approx g(\theta)$$

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

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

Approx error: 0.0001
(prev slide: 3.0301 , error: 0.03)

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

Andrew Ng

Two sided difference is better for derivative compared to single sided derivative

Gradient checking (Grad check)

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

for each i :

$$\rightarrow \underline{\Delta\theta_{\text{approx}}[i]} = \frac{J(\theta_0, \theta_1, \dots, \theta_i + \varepsilon, \dots) - J(\theta_0, \theta_1, \dots, \theta_i - \varepsilon, \dots)}{2\varepsilon}$$

$$\approx \underline{\Delta\theta[i]} = \frac{\partial J}{\partial \theta_i} \quad | \quad \Delta\theta_{\text{approx}} \approx \Delta\theta$$

Check

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

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

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

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

Batch vs. mini-batch gradient descent

Vectorization allows you to efficiently compute on m examples.

$$X = [x^{(1)} \ x^{(2)} \ x^{(3)} \ \dots \ x^{(m)}] \quad | \quad \dots \quad | \quad \dots \ x^{(5000)} \\ \underbrace{x^{(1)}}_{(n_x, m)} \quad \underbrace{x^{(2)} \dots x^{(1000)}}_{(n_x, 1000)} \quad \underbrace{x^{(5001)} \dots x^{(5000)}}_{(n_x, 1000)} \quad \dots \quad \underbrace{x^{(5000)}}_{(n_x, 1000)}$$

$$Y = [y^{(1)} \ y^{(2)} \ y^{(3)} \ \dots \ y^{(m)}] \quad | \quad \dots \quad | \quad \dots \ y^{(5000)} \\ \underbrace{y^{(1)} \dots y^{(1000)}}_{(1, m)} \quad \underbrace{y^{(1001)} \dots y^{(2000)}}_{(1, 1000)} \quad \dots \quad \underbrace{y^{(5000)}}_{(1, 1000)}$$

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

5,000 mini-batches of 1,000 each

Mini-batch t : X^{t+3}, Y^{t+3}

$$\begin{array}{l} X^{(i)} \\ z^{(t)} \\ X^{t+3} \end{array}$$

$X(i)$ is i th training example

$Z[t]$ is t th layer

$X[t]$ is t mini batch

Mini-batch gradient descent

for $t = 1, \dots, 5000 \}$

Forward prop on X^{t+1} .

$$z^{(t)} = w^{(t)} X^{t+1} + b^{(t)}$$

$$A^{(t)} = g^{(t)}(z^{(t)})$$

$$A^{(t)} = g^{(t)}(z^{(t)})$$

$$\text{Compute cost } J^{t+1} = \frac{1}{m} \sum_{i=1}^m l(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot m} \sum_{j=1}^n \|w^{(t)}\|_F^2.$$

Backprop to compute gradients wrt J^{t+1} (using (X^{t+1}, Y^{t+1}))

$$\tilde{w}^{(t)} = w^{(t)} - \alpha \nabla J^{t+1}, \quad \tilde{b}^{(t)} = b^{(t)} - \alpha \nabla b^{(t)}$$

}

"1 epoch"

pass through training set.

1 step of gradient descent
using $\underline{X^{t+1}}, \underline{Y^{t+1}}$
(as if $m=1000$)

X, Y

Andrew Ng

Epochs are not needed for batch gradient descent as you just need one step.

But for mini-batch - you will take 5000 steps for one training set. That's why you need epochs to be absolutely sure.

SGD problem-- lose advantage of vectorization!!

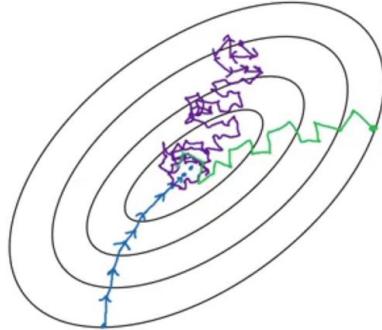
Mini batch is best as you can speed up --vectorization as well as time aspect

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
 $(X^{(1)}, Y^{(1)}) = (x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)})$ mini-batch.

In practice: Somewhere in between \downarrow and \uparrow



Stochastic
gradient
descent
↓
Use sparse
feature vectorization

In-between
(mini-batch size
not too big/small)
↓
Fastest learning.
• Vectorization.
(w_{1000})
• Make passes without
processing entire tiny set.

Batch
gradient descent
(mini-batch size = m)
↓
Too long
per iteration

Andrew Ng

Choosing your mini-batch size

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

Typical mini-batch sizes:

$$\underbrace{64, 128, 256, 512}_{2^6, 2^7, 2^8, 2^9} \quad \frac{1024}{2^{10}}$$

Make sure mini-batch fits in CPU/GPU memory.
 $X^{(k)}, Y^{(k)}$.

Exponentially weighted averages

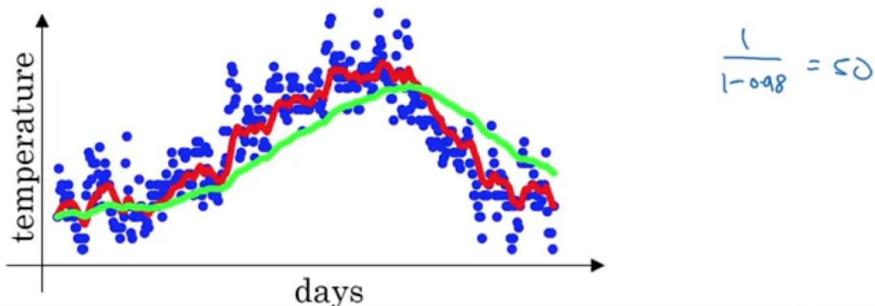
$$\underline{V_t} = \underline{\beta} \underline{V_{t-1}} + \underline{(1-\beta) \underline{\Theta_t}}$$

$\beta = 0.9$: ≈ 10 days 'teperatur'

$\beta = 0.98$: ≈ 50 days

$\beta = 0.5$: ≈ 2 days

We can approximately
overlook other
 $\rightarrow n \frac{1}{1-\beta}$ days
temperature.



