COURSE 2: HYPERPARAMETER TUNING

Bias and Variance

Cat classification

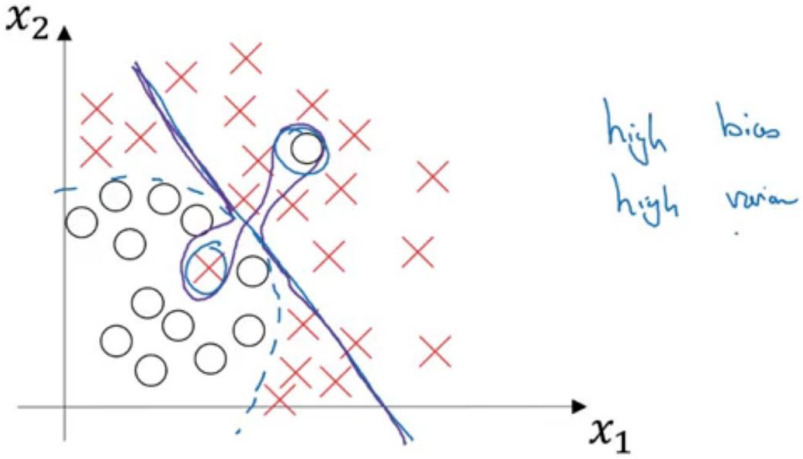

Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high bias & high variance	low bias low variance

Human: ~0%

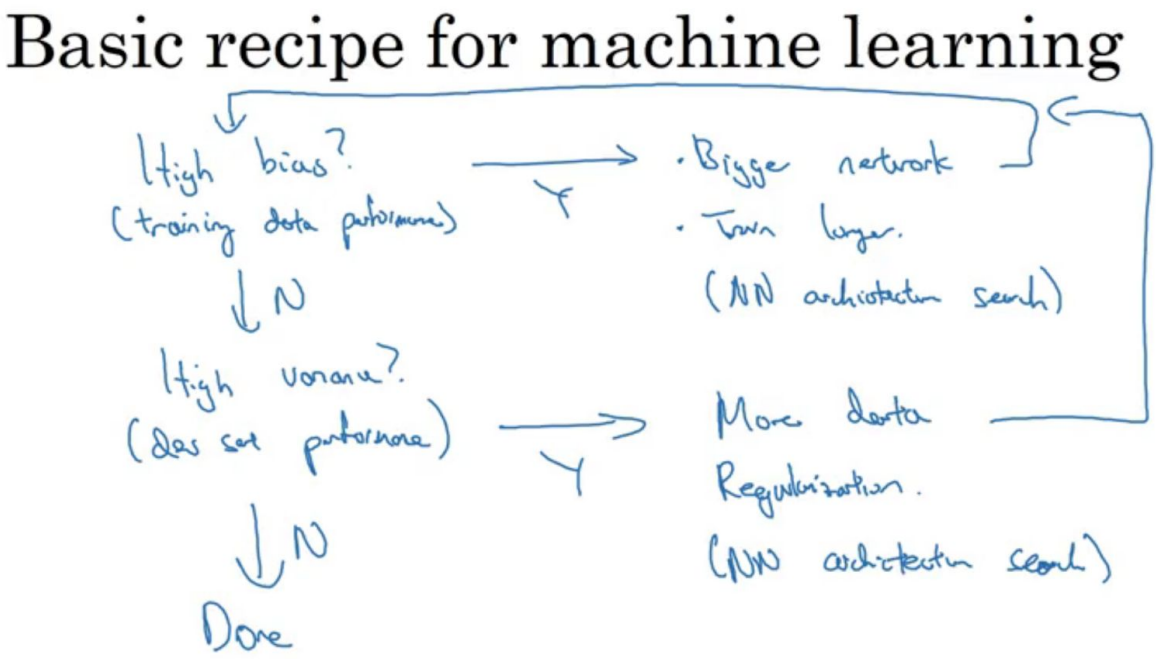
Optimal (Bayes) error: ~~0%~~ 15%

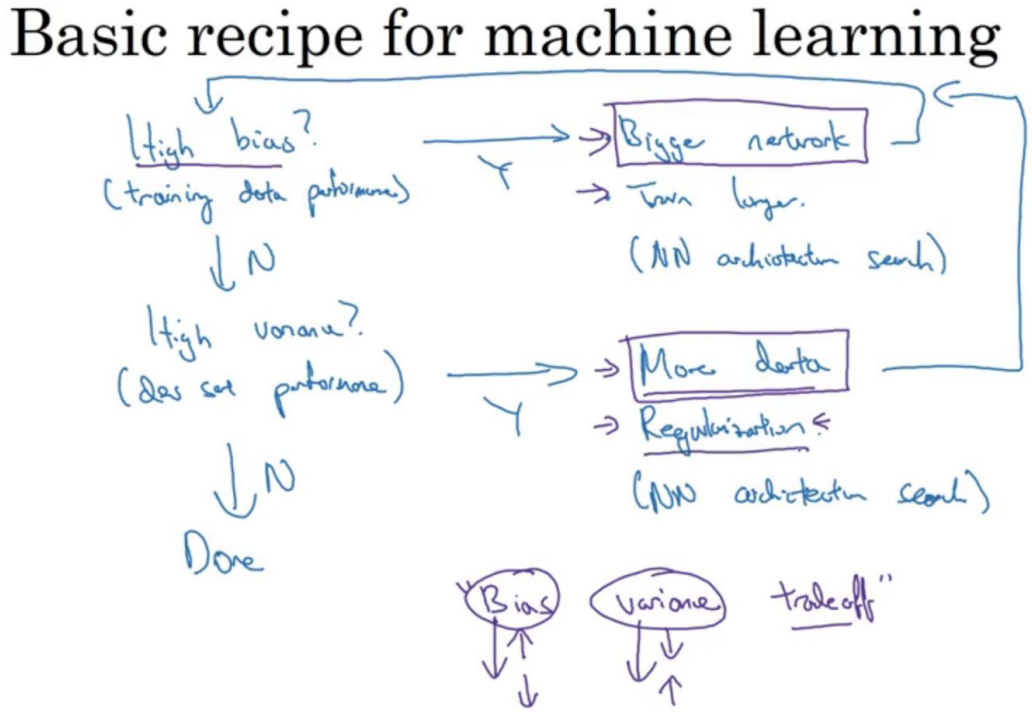
Andrew Ng

High bias and high variance



Mostly linear and sudden curves to overfit at some points





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

Logistic regression

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

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

λ = regularization parameter (lambda, lambd)

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

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

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

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

w will be sparse

L2 norm is also called weight decay

Neural network

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

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

$w: (n^{[l]}, n^{[l-1]})$

"Frobenius norm"

$\|\cdot\|_2^2$ $\quad$ $\|\cdot\|_F^2$

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

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

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

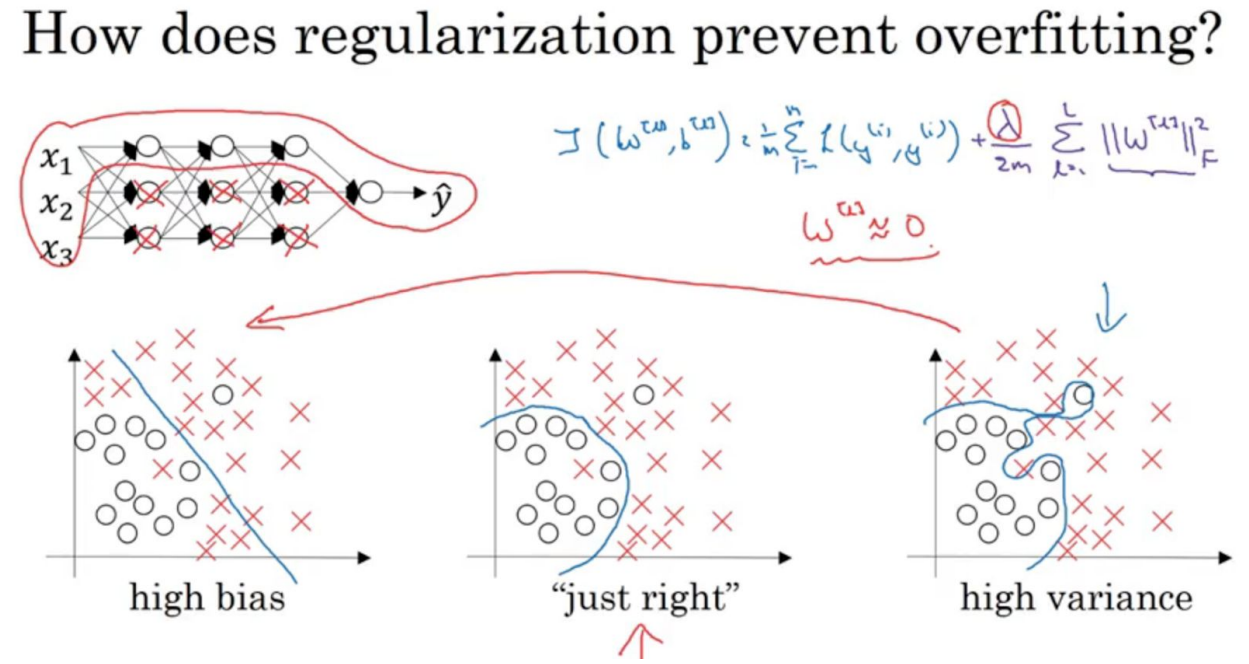
"Weight decay"

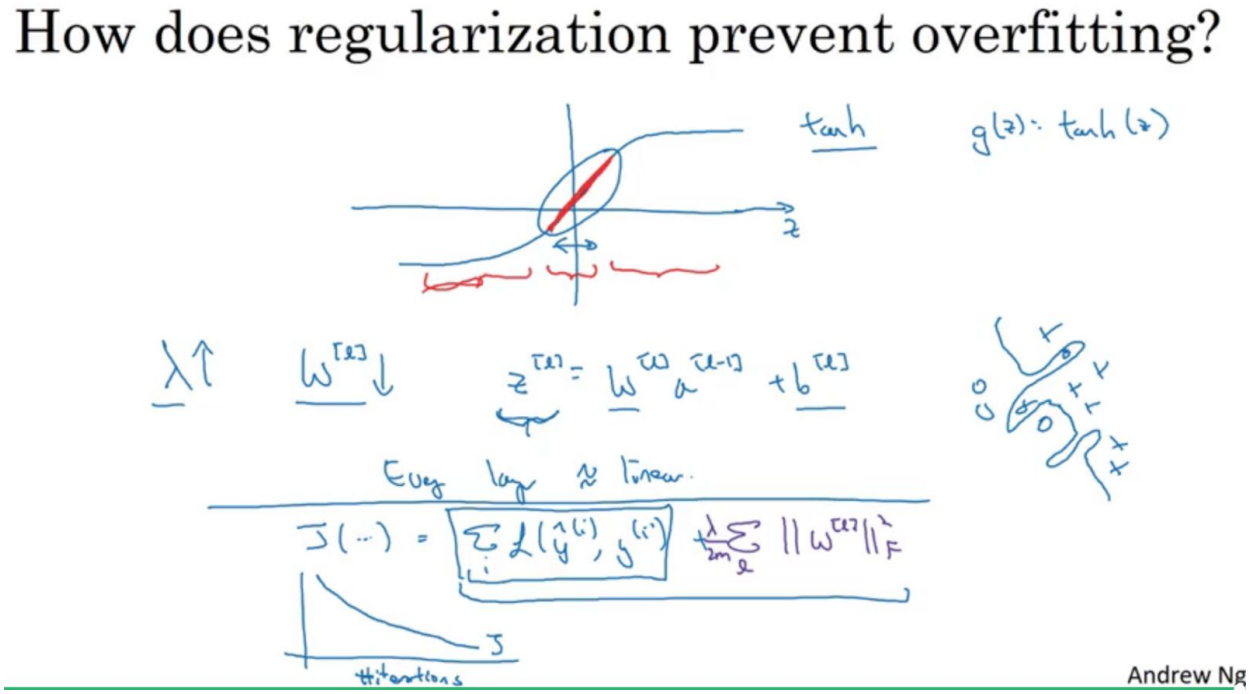
$$w^{[l]} := w^{[l]} - \alpha\left[(\text{from backprop}) + \frac{\lambda}{m}w^{[l]}\right]$$

