Course 1 Neural Networks and Deep Learning:

Week 1

What is a neural network?

Neural Network takes the input features, and we let the neural network decide what the hidden layers should be.

The input layer is densely connected to the hidden layer, every feature is connect to the hidden layer neurons.

Giving Neural network input x and output y, neural networks are really good at finding functions from x to y

Types of Neural Networks

- Standard
- Image Recognition (Convolution Neural Networks)
- Sequence Data (Example Audio, translation, time series) (Recurrent NN)

Structured Data: Like A DataBase (Table)

UnStructured Data: Images, Audio, Text

More Data, Larger Neural Network -> better performance

Small training data, can't tell what size of neural net to use
For larger training data, bigger the size of neural net the better! /
/
Algorithms: (FASTER) better to use a RELU function, stays at zero than a straight line _____/

RELU: h = max(0,a) where a = Wx + b

Reason: One major benefit is the reduced likelihood of the gradient to vanish. This arises when $a>0$. In this regime the gradient has a constant value. In contrast, the gradient of

sigmoids becomes increasingly small as the absolute value of x increases. The constant gradient of ReLUs results in faster learning.

Week 2

Binary Classification (USE Logistic Regression)
To train neural nets the matrix X has each sample data down the column, where the number of columns represent the number of samples in the training set and the number of rows represent the number of features.

So if training set is (x1,y1) (x2,y2) .. (xn,yn), where each xi is a vector of features for the input and yi is the result (0,1)

X = [x1 x2 … xn]

Y = [y1 y2 … yn] ← result matrix

Error function the function to get loss between what the estimated probability and the true result. (Estimated probability is y hat) (Remember y can only be 0 or 1)(its a cat or not a cat)

- should only have one minimum so gradient descent doesn't get stuck on a local minimum, and should be convex.
- Logistic regression (loss) error function is L(y^, y) = -(ylog(y^) + (1-y)log(1 - y^))
- The equation explains that the closer y^ is too y the smaller the error function.
- Remember this done on one sample, a specific training example.

The cost function is just the sum of the loss(error) function values over the number of samples in the training set J(w,b) = 1/m (Sigma(i = 1 to m) (L(y^i, yi))

- Basically trying to find w and b to minimize the cost function

Gradient Descent (Take steps in the steepest path downward, by subtracting the slope)

- Fill parameters w and b with random values, (for logistic regression usually 0) and gradient descent, figures out the parameter values for where the cost function is minimized.
- Algorithm: Repeat { w = w - alpha(derivative of the cost function respect to w)
- Where alpha is the learning rate

Computation Graph

Basically the cost function is made up of many variables
forward propagation step is just one operation in the calculation of the cost function. (Like a binary tree of operations and going down one node)
Backward propagation is used to find the derivatives respect to one of the variables in the cost function. (Chain Rule) (Need this derivative for the gradient descent algorithm)

Logistic Regression Gradient Descent

To get the derivative value in the gradient descent formula:
Backpropagation, we get

dL/dz = a-y where L is the loss functions and z = Wx + b

To get the derivatives for all parameters w1 … wn since z = w1x1 + … + wnxn + b

dL/dwi = (dL/dz)xi which is just (a - y)xi and dL/db = dL/dz = a - y

So the gradient descent algorithm would just be:

(let dwi represent dL/dwi since we always finding derivatives regarding the loss function

w1 := w1 - alpha(dw1)
w2 := w2 - alpha(dw2)
‘
‘ where alpha is just the learning rate
‘
wn := wn - alpha(dwn)
b := b - alpha(db)