To calculate statistical moving average as shown above

Exponentially weighted averages

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

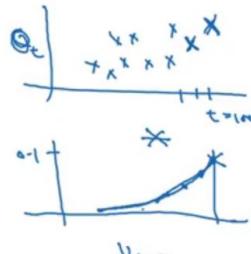
$$v_{100} = 0.9v_{99} + 0.1\theta_{100}$$

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

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

• • •

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



Exponentially weighted averages

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

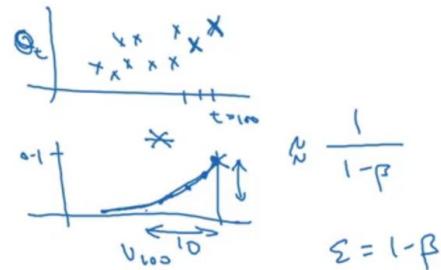
$$v_{100} = 0.9 v_{99} + 0.1 \theta_{100}$$

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

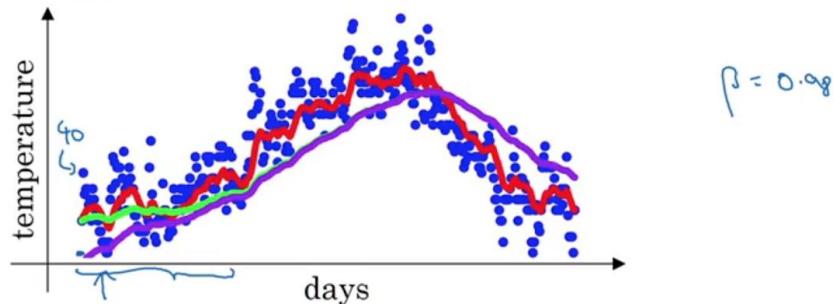
$$v_{98} = 0.9 v_{97} + 0.1 \theta_{98}$$

...

$$\begin{aligned} v_{100} &= 0.1 \theta_{100} + 0.9 (0.1 \theta_{99} + 0.9 (0.1 \theta_{98} + \dots)) \\ &= 0.1 \theta_{100} + 0.1 \times 0.9 \cdot 0.9 \theta_{99} + 0.1 (0.9)^2 \theta_{98} + 0.1 (0.9)^3 \theta_{97} + 0.1 (0.9)^4 \theta_{96} \\ 0.9^{10} &\approx 0.35 \approx \frac{1}{e} \quad \frac{(1-\varepsilon)^{\frac{1}{\varepsilon}}}{0.9} = \frac{1}{e} \quad \text{?} \\ &\varepsilon = 0.02 \rightarrow 0.98^{\frac{1}{0.02}} \approx \frac{1}{e} \quad \text{Andrew Ng} \end{aligned}$$



Bias correction



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

$$v_0 = 0$$

$$v_1 = 0.98 v_0 + 0.02 \theta_1$$

$$v_2 = 0.98 v_1 + 0.02 \theta_2$$

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

$$= 0.0196 \theta_1 + 0.02 \theta_2$$

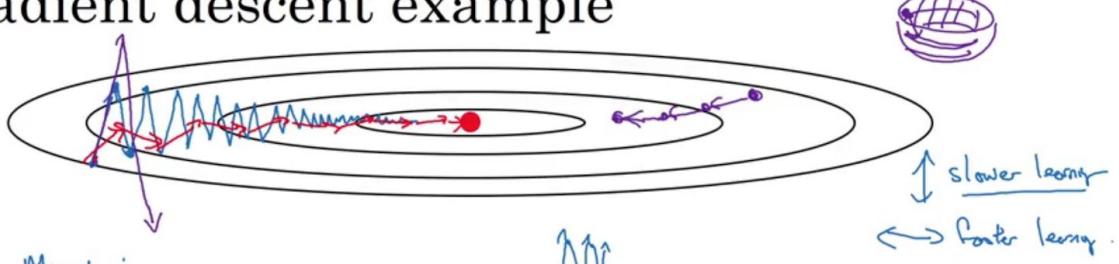
$$\begin{aligned} &\frac{v_t}{1 - \beta^t} \\ t=2: \quad 1 - \beta^t &= 1 - (0.98)^2 = 0.0396 \\ \frac{v_2}{0.0396} &= \frac{0.0196 \theta_1 + 0.02 \theta_2}{0.0396} \end{aligned}$$

Andrew Ng

In the beginning slow to warm up, as the previous days are all 0, so you do $v_t / 1 - b^t = b \cdot v_{t-1} + (1-b)$ theta --- reason being that initially the bias correction ratio is small and then gets larger and becomes irrelevant when t gets larger.

MOMENTUM:

Gradient descent example



Momentum:

On iteration t :

Compute dW, db on current mini-batch.

$$v_{dw} = \beta v_{dw} + (1-\beta) dW \quad "v_0 = \beta v_0 + (1-\beta) \theta_t"$$

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

↑ friction ↑ velocity ↑ acceleration

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

Implementation details

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

On iteration t :

Compute dW, db on the current mini-batch

$$\begin{aligned} v_{dw} &= \beta v_{dw} + (1-\beta) dW & | \quad v_{dw} = \beta v_{dw} + dW \leftarrow \\ v_{db} &= \beta v_{db} + (1-\beta) db \\ W &= W - \underbrace{\alpha v_{dw}}, \quad b = \underbrace{b - \alpha v_{db}} \end{aligned}$$