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

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

Andrew

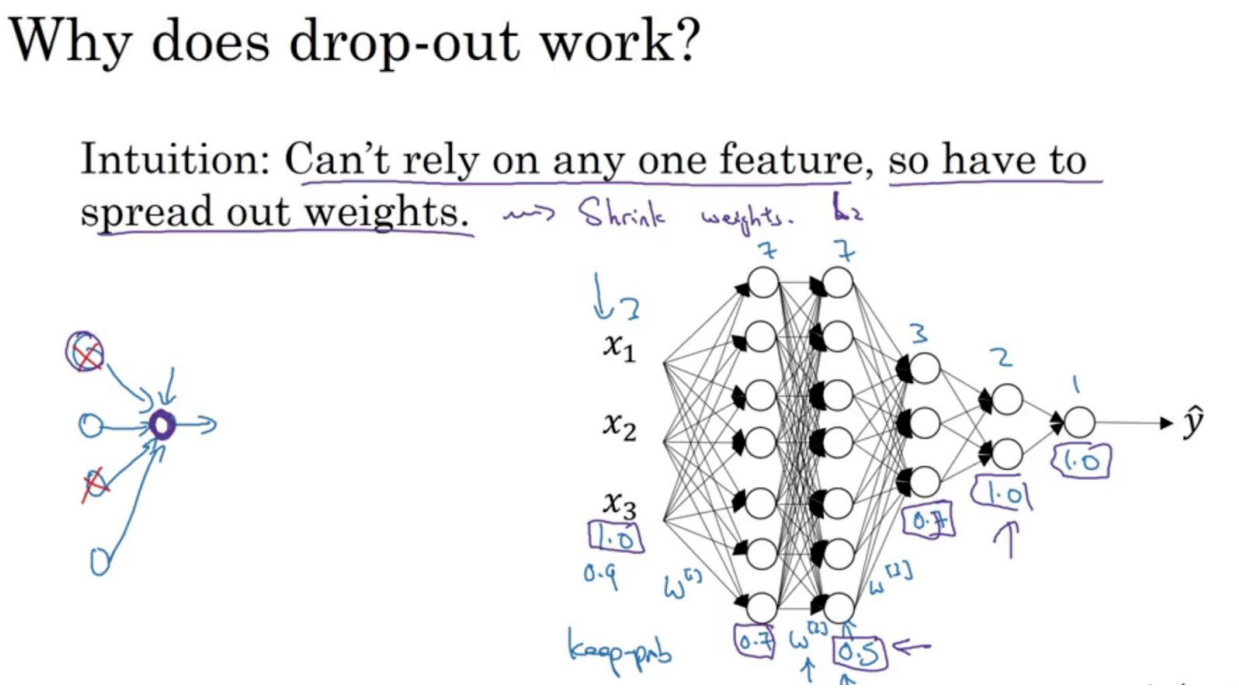


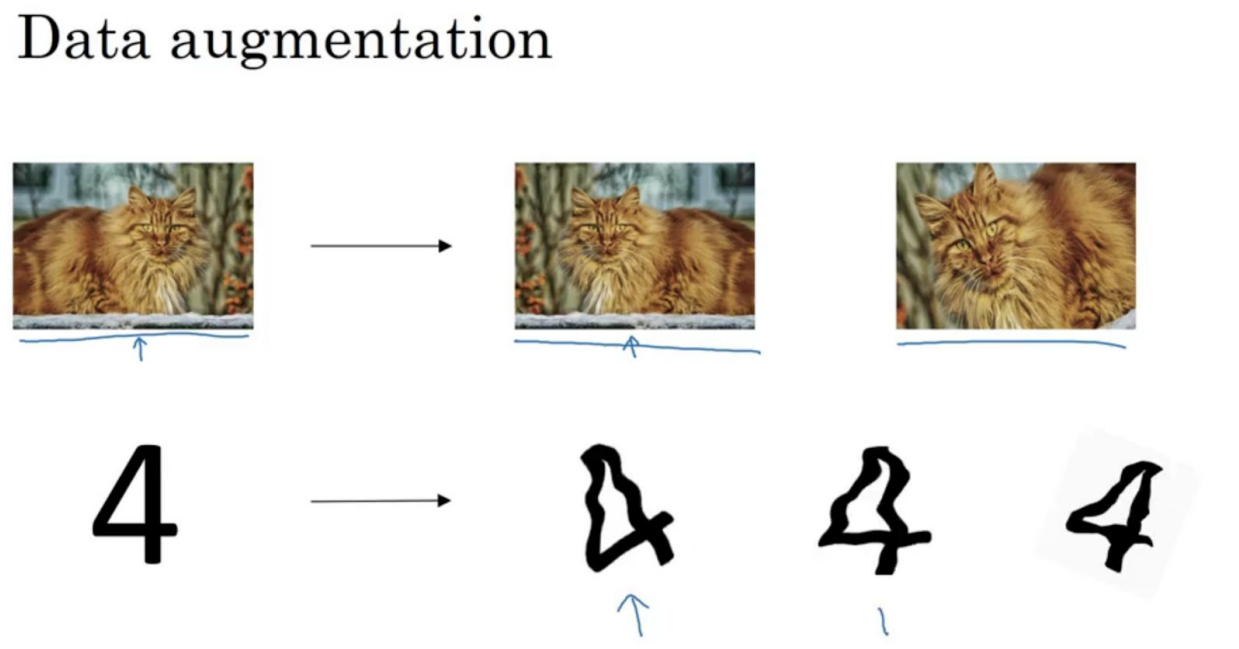


Regularization actually reduces the weights and thus reduces the Z and thus keeps the activation of Z in the linear region, thus preventing VERY complex fitting to data like the one shown above on right.

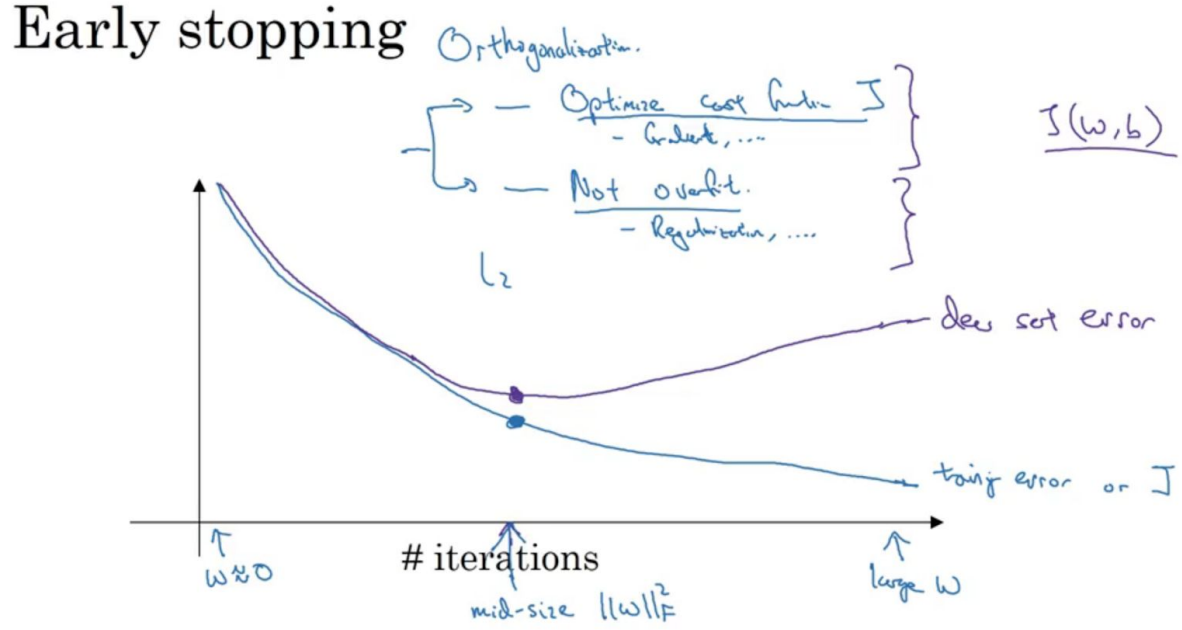
Dropout has the same affect as L2 regularization (Somewhat) -- dropout you are spreading the weights and not put all the bets on one input, in L2 regularization you are penalizing large weights. So the effect is similar.

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.



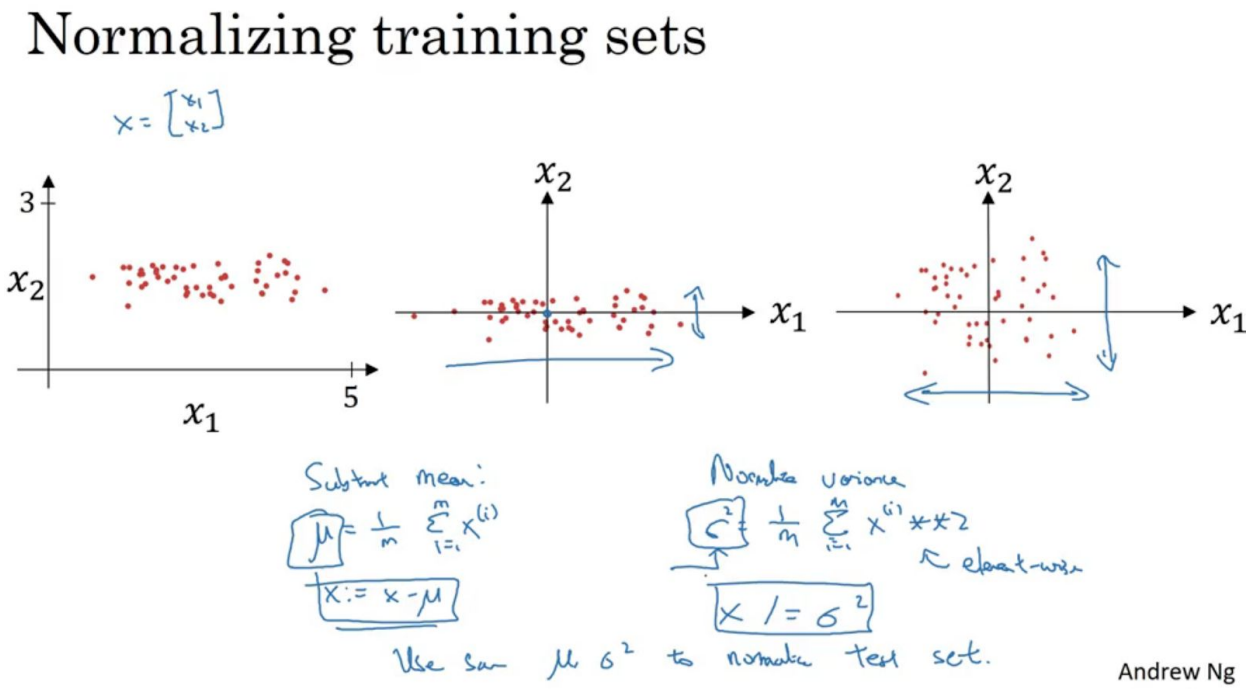


This is not as good as getting more training examples, but it is inexpensive way of doing regularization to the data.



L2 is better for orthogonalization -- separating task for first getting minimum cost and then not overfit. Early stopping kind of combines both at the same thing -- so its little more complex. Disadvantage of L2 is that you need to try out many different lambdas before getting a good one so its more computationally expensive.