SUMMARY OF GRADIENT DESCENT ALGORITHM for LOGISTIC REGRESSION

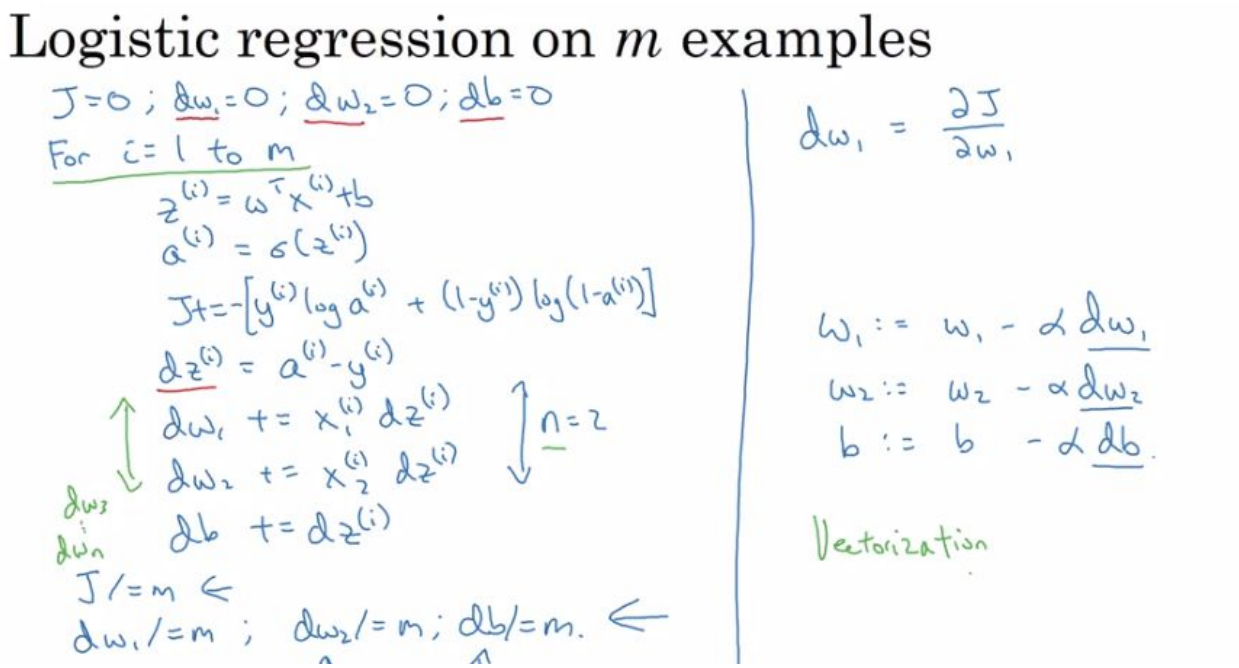
For the cost function, since we have m samples in training set, just need to sum up all the derivative values and divide by m

```
For i = 1 to m
          zi  = W^T(xi) + b
          ai  = sigma(zi)

         J += yi(log(ai)) + (1 - yi)
         dzi = ai - yi
         dw1 += (x1i)(dzi)
         .
         .
        dwn += (xni)(dzi)
        db += (dzi)
Done
J /= m
dw1 /= m;  . . . dwn /= m;  db /= m
```

Then just adjust all wi = wi - alpha(dwi)