~~$$\frac{v_{dw}}{1-\beta^t}$$~~

Hyperparameters: α, β

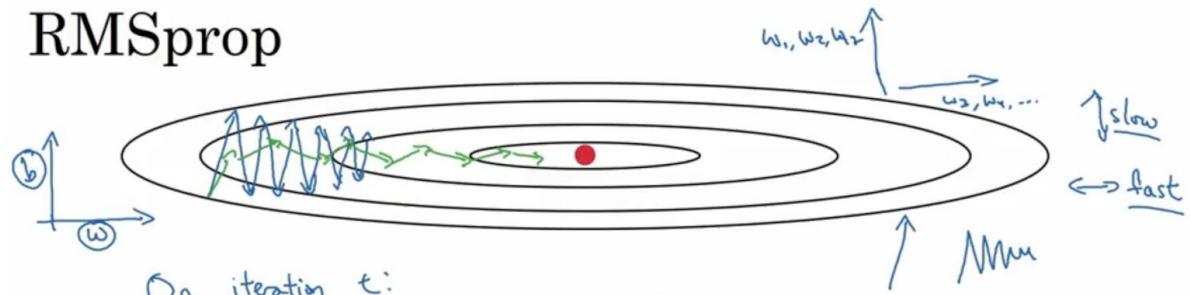
$$\beta = 0.9$$

average over last ≈ 10 gradients

Andrew Ng

Right in purple is in literature but left is more intuitive

RMSprop



On iteration t :

Compute dW, db on current mini-batch
element-wise

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

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

$$w := w - \alpha \frac{dW}{\sqrt{S_{dw} + \epsilon}} \quad b := b - \alpha \frac{db}{\sqrt{S_{db} + \epsilon}}$$

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

Square and square-root! Larger learning rate and speed up learning!! You want horizontal side to be fast and vertical to be slow!! You want larger learning rate so that you can train your NN faster!

ADAM:

Adam optimization algorithm

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

On iteratn t :

Compute dW, db using current mini-batch

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

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

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

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

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

Hyperparameters choice:

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

Adam: Adaptive moment estimation

Learning rate decay

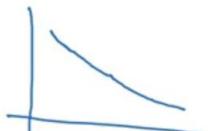
1 epoch = 1 pass through data.

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

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
:	:



$$\alpha_0 = 0.2$$
$$\text{decy. rate} = 1$$

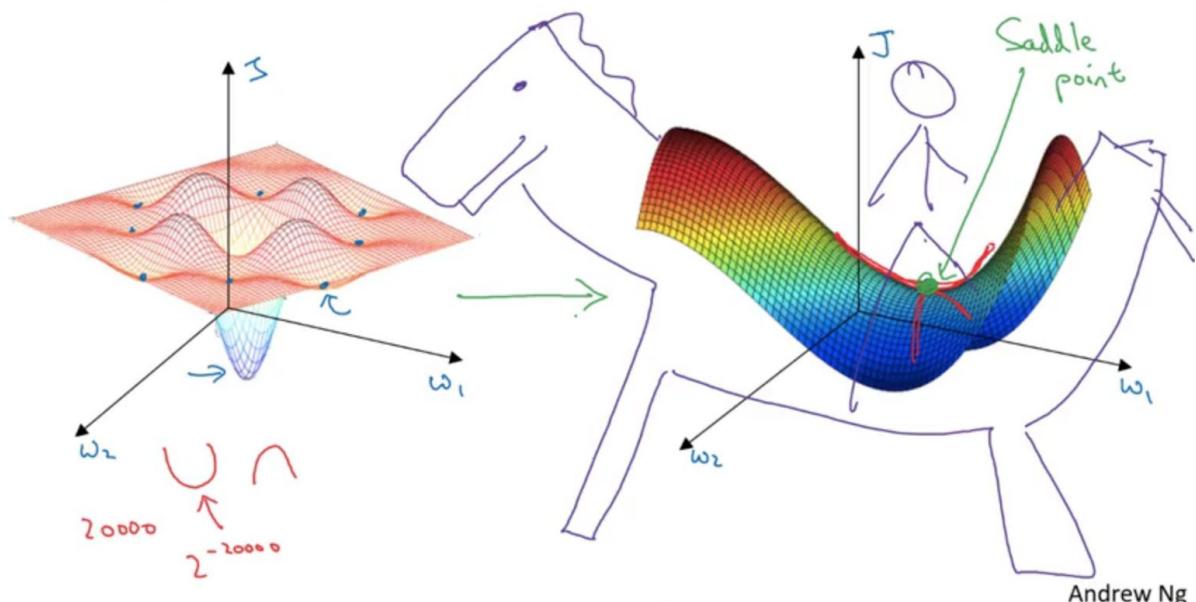


Other learning rate decay methods

$$\left\{ \begin{array}{l} \alpha = 0.95^{\text{epoch_num}} \cdot \alpha_0 \quad \text{- exponentially decay.} \\ \alpha = \frac{k}{\sqrt{\text{epoch_num}}} \cdot \alpha_0 \quad \text{or} \quad \frac{k}{\sqrt{t}} \cdot \alpha_0 \end{array} \right.$$