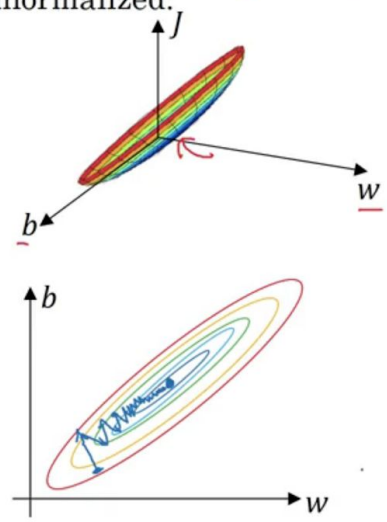
NORMALIZATION:



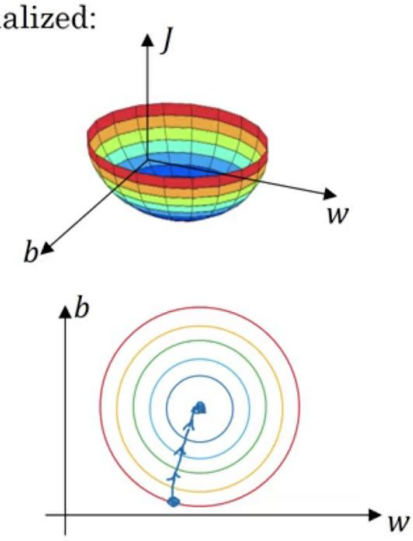
Why normalize inputs?

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

Unnormalized:



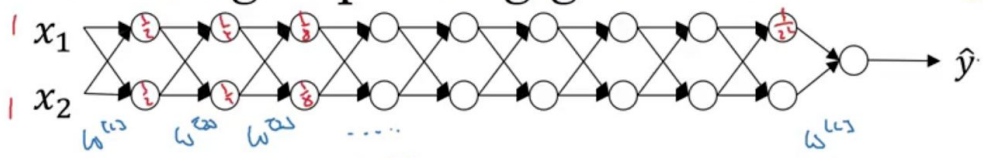
Normalized:



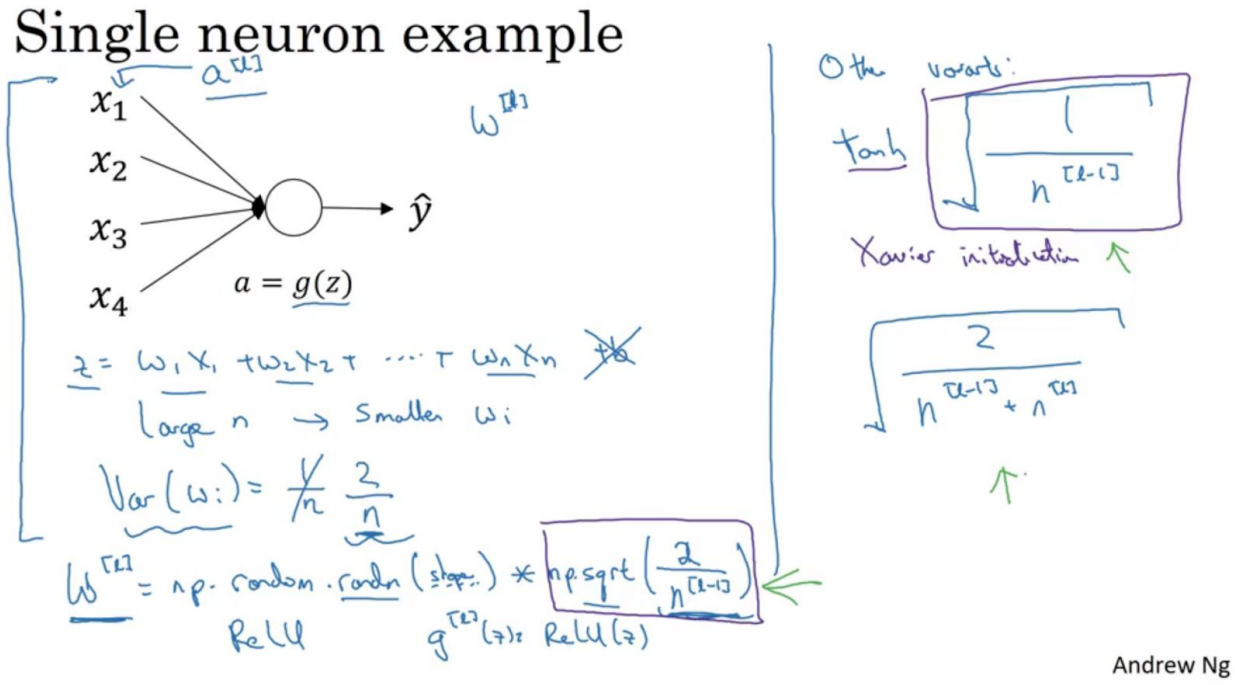
Andrew Ng

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

Vanishing/exploding gradients

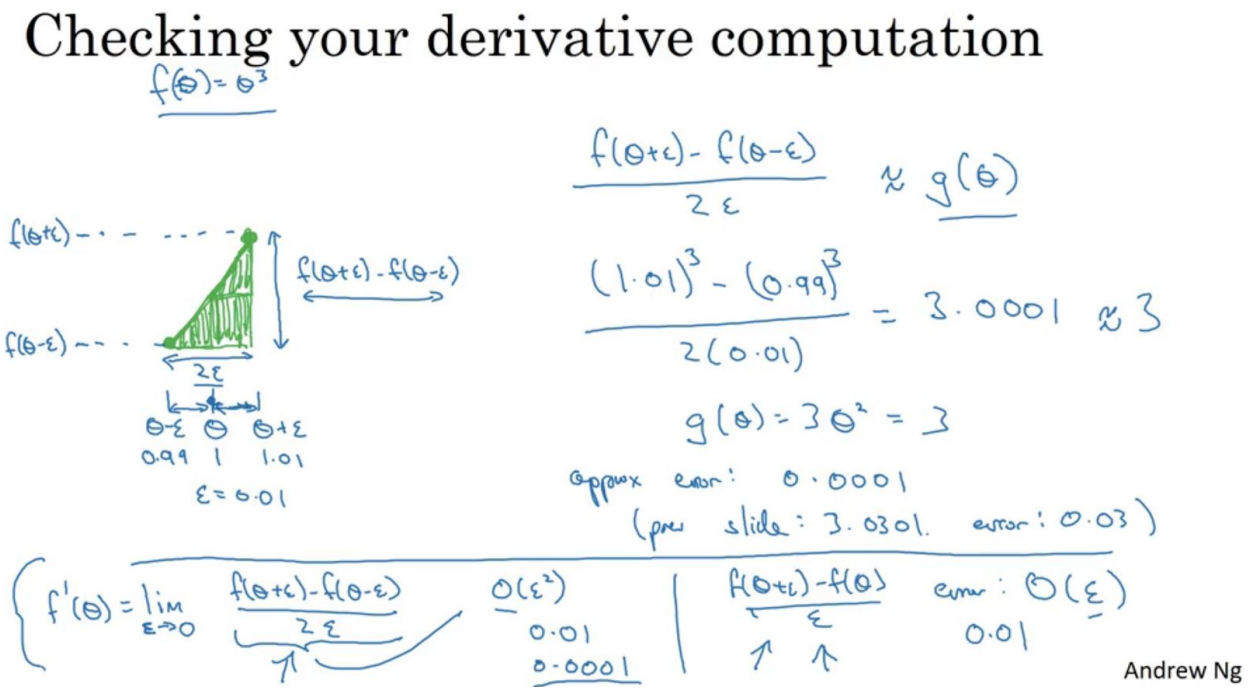


Andrew Ng



Andrew Ng

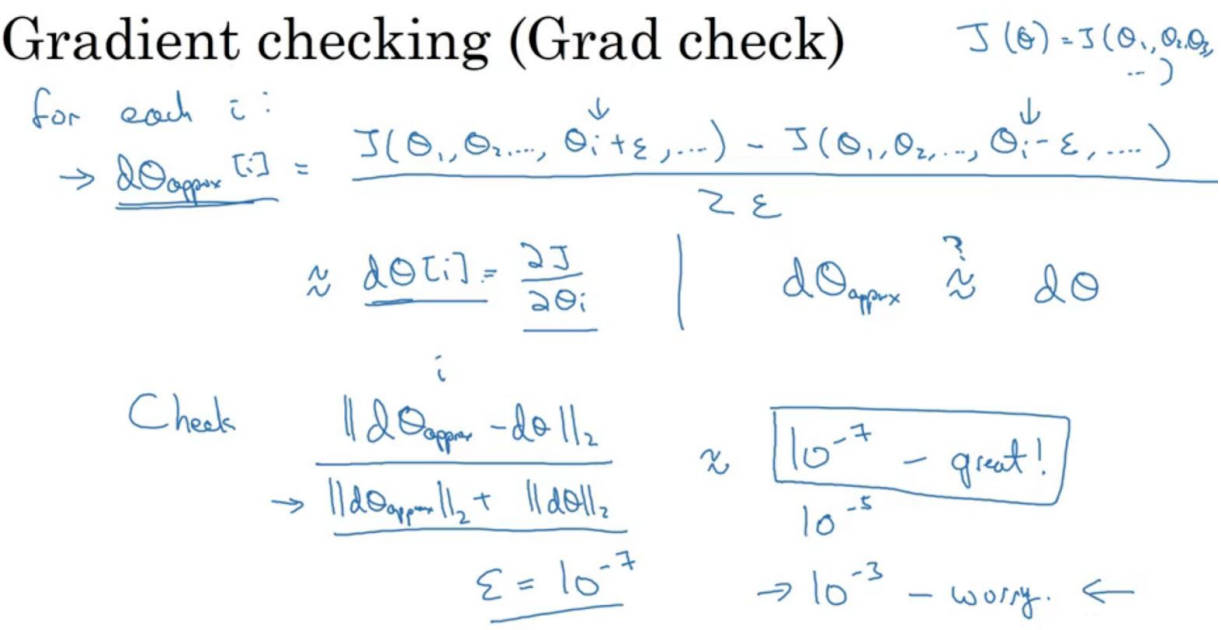
2/Sqrt(n) for relu for random variable variance. Reason is more number of layers, you want weights to be smaller



Andrew Ng

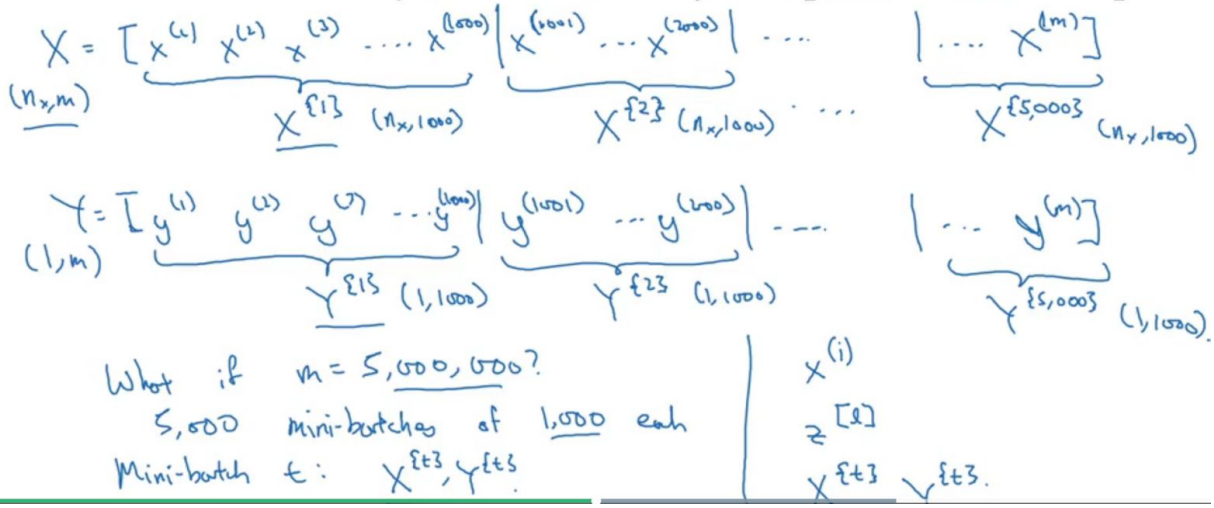
Two sided difference is better for derivative compared to single sided derivative