THAT IS ONE STEP IN GRADIENT DESCENT

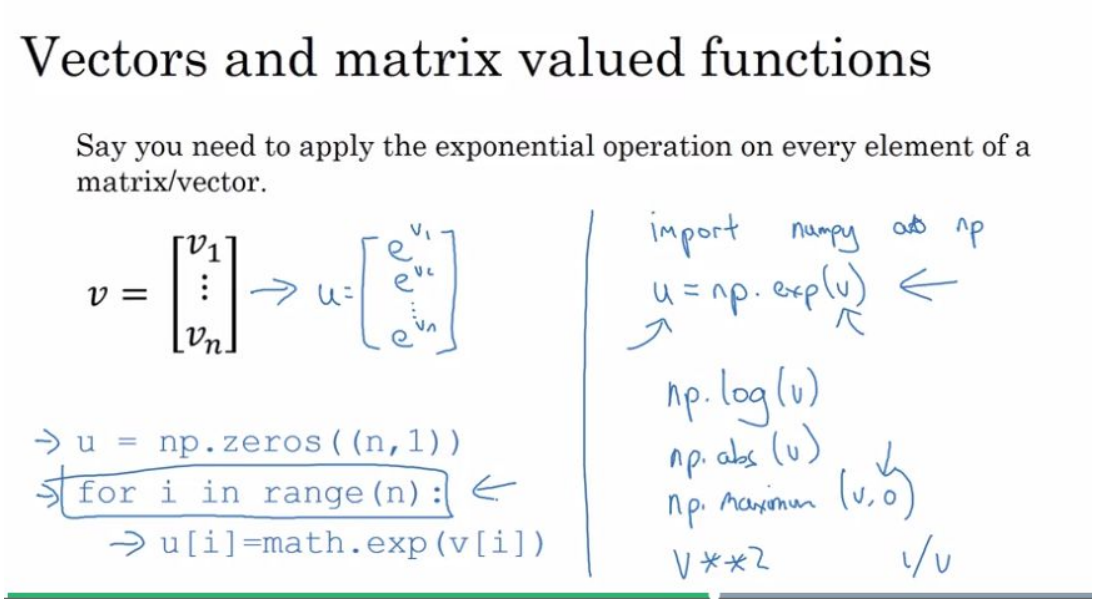


Vectorization

Basically getting rid of for loops for faster execution

Basically to do a dot product, (Example Av, or AB)

Numpy:
Every Time You need a for loop, see if there is a numpy build in function to the same thing .



Basically, can cut for loops by making a vector and then doing vectorized

Vectorization on Logistic Regression

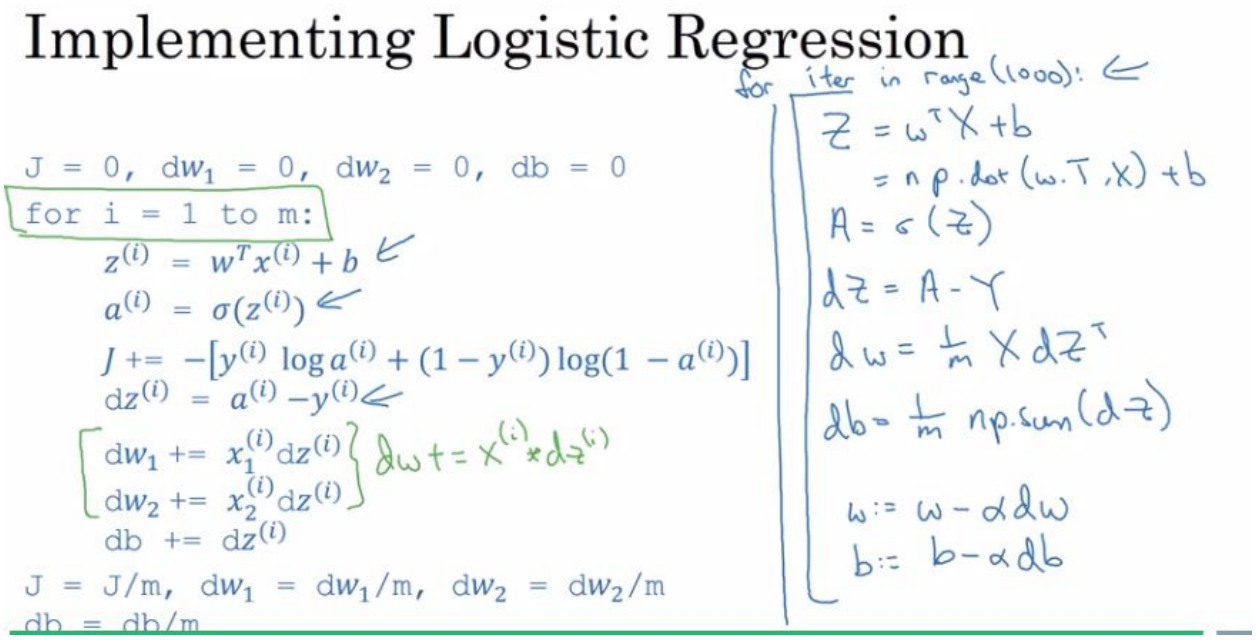
Broadcasting -> when a 1 x 1 matrix like b acts like a 1 x m matrix when doing …