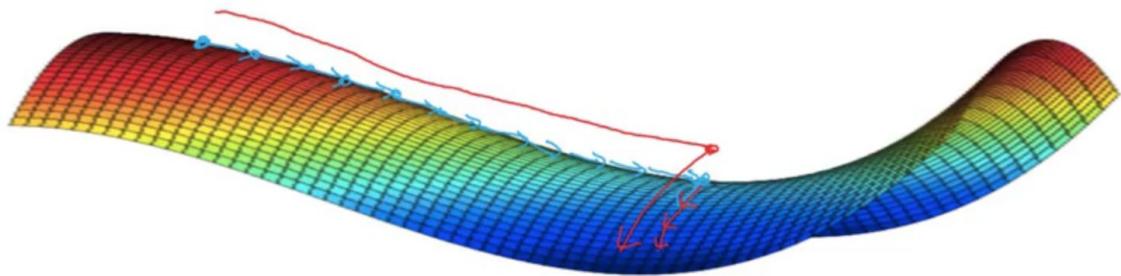
 discrete staircase

Local optima in neural networks



instead of local optima its usually saddle points in high D space

Problem of plateaus

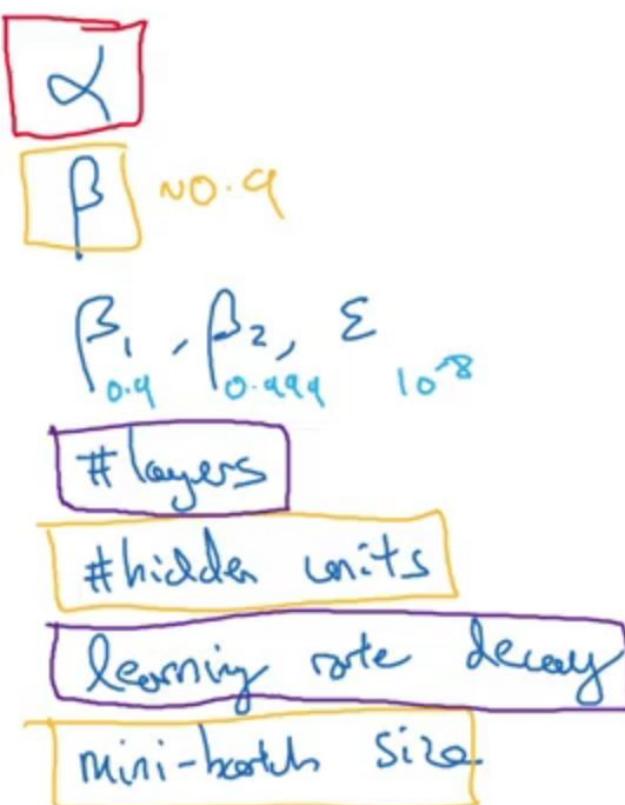


- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow

Adam can help a lot with plateaus!

TUNING THE HYPERPARAMETERS

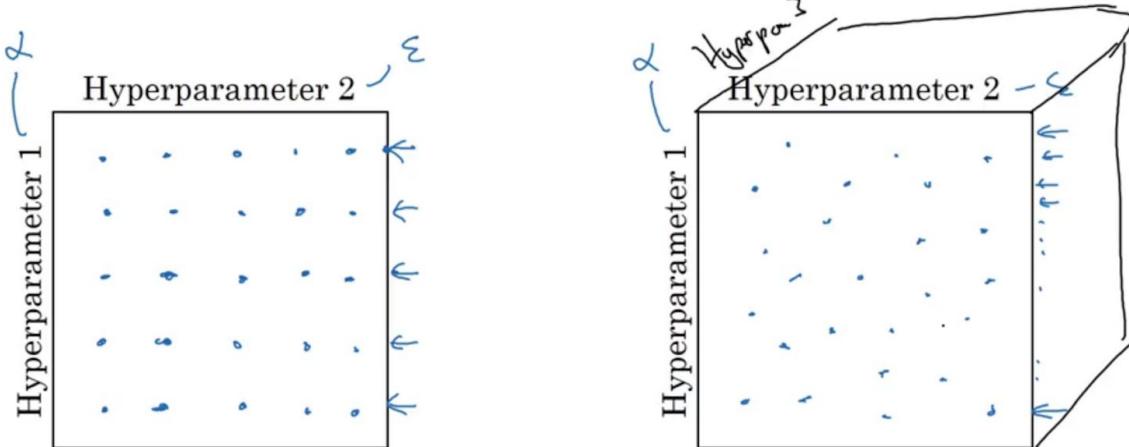
Hyperparameters



RED to ORANGE to PURPLE in terms of importance

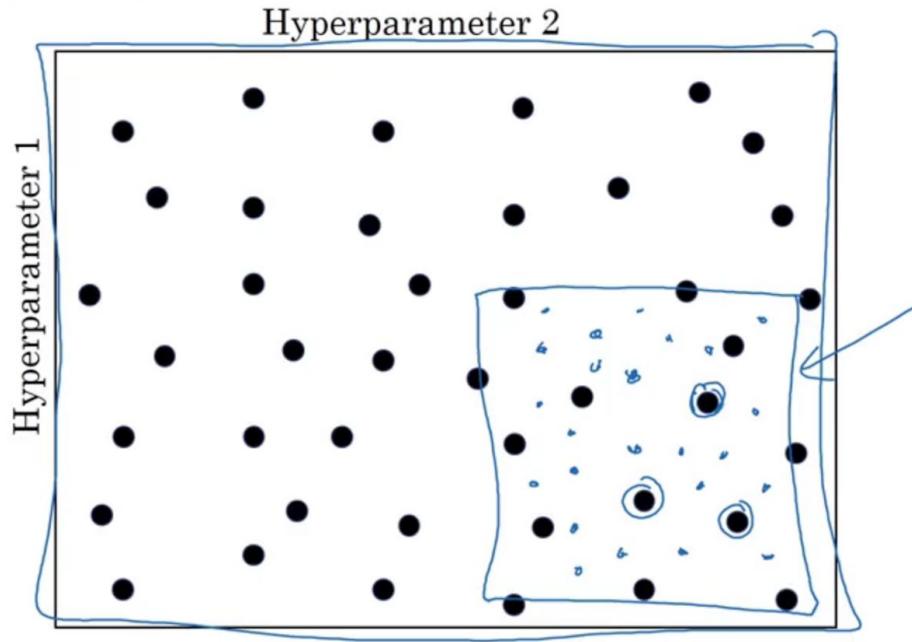
1. Learning Rate
2. Hidden Units, Mini Batch Size, Momentum
3. Learning Rate Decay and No of Layers

Try random values: Don't use a grid



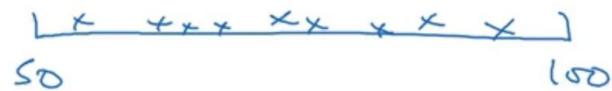
Why random values for hyper-parameters? Because you are trying 25 random values for both 1 and 2 instead of just 5 for Hyper Parameter 1 (if h2 is not much relevant) -- randomness you can try many more values and see how the hyperparameter behaves.