Gradient checking (Grad check)

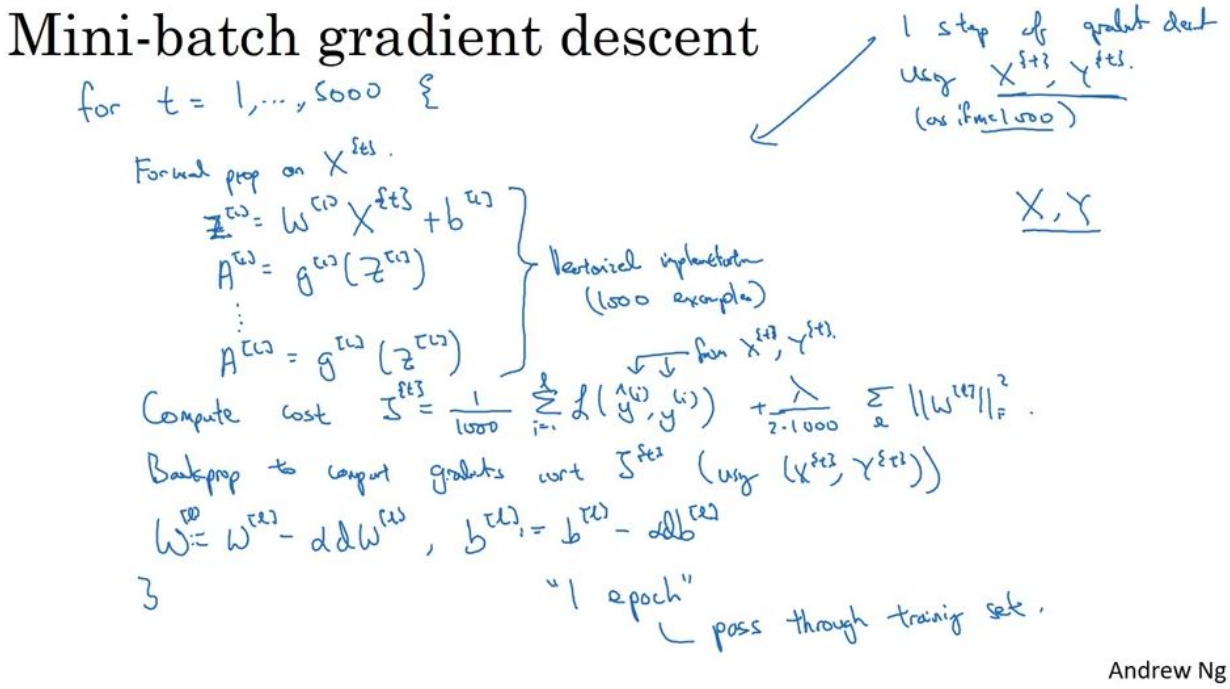


Batch vs. mini-batch gradient descent

Vectorization allows you to efficiently compute on m examples.



X(i) is ith training example
Z[l] is lth layer
X{t} is t mini batch



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. Thats why you need epochs to be absolutely sure.

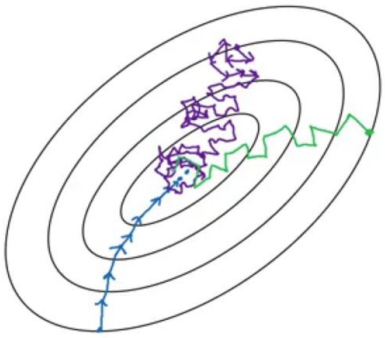
SGD problem-- lose advantage of vectorization!!
MIni batch is best as you can speed up --vectorization as well as time aspect

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

In practice: Somewhere in-between 1 and m



Stochastic gradient descent
- Lose speedup from vectorization

In-between (mini-batch size not too big/small)
- Fastest learning.
- Vectorization. ($\sim$ 1000)
- Make progress without processing entire training set.

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

Andrew Ng

Choosing your mini-batch size

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

Typical mini-batch sizes:

64, 128, 256, 512 1024

2^6 2^7 2^8 2^9 2^{10}

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

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

Exponentially weighted averages

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

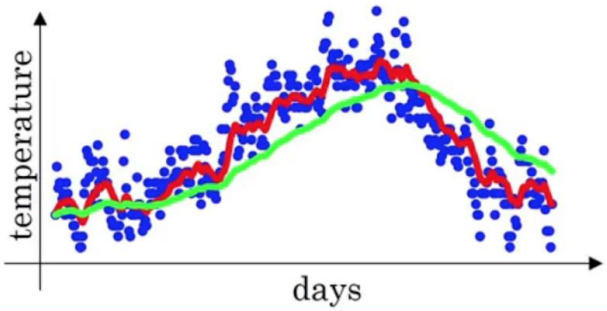
$\beta = 0.9$: $\approx$ 10 days' temperature

$\beta = 0.98$: $\approx$ 50 days

$\beta = 0.5$: $\approx$ 2 days

V_t as approximately average over $\rightarrow \approx \frac{1}{1-\beta}$ days' temperature.

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



To calculate statistical moving average as shown above

Exponentially weighted averages

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

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

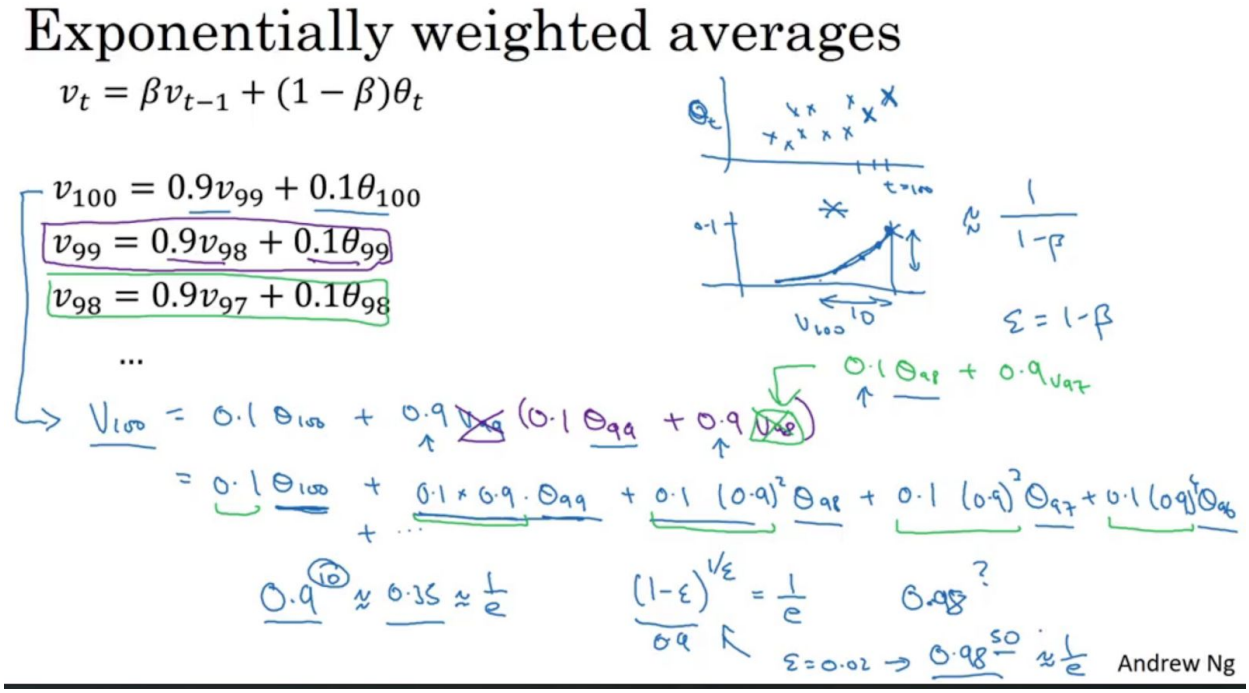
$v_{99} = 0.9v_{98} + 0.1\theta_{99}$

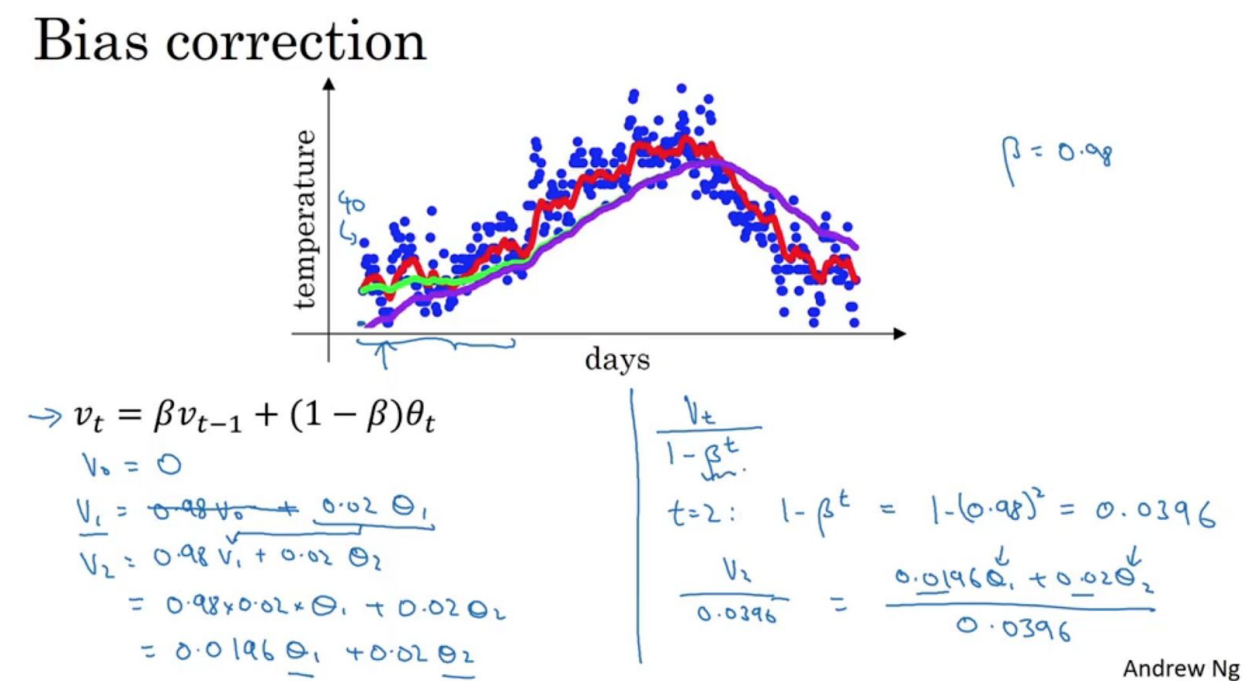
$v_{98} = 0.9v_{97} + 0.1\theta_{98}$

...