Z = np.dot(w.T, X) + b

Vectorized Implementation of Gradient Descent

1) Z = np.dot(w.T, X) + b
2) A = sigmoi(Z)
3) dZ = A - Y <<--- dZ is a 1 x m matrix for the derivative with respect to z for each Training example
4) db = 1/m (np.sum(dZ)
5) dw = 1/m (np.dot(X,dZ.T)) <<-- shown above in a for loop, dw is a 1 x m matrix for all optimized parameters

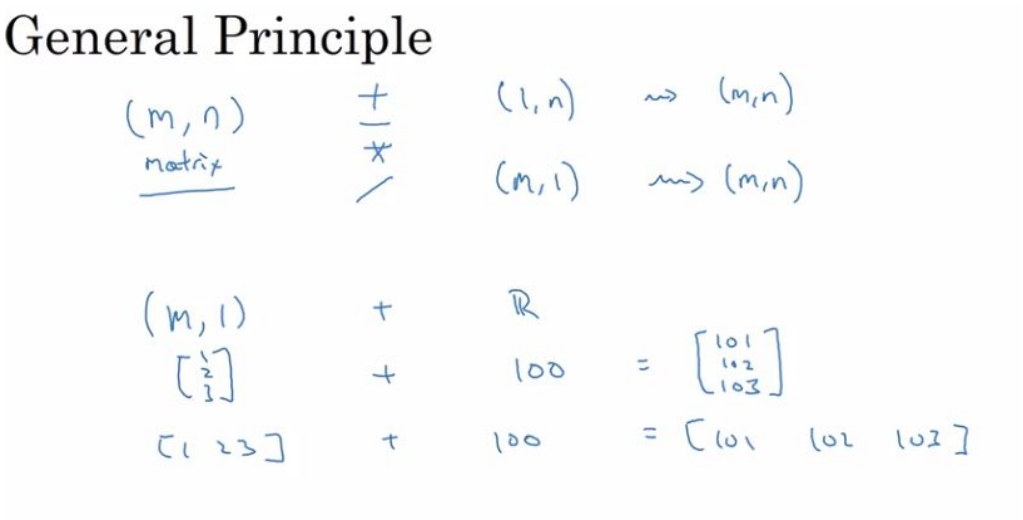
SUMMARY OF ONE STEP OF GRADIENT DESCENT



Broadcasting

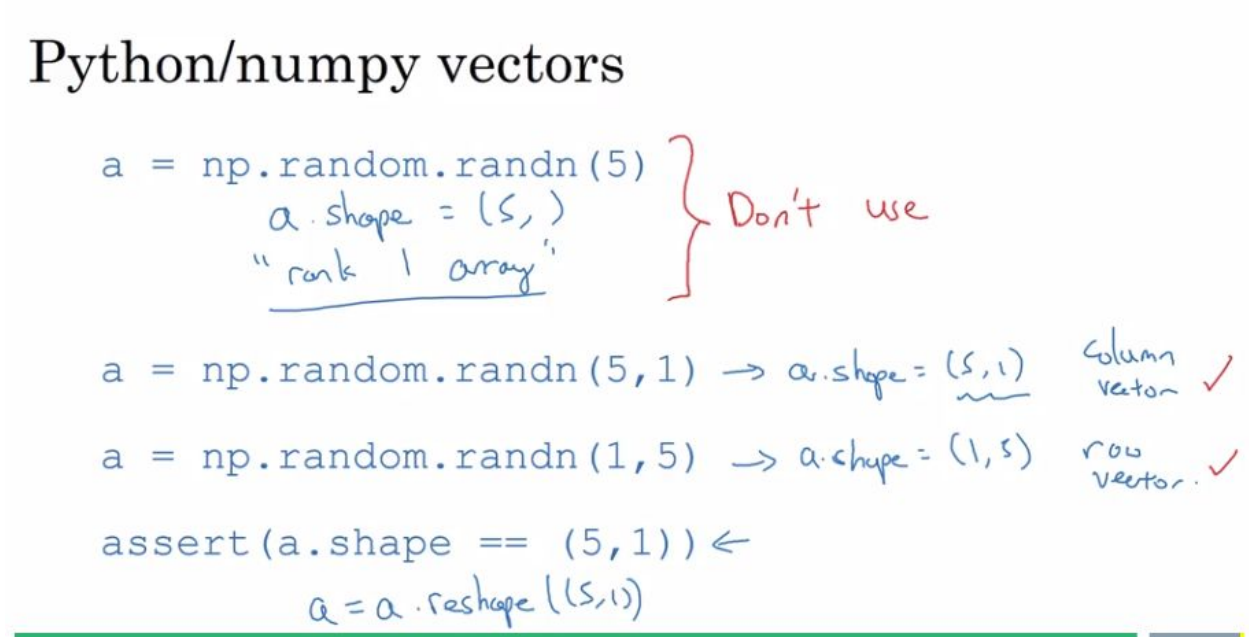
1) | 1 |
   | 2 | + 100 python automatically converts it too a 4 x 1 matrix of 100's
   | 3 | add 100 to all elements
   | 4 |

General Principle of BroadCasting



Python: A.sum(axis = 0) sums axis vertically , if axis = 1 the horizontally

Python TIPS -> AVOID RANK 1 Array's and use col/row vectors



TO MAKE a IMAGE MATRIX Flatter -> (So that every column vector repersents one image

[image1, image2 ….. Imagem]

A trick when you want to flatten a matrix X of shape (a,b,c,d) to a matrix X_flatten of shape (b∗c∗d, a) is to use:

```
X_flatten = X.reshape(X.shape[0], -1).T      # X.T is the transpose of X
```

Feature scaling is a method used to standardize the range of independent variables or features of data

Motivation:

Since the range of values of raw data varies widely, in some machine learning algorithms, objective functions will not work properly without normalization. For example, the majority of classifiers calculate the distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature. Therefore, the range of all features should be normalized so that each feature contributes approximately proportionately to the final distance.

Another reason why feature scaling is applied is that gradient descent converges much faster with feature scaling than without it.[1]

We use the standardization technique, but for images better to just divide by 255

> To represent color images, the red, green and blue channels (RGB) must be specified for each pixel, and so the pixel value is actually a vector of three numbers ranging from 0 to 255.
>
> One common preprocessing step in machine learning is to center and standardize your dataset, meaning that you substract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array. But for picture datasets, it is simpler and more convenient and works almost as well to just divide every row of the dataset by 255 (the maximum value of a pixel channel).
>
> Let's standardize our dataset.

Overfitting: Is extremely affected by the number of iteration for gradient descent, since the more you optimize the the function based on the training data, the more noise you can pick up within the parameters.