Coarse to fine



Picking hyperparameters at random

$$\rightarrow n^{[l]} = 50, \dots, 100$$



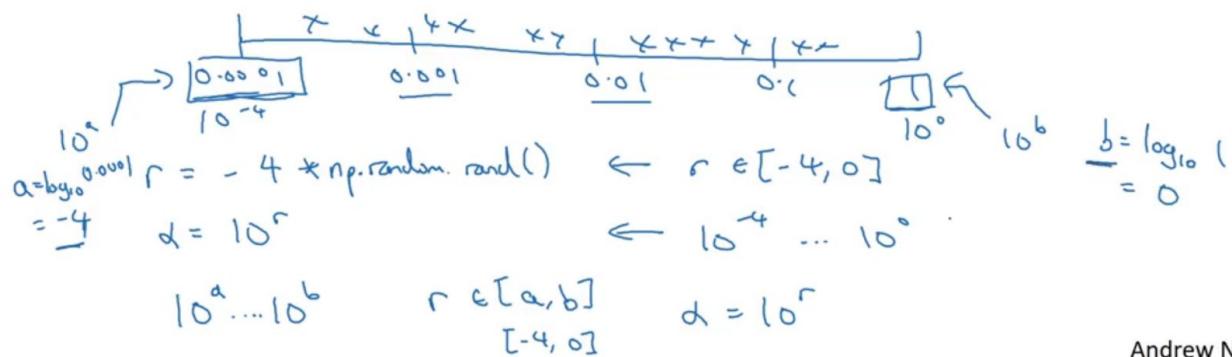
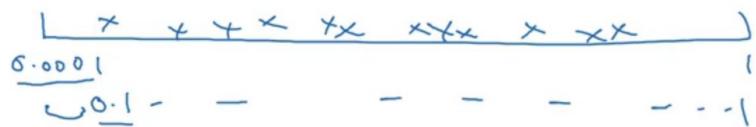
$$\rightarrow \# \text{layers} \quad L: 2 - 4$$

2, 3, 4

At random works for specific scales, but not for things like learning rates (use log instead for this)

Appropriate scale for hyperparameters

$$\alpha = 0.0001, \dots, 1$$



Andrew Ng

Hyperparameters for exponentially weighted averages

$$\beta = 0.9 \dots 0.999$$

\downarrow \downarrow

10 1000

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

$\beta: 0.900 \rightarrow 0.9005$

$\beta: 0.999 \rightarrow 0.9995$

.

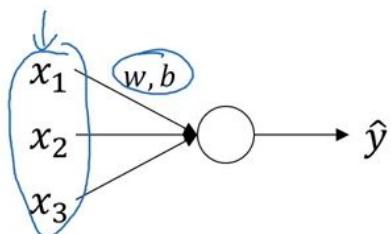
10^{-1} 10^{-3}

$r \in [-3, -1]$

$1 - \beta = 10^r$

$\beta = 1 - 10^r$

Normalizing inputs to speed up learning

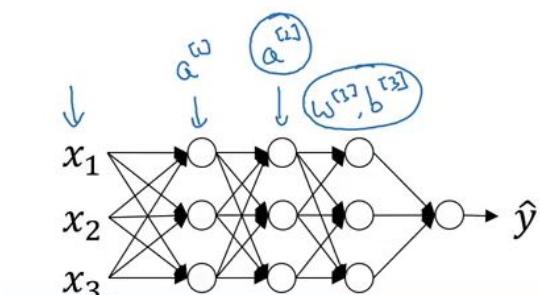
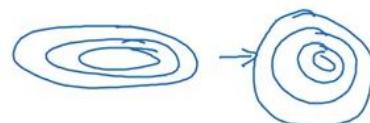


$$\mu = \frac{1}{m} \sum_i x^{(i)}$$

$$X = X - \mu$$

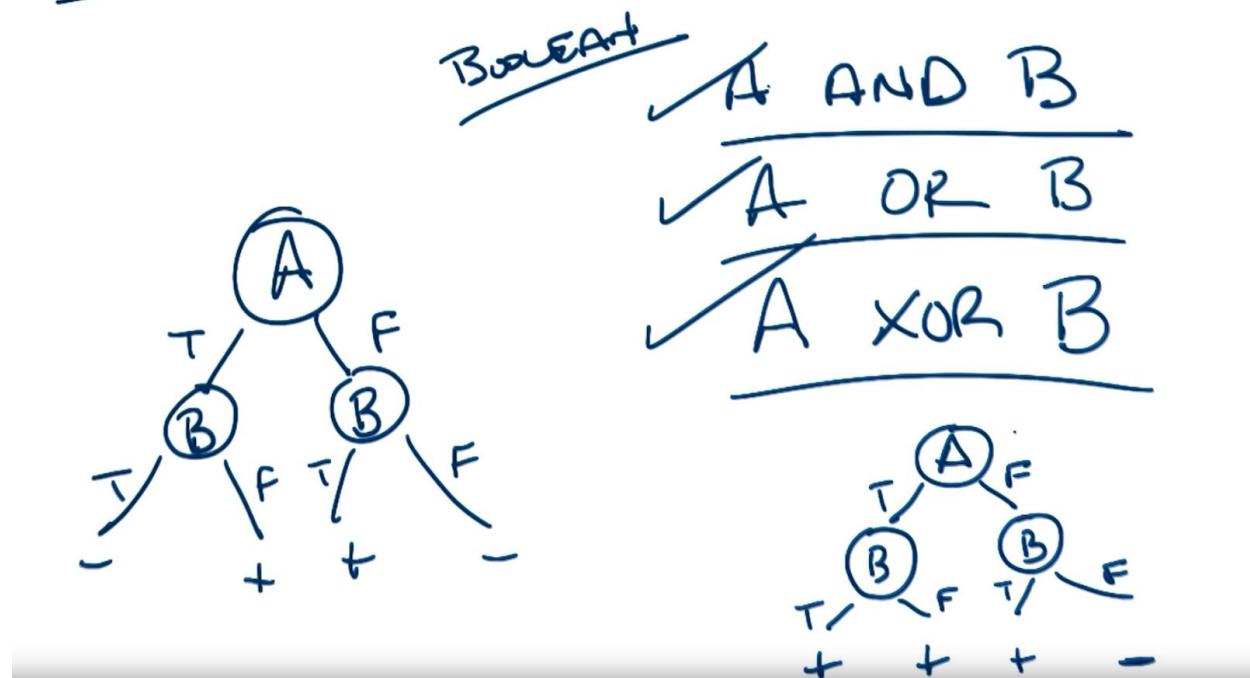
$$\sigma^2 = \frac{1}{m} \sum_i (x^{(i)} - \mu)^2$$