$V_{100} = 0.1\,\theta_{100} + 0.9\,(0.1\,\theta_{99} + 0.9\,(0.1\,\theta_{98} + 0.9\,(\ldots)))$

$= 0.1\,\theta_{100} + 0.1 \times 0.9\,\theta_{99} + 0.1\,(0.9)^2\,\theta_{98} + 0.1\,(0.9)^3\,\theta_{97} + 0.1\,(0.9)^4\,\theta_{96} + \ldots$

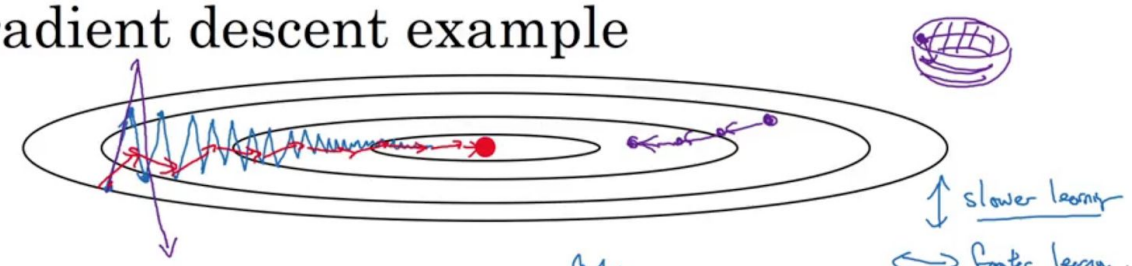




In the beginning slow to warm up, as the previous days are all 0, so you do vt/ 1-b^t = b.vt-1 + (1-b) theta --- reason being that initially the bias correction ratio is small and then gets larger and becomes irrelevant when t gets larger.

MOMENTUM:

Gradient descent example



Momentum:

On iteration t:

Compute dW, db on current mini-batch.

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

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

friction, velocity, acceleration

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

Implementation details

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

On iteration t:

Compute dW, db on the current mini-batch

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

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

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

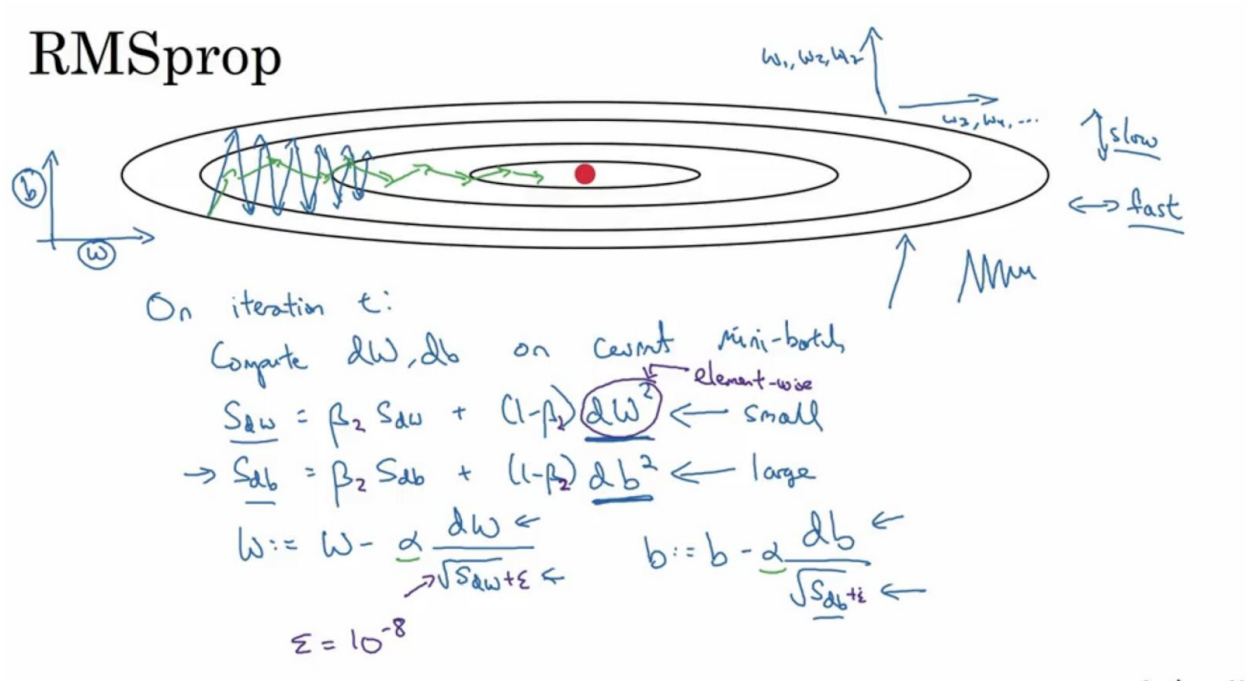
v_{dW}, $1-\beta^t$

Hyperparameters: α, β $\beta = 0.9$

average over last ≈10 gradients

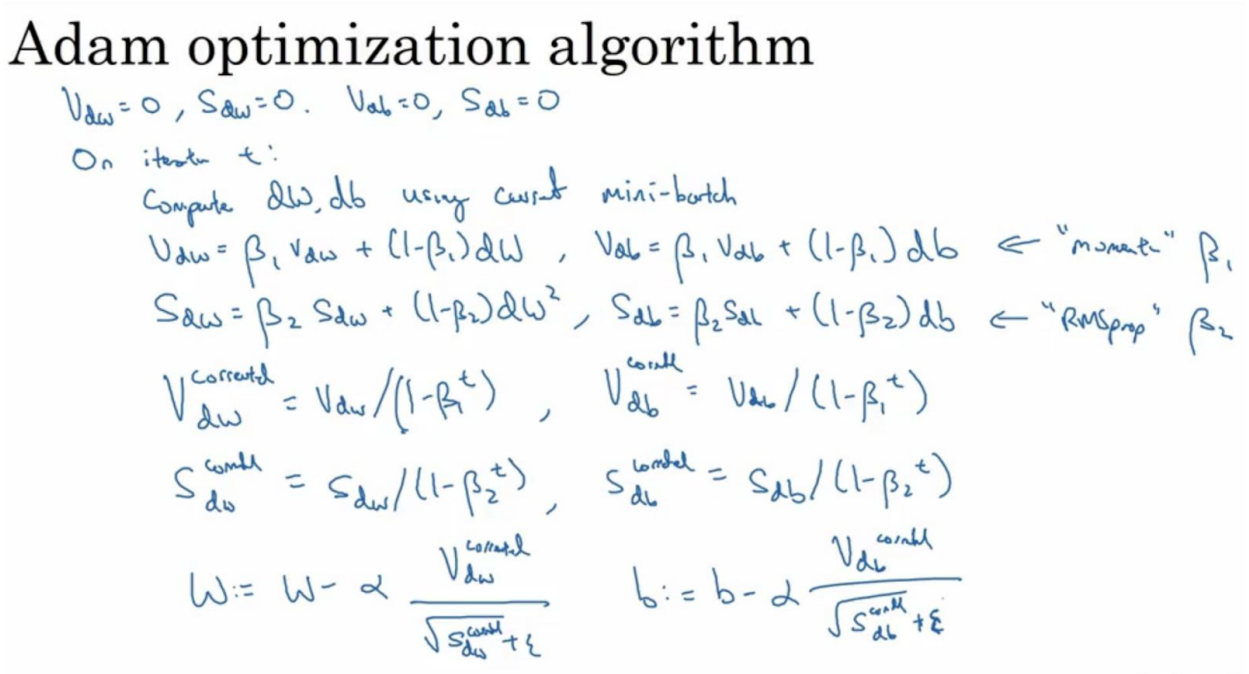
Andrew Ng

Right in purple is in literature but left is more intuitive



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:



Hyperparameters choice:

→ α : needs to be tune

→ β_1 : 0.9 → (dw)

→ β_2 : 0.999 → (dw^2)

→ ε : 10^{-8}

Adam: Adaptive moment estimation

Learning rate decay

1 epoch = 1 pass through data.

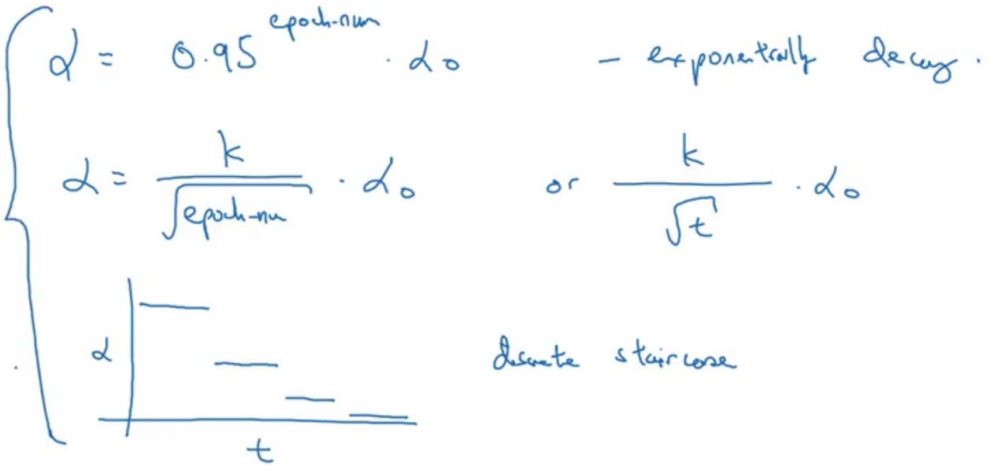
$$\alpha = \frac{1}{1 + \text{decay-rate} * \text{epoch-num}} \alpha_0$$