- Also affected by the learning rate

More complex Neural Networks

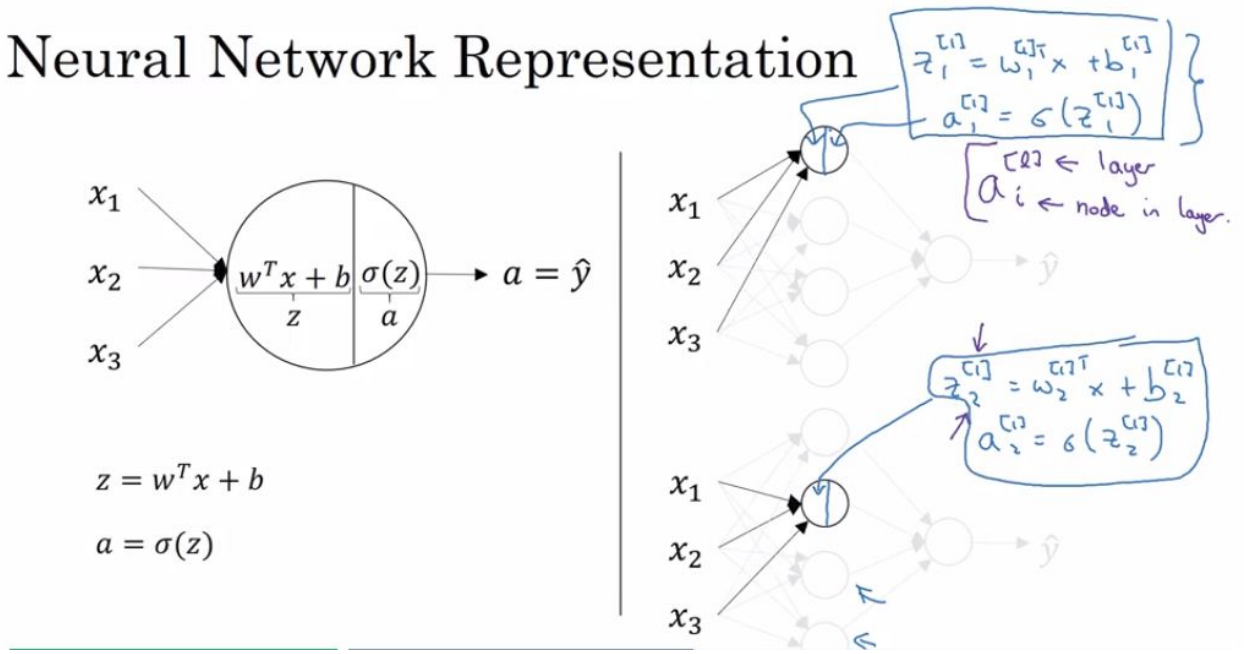
The superscripts [ ] represent a layer

Taking logistic regression and repeating it

Hidden Layer: Don't see it in the training set

Number of Layers: Don't count the input layer

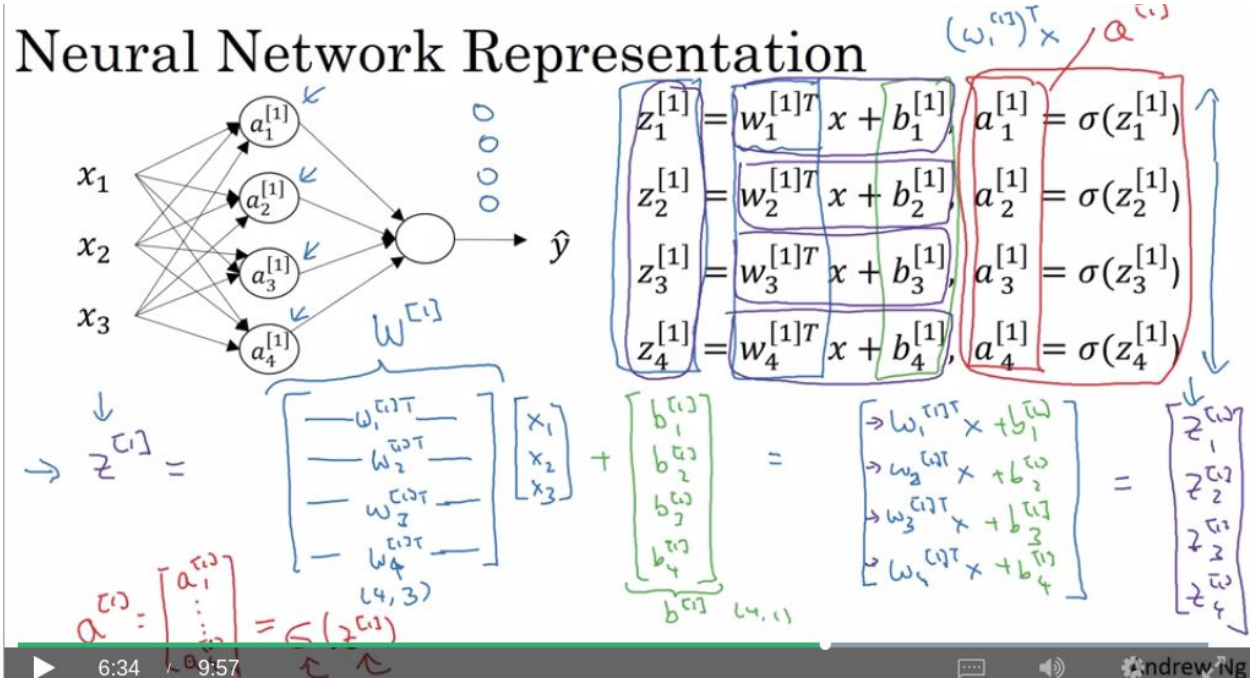
Hidden Layers and Output Layers have parameters associated with it