$$X = X / \sigma^2$$



Can we normalize $w^{[2]}, b^{[2]}$ so
as to train $w^{[2]}, b^{[2]}$ faster
Normalizing $z^{[2]}$

DECISION TREES : EXPRESSIVENESS



Implementing Batch Norm

Given some intermediate values in NN $\underbrace{z^{(1)}, \dots, z^{(n)}}_{z^{(i)}}$

$$\begin{cases} \mu = \frac{1}{m} \sum z^{(i)} \\ \sigma^2 = \frac{1}{m} \sum (z^{(i)} - \mu)^2 \\ z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \hat{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta \end{cases}$$

If $\gamma = \sqrt{\sigma^2 + \epsilon} \leftarrow$
 $\beta = \mu \leftarrow$
 then $\hat{z}^{(i)} = z^{(i)}$

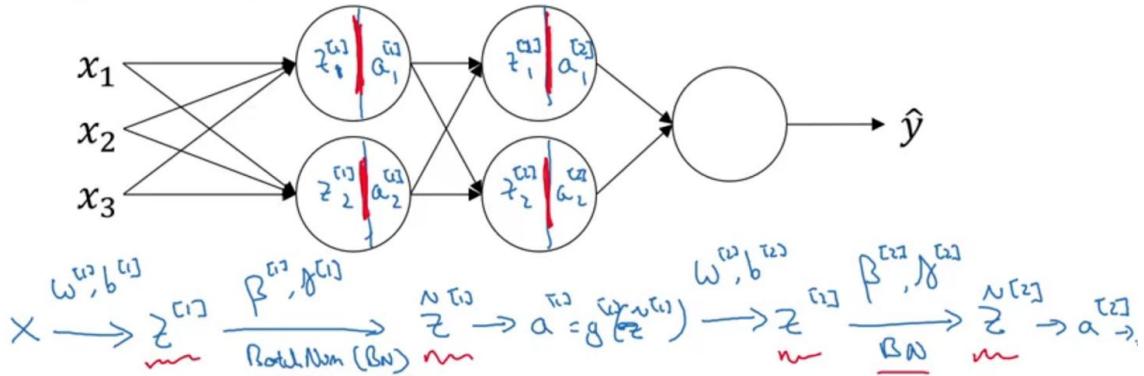
X \leftarrow
 $\hat{z}^{(i)} \leftarrow$

learnable parameters of model.

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

Batch norm is dividing by standard deviation, you can also do arbitrary mean and variance of normalized data (and not just 0 and 1 respectively)

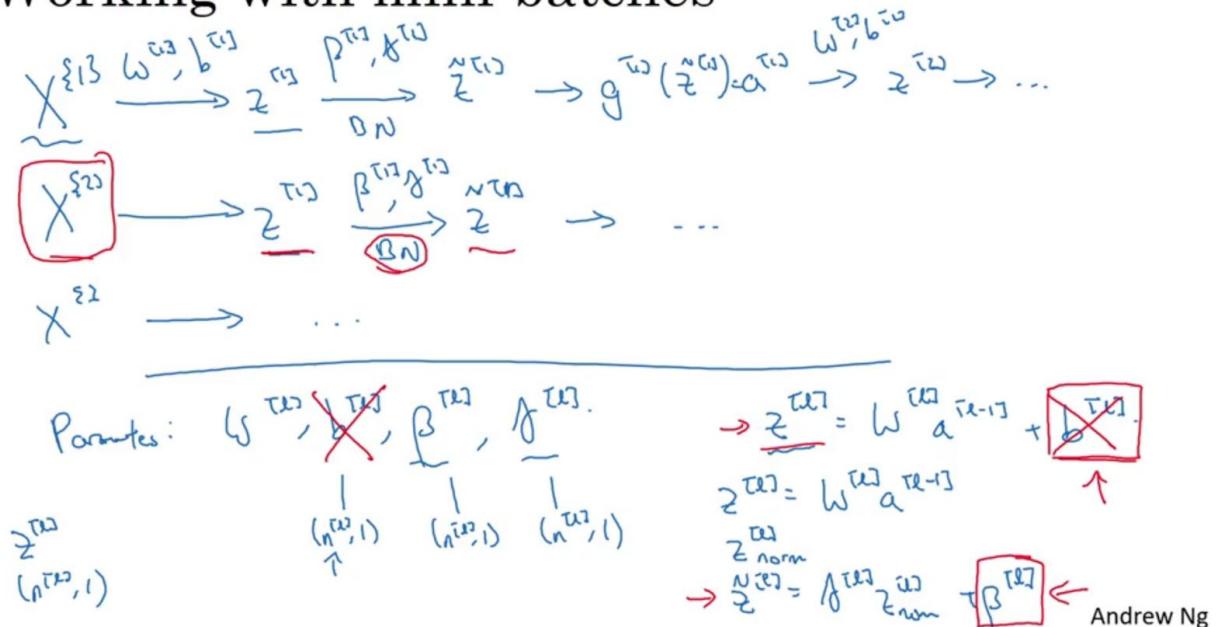
Adding Batch Norm to a network



Parameters: $w^{1,2}, b^{1,2}, w^{2,3}, b^{2,3}, \dots, w^{L-1, L}, b^{L-1, L}$,
 $\beta^{1,2}, \gamma^{1,2}, \beta^{2,3}, \gamma^{2,3}, \dots, \beta^{L-1, L}, \gamma^{L-1, L}$
 $\rightarrow \beta$