$\alpha_0 = 0.2$

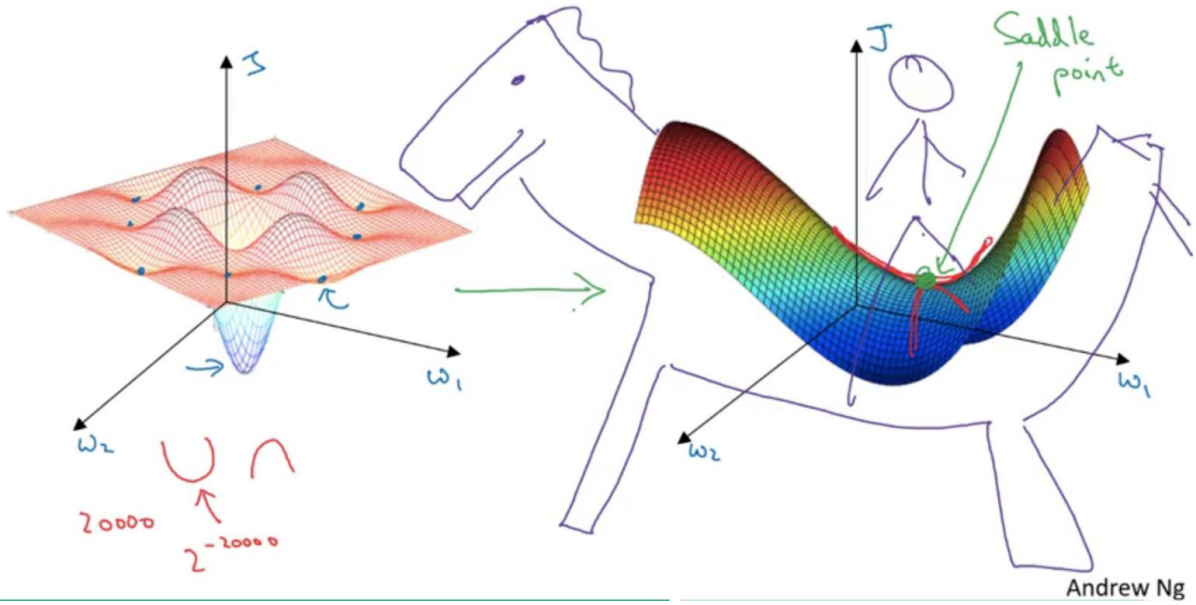
decay-rate = 1

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
⋮	⋮

Other learning rate decay methods

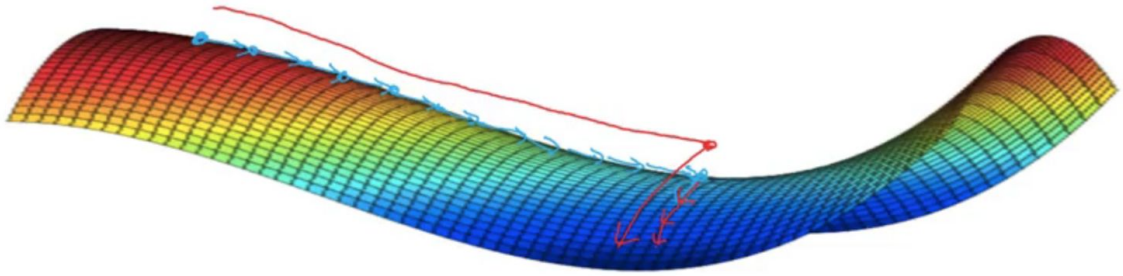


Local optima in neural networks



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

Problem of plateaus

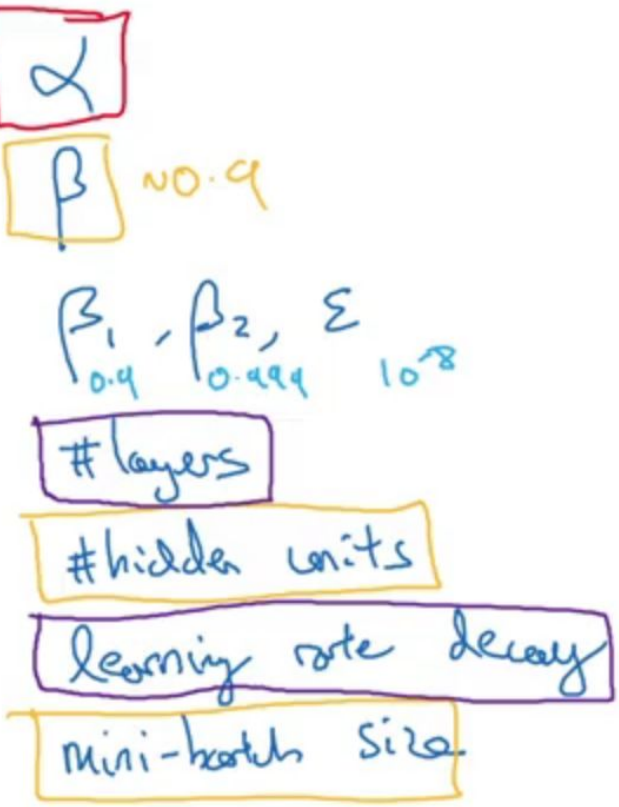


- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow

Adam can help a lot with plateaus!

TUNING THE HYPERPARAMETERS

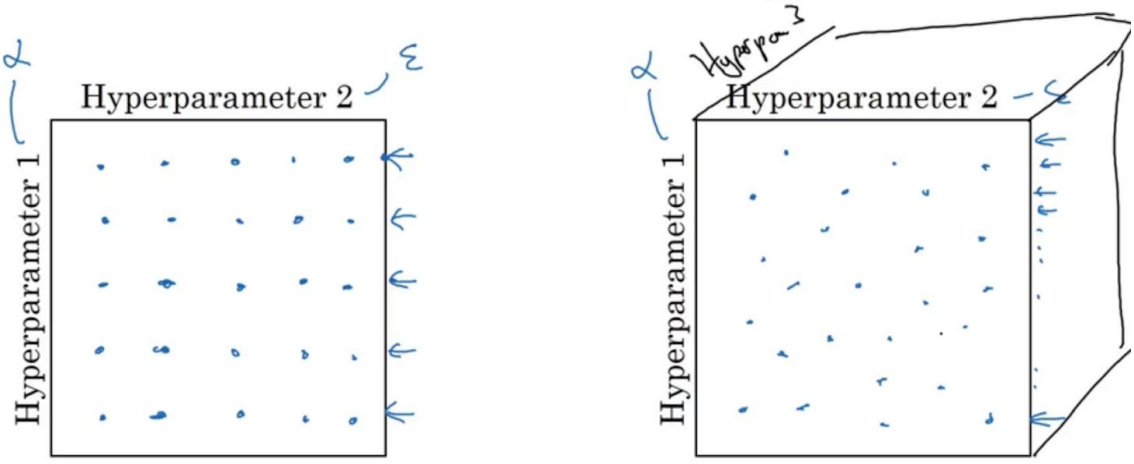
Hyperparameters



RED to ORANGE to PURPLE in terms of importance

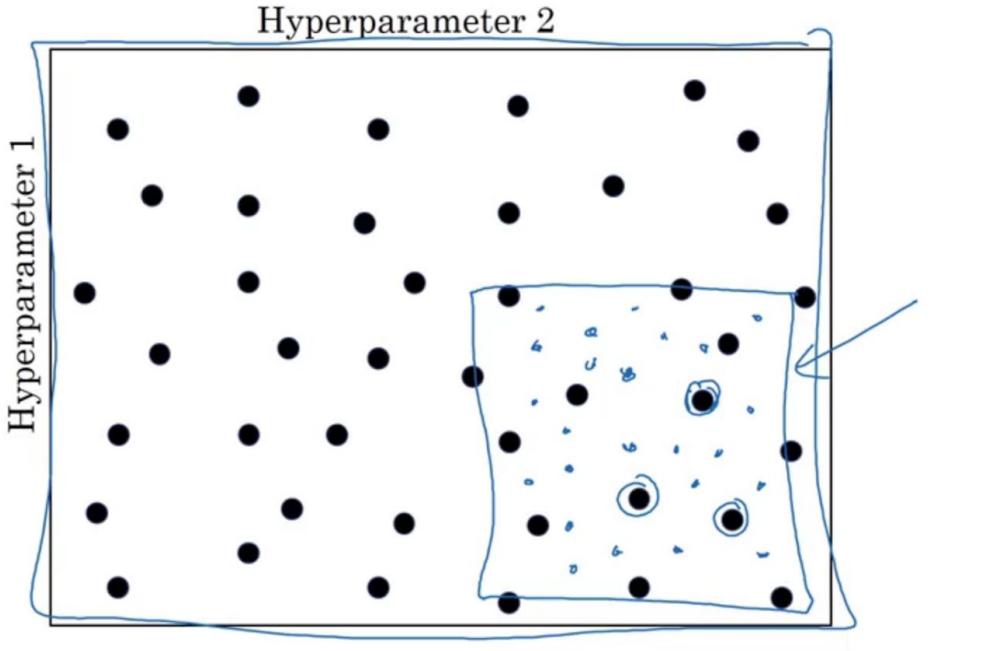
1. Learning Rate
2. Hidden Units, Mini Batch Size, Momentum
3. Learning Rate Decay and No of Layers

Try random values: Don't use a grid

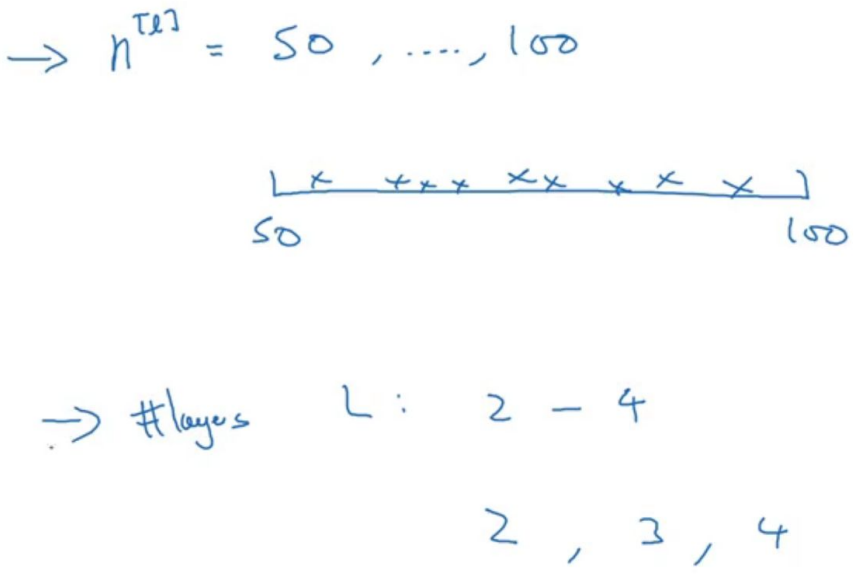


Why random values for hyper-parameters? Because you are trying 25 random values for both 1 and 2 instead of just 5 for Hyper Parameter 1 (if h2 is not much relevant) -- randomness you can try many more values and see how the hyperparameter behaves.

Coarse to fine

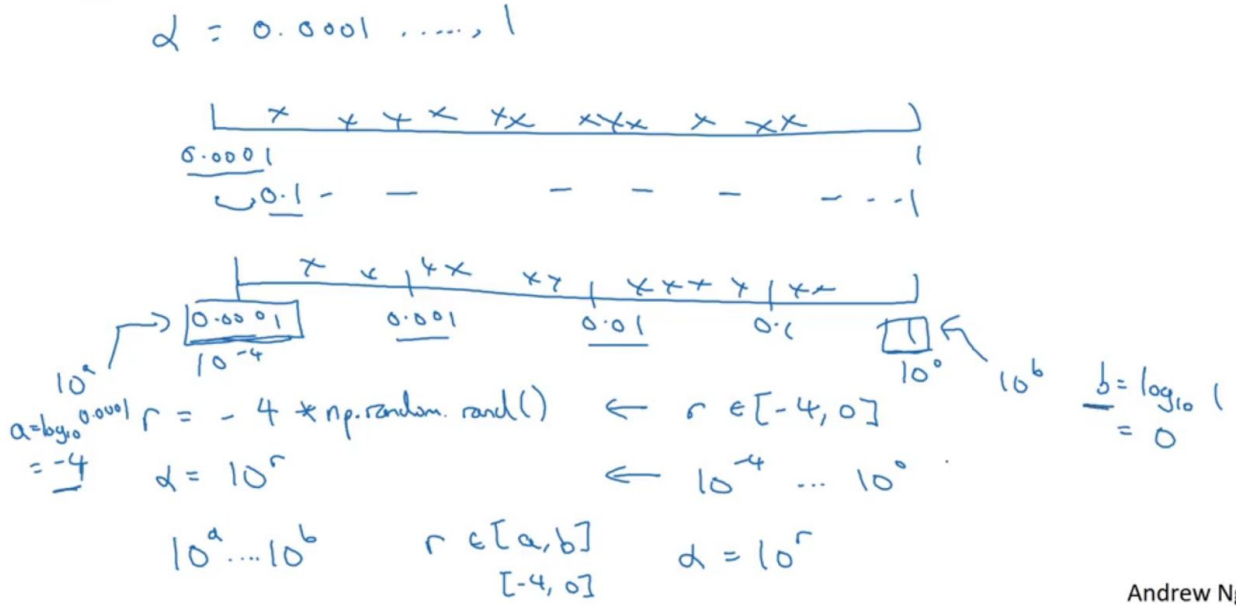


Picking hyperparameters at random



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

Appropriate scale for hyperparameters



Andrew N

Hyperparameters for exponentially weighted averages

$\beta = 0.9 \quad \ldots \quad 0.999$

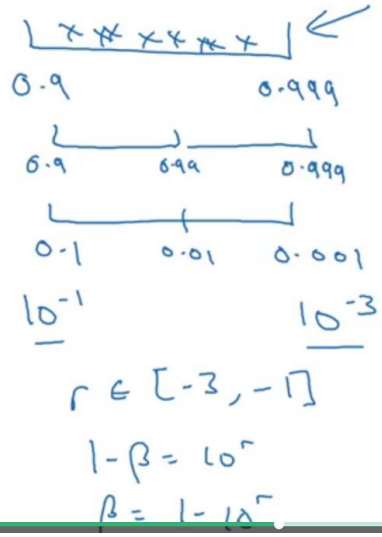
$\downarrow \qquad \qquad \downarrow$

$10 \qquad \qquad 1000$

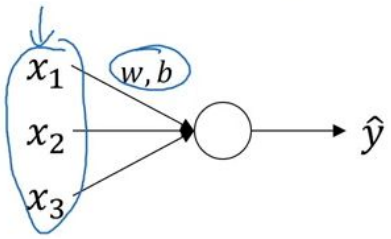
$1-\beta = 0.1 \quad \ldots \quad 0.001$