Andrew Ng

Working with mini-batches



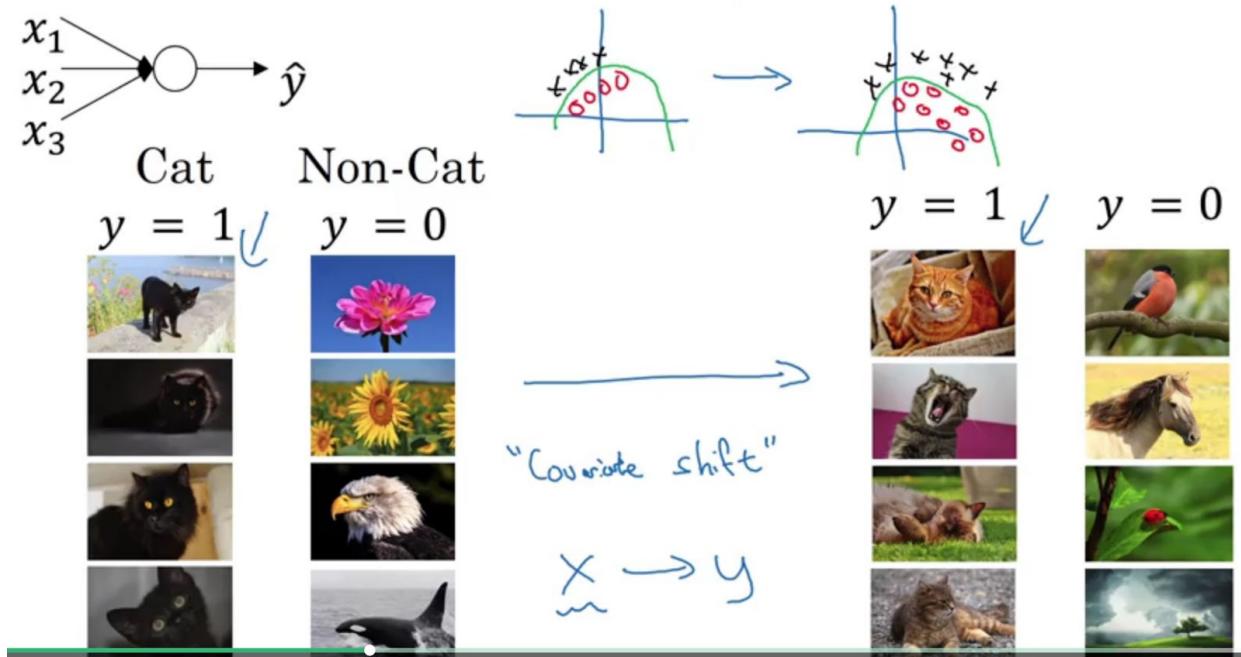
Remove bias terms initially before batch norm as all subtracted during subtraction of mean step, beta terms come in later anyways

Why Does Batch Norm work?

It is because the distribution of z values in hidden layers is changing a lot. Or rather covariate shift. This is because the distribution depends on previous values as well which are also changing based on cost function and gradient descent. So its all in a state of flux. Batch norm

kind of keeps a check on that. Atleast batch norm ensures that the change is restricted by mean of 0 and variance of 1.

Learning on shifting input distribution



Batch Norm as regularization

X

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
 $\tilde{z}^{[l]}$ $\mu_{\tilde{z}^{[l]}}$ $\sigma^2_{\tilde{z}^{[l]}}$
- 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 . 512

Unintended side effect of batch norm is regularization but dont rely on batch norm for regularization. Dropout is better for that.

Batch Norm at test time

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

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

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

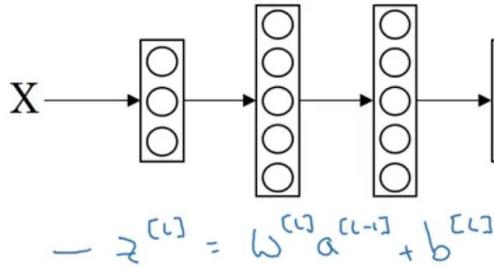
$$\tilde{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta$$

μ, σ^2 : estimate using exponentially weighted average (across mini-batches).
 $X^{(1)}, X^{(2)}, X^{(3)}, \dots$
 $\mu^{(1)}, \mu^{(2)}, \mu^{(3)}, \dots$
 $\theta_1, \theta_2, \theta_3, \dots$
 $\epsilon^{(1)}, \epsilon^{(2)}, \epsilon^{(3)}, \dots$
 $\tilde{z}_{\text{norm}} = \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}}$
 $\tilde{z} = \gamma \tilde{z}_{\text{norm}} + \beta$