$\beta: \ 0.9000 \rightarrow 0.9005$

$\beta: \ 0.999 \rightarrow 0.9995$



Normalizing inputs to speed up learning

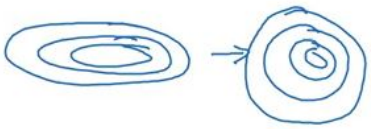


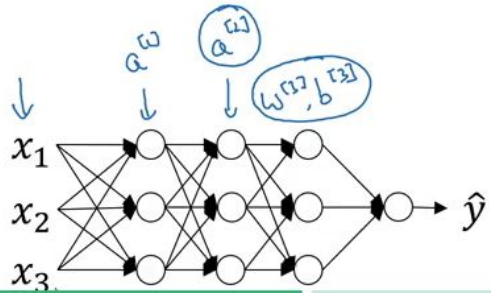
$\mu = \frac{1}{m} \sum_i x^{(i)}$

$X = X - \mu$ ← element-wise

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

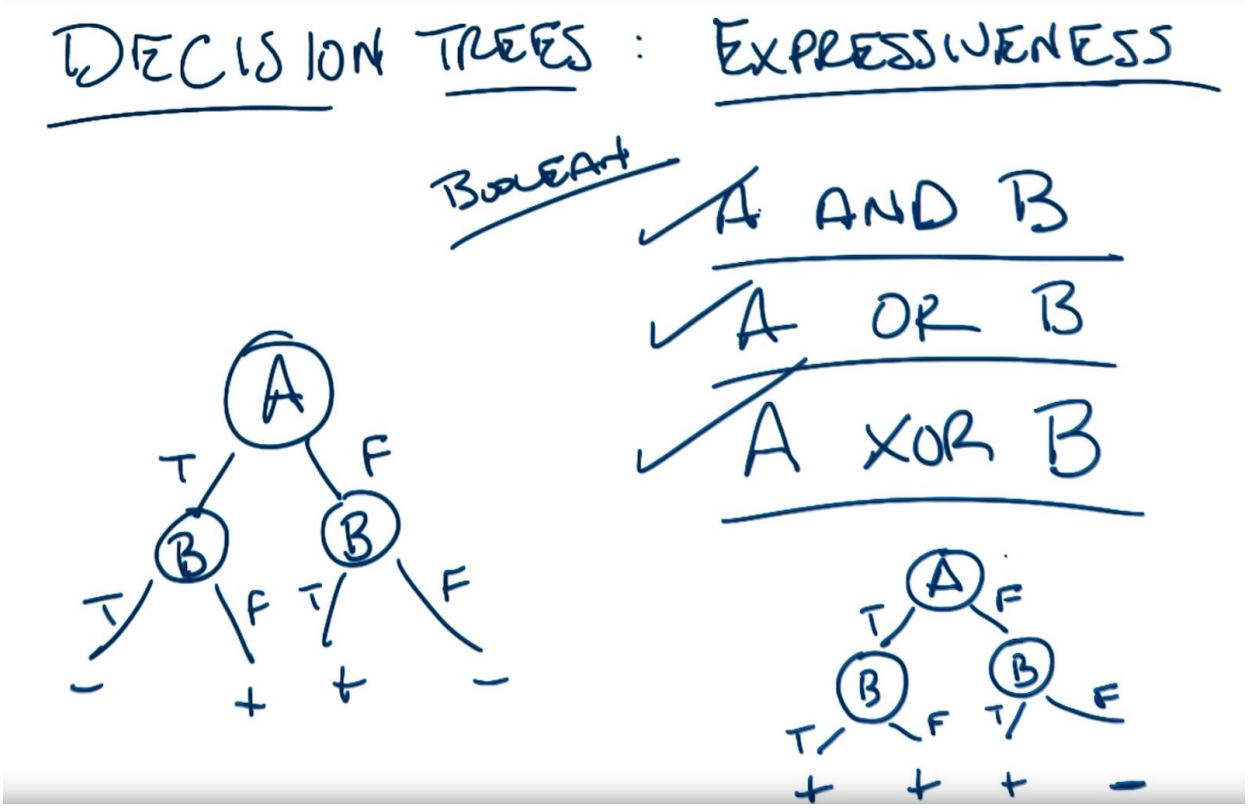
$X = X / \sigma^2$

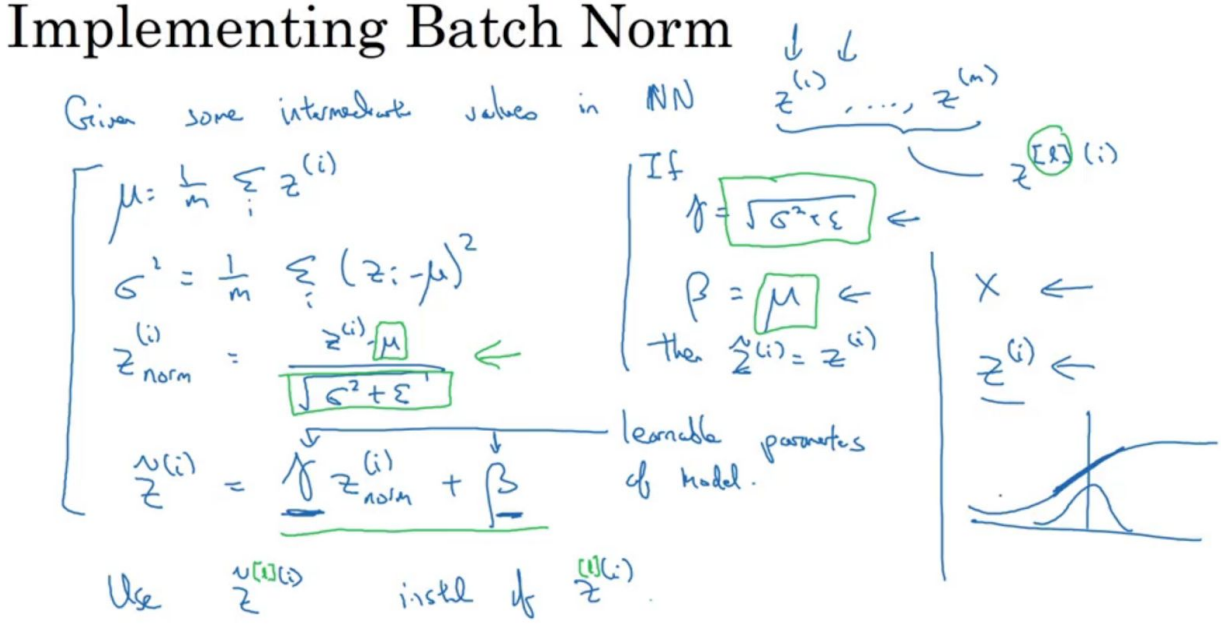




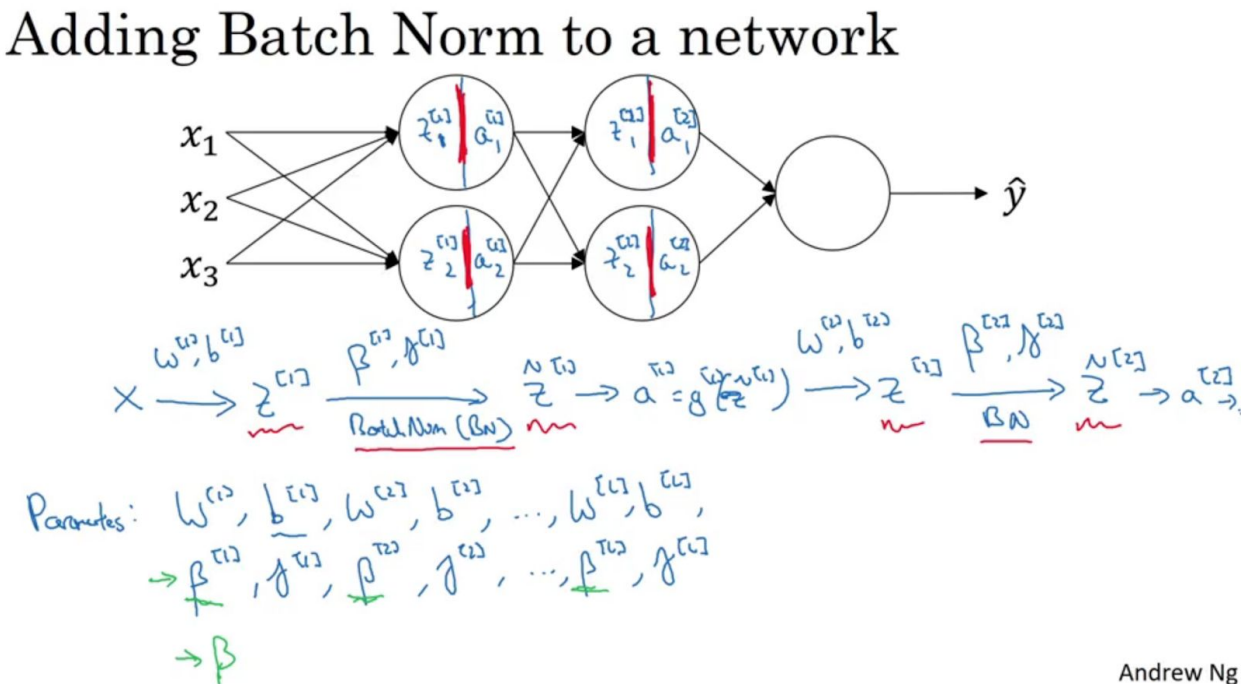
Can we normalize $a^{[2]}$ so as to train $w^{[3]}, b^{[3]}$ faster

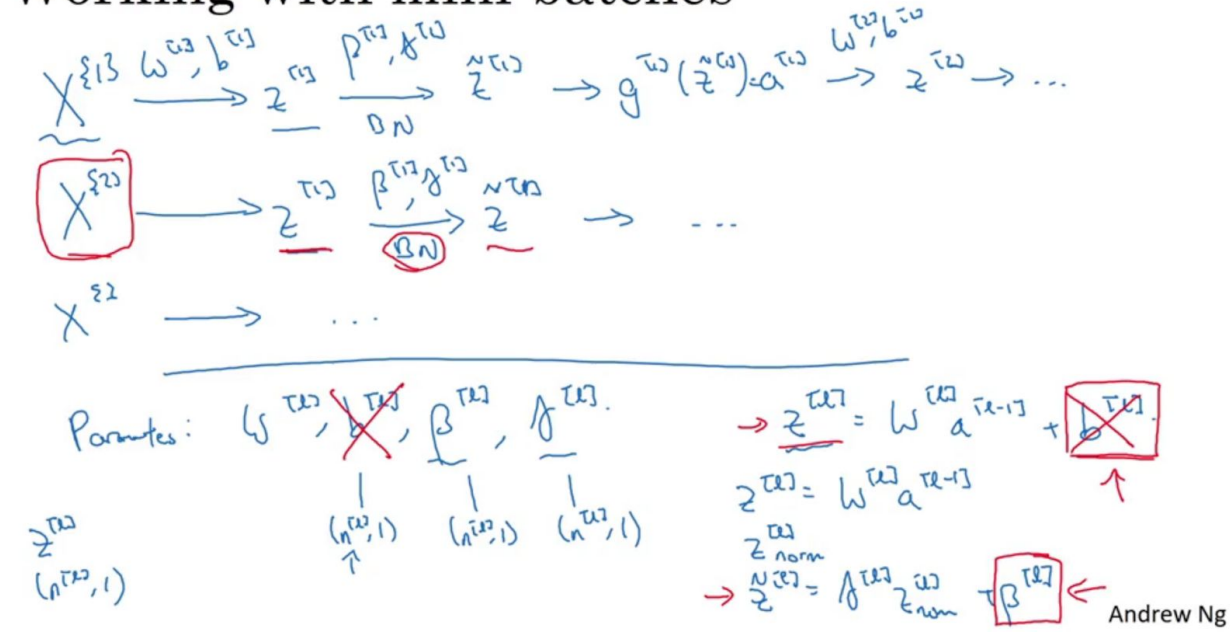
Normalize $z^{[2]}$





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)