Andrew Ng

MULTI-CLASS CLASSIFICATION

Softmax layer



$$z^{[L]} = \omega^{[L]} a^{[L-1]} + b^{[L]}$$

Activation function:

$$\rightarrow t = e^{z^{[L]}}$$

$$\rightarrow a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1}^4 t_j}, \quad a_i^{[L]} = \frac{t_i}{\sum_{j=1}^4 t_j}$$

layer L

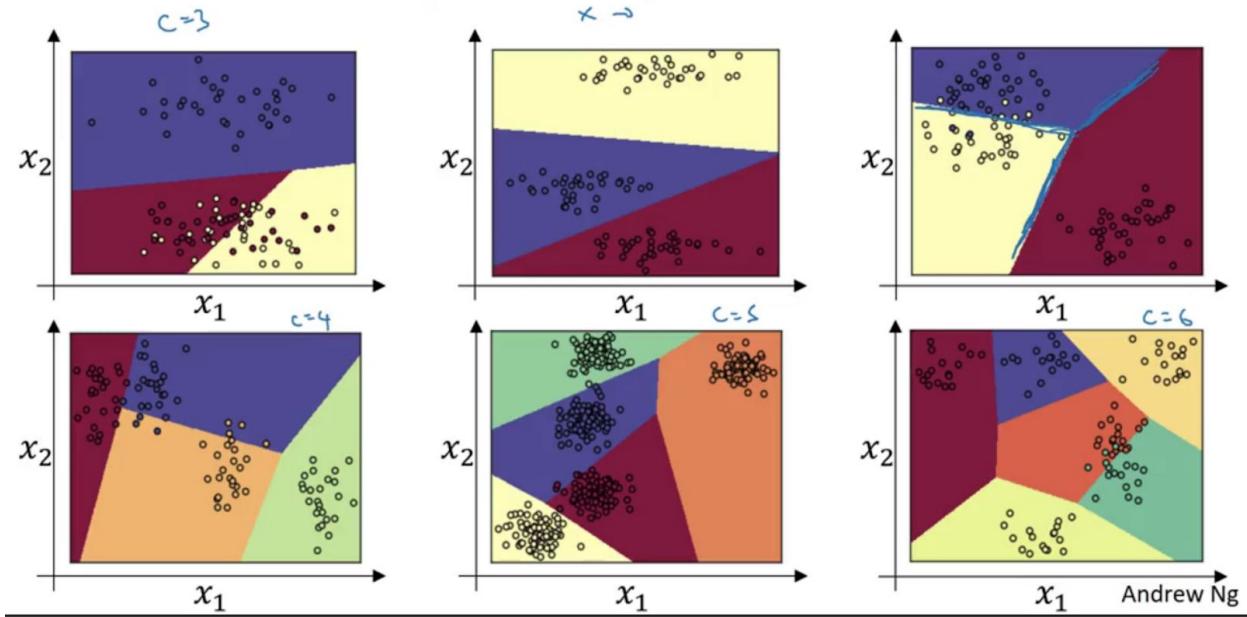
$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \leftarrow$$

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix} \cdot \sum_{j=1}^4 t_j = 176.3$$

$$a^{[L]} = \frac{t}{176.3}$$

$$\hat{y} = \begin{bmatrix} \frac{e^5}{176.3} = 0.842 \\ \frac{e^2}{176.3} = 0.042 \\ \frac{e^{-1}}{176.3} = 0.002 \\ \frac{e^3}{176.3} = 0.114 \end{bmatrix}$$

Softmax examples



Understanding softmax

$$(4,1)$$

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$$

"Soft max"

$$\alpha^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/(e^5 + e^2 + e^{-1} + e^3) \\ e^2/(e^5 + e^2 + e^{-1} + e^3) \\ e^{-1}/(e^5 + e^2 + e^{-1} + e^3) \\ e^3/(e^5 + e^2 + e^{-1} + e^3) \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

"hard max"

$$\alpha^{[L]} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$C = 4 \quad g^{[L]}(\cdot)$

Softmax regression generalizes logistic regression to C classes.

If $C=2$, softmax reduces to logistic regression. $\alpha^{[L]} = \begin{bmatrix} 0.842 \\ 0.158 \end{bmatrix}$

For $C=2$ you need only one number actually which is what binary logistic regression is doing

$1 / 1+e^{-z}$ is actually just $e^z / (e^0 + e^z)$ (assume other one is 0 and the current z is z)

And the other class is just $e^0 / (e^0 + e^z)$ (which is 1-above)

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)

TENSORFLOW BASICS:

```
In [1]: import numpy as np
import tensorflow as tf

In [2]: w = tf.Variable(0,dtype=tf.float32)
cost = tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

0.0

In [3]: session.run(train)
print(session.run(w))

0.1

In [4]: for i in range(1000):
    session.run(train)
print(session.run(w))

4.99999
```

The above is for fixed cost function. What if the cost function was dependent on different kinds of training set? You'll make placeholder and then use feed_dict. Cost would also be defined based on arbitrary x.

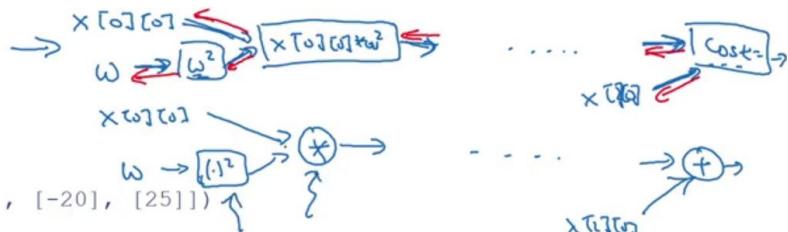
```
In [8]: coefficients = np.array([[1.], [-10.], [25.]])  
  
w = tf.Variable(0,dtype=tf.float32)  
x = tf.placeholder(tf.float32, [3,1])  
#cost = tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)  
#cost = w**2 - 10*w + 25  
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]  
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)  
  
init = tf.global_variables_initializer()  
session = tf.Session()  
session.run(init)  
print(session.run(w))  
  
0.0
```

```
In [9]: session.run(train, feed_dict={x:coefficients})  
print(session.run(w))  
  
0.1
```

```
In [10]: for i in range(1000):  
    session.run(train, feed_dict={x:coefficients})  
print(session.run(w))
```

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)) ←
```

Andrew Ng