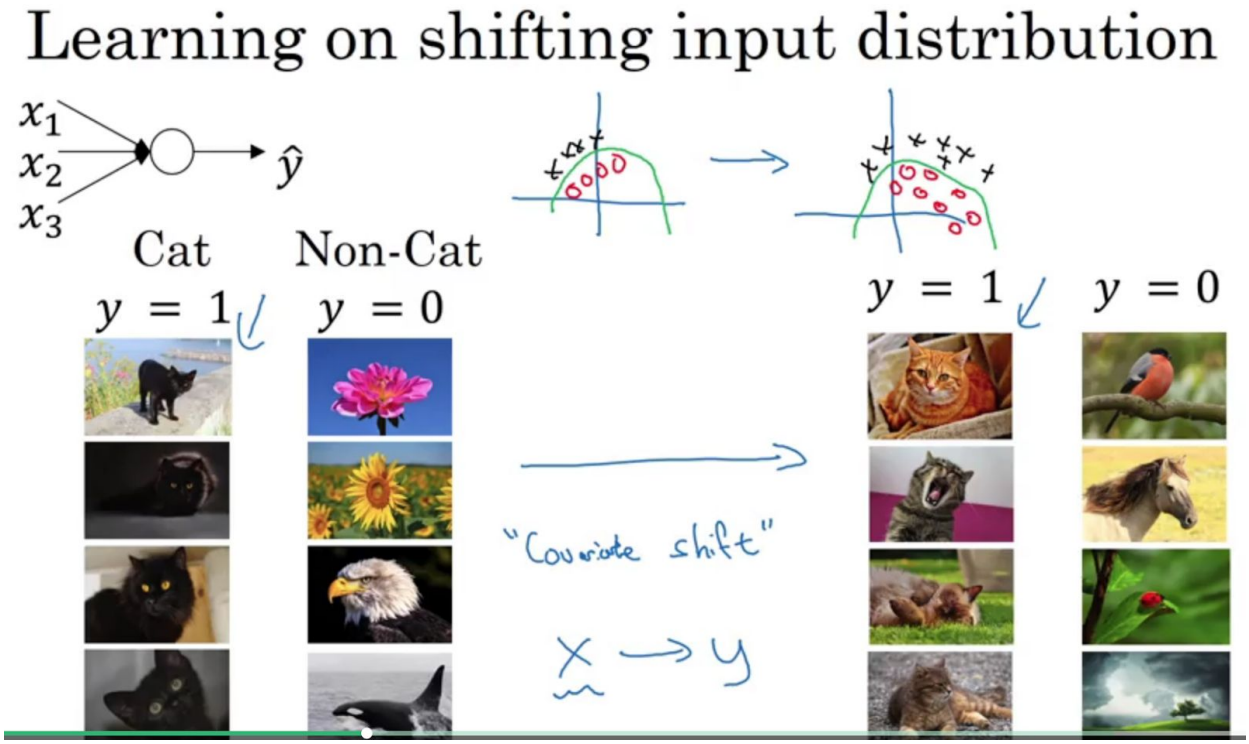




Remove bias terms initially before batch norm as all subtracted during subtraction of mean step, beta terms come in later anyways

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.



Batch Norm as regularization

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- 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.
- 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

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

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

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

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

μ, σ^2: estimate using exponentially weighted average (across mini-batches).

$X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \ldots$

$\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]} \rightarrow \mu$

$\theta_1, \theta_2, \theta_3$

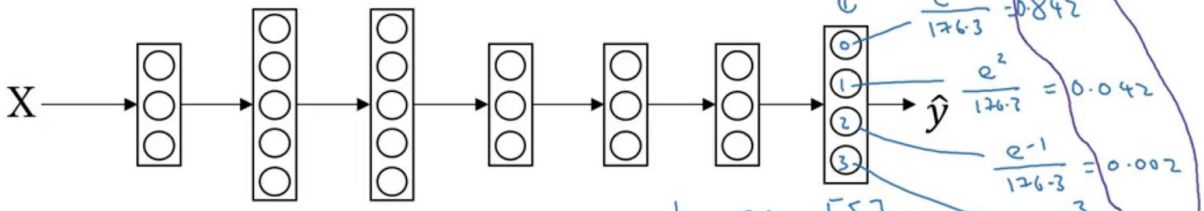
$\sigma^{2\{1\}[l]}, \sigma^{2\{2\}[l]}, \ldots \rightarrow \sigma^2$

$z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}}$ $\quad \tilde{z} = \gamma z_{norm} + \beta$

Andrew Ng

MULTI-CLASS CLASSIFICATION

Softmax layer



$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$ $\quad (4,1)$

Activation function:

$t = e^{(z^{[l]})}$ $\quad (4,1)$

$a^{[l]} = \frac{e^{z^{[l]}}}{\sum_{j=1}^{4} t_i}$, $a_i^{[l]} = \frac{t_i}{\sum_{j=1}^{4} t_i}$

$a^{[l]}$ $(4,1)$ $\rightarrow \hat{y}$

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

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

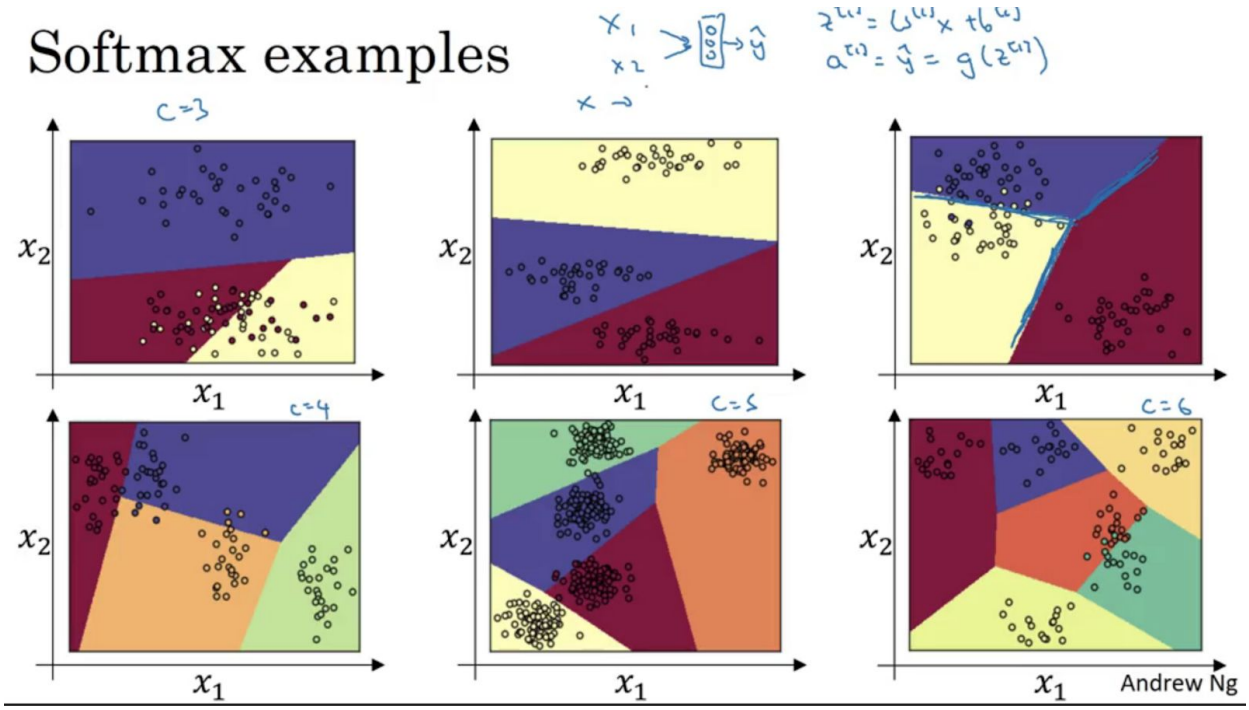
$a^{[l]} = \frac{t}{176.3}$

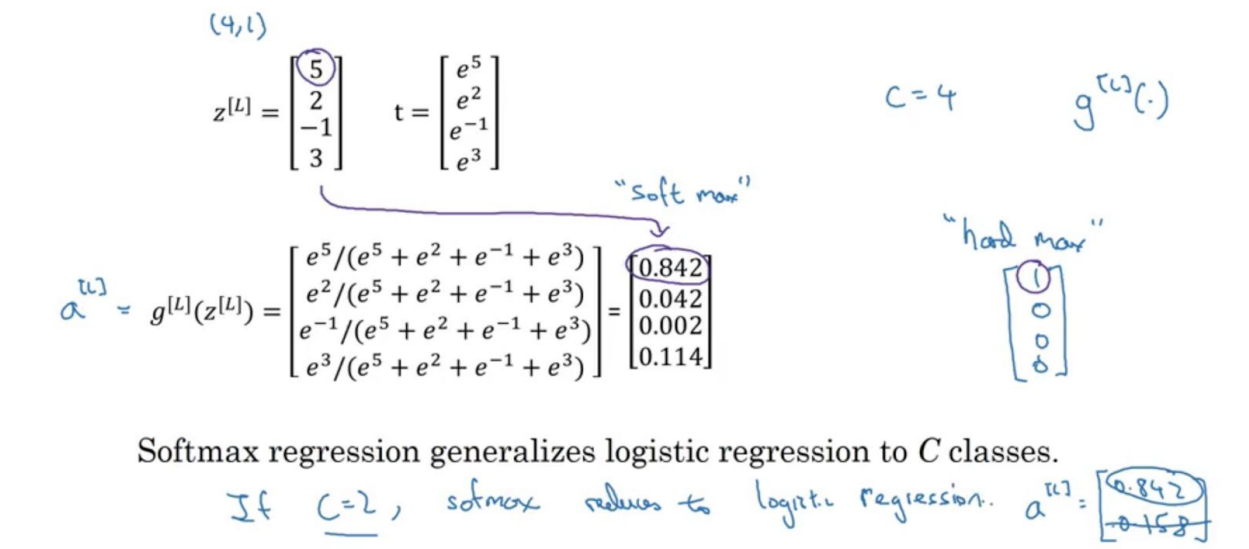
$\frac{e^5}{176.3} = 0.842$

$\frac{e^2}{176.3} = 0.042$

$\frac{e^{-1}}{176.3} = 0.002$

$\frac{e^3}{176.3} = 0.114$





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
```

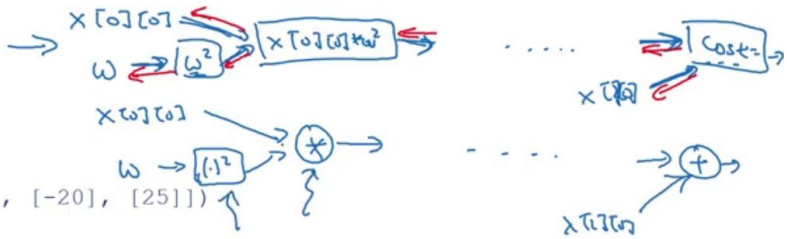
In [4]:
```
for i in range(1000):
    session.run(train)
print(session.run(w))
```

```
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()                        with tf.Session() as session:
session.run(init)                                 session.run(init)
print(session.run(w))                             print(session.run(w))

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

Andrew Ng