Neural Networks dimension

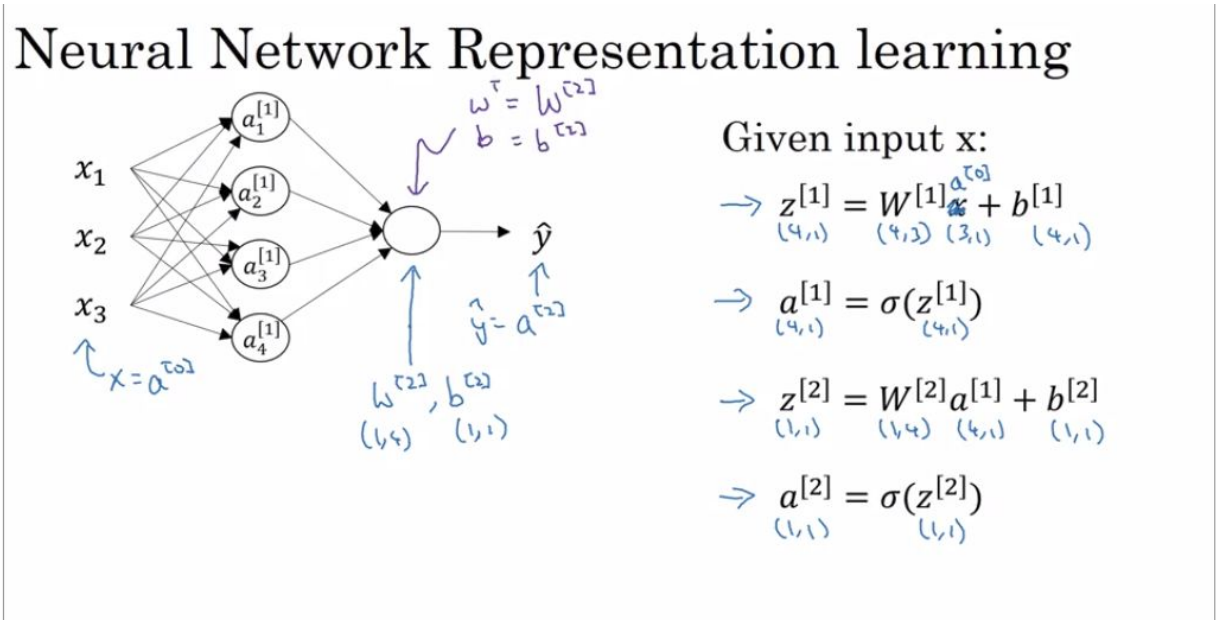
Output Layer Dimension W: (hidden layer nodes, 1)

Hidden Layer Dimension W[1]: (hidden layer nodes, # of inputs)



Summarizing the Notation:

W[1] basically now have multiple pairs of parameters where the number of pairs equals the nodes in the hidden layer and the number of parameters always equals the number of inputs.Thats how we get the W[1] matrix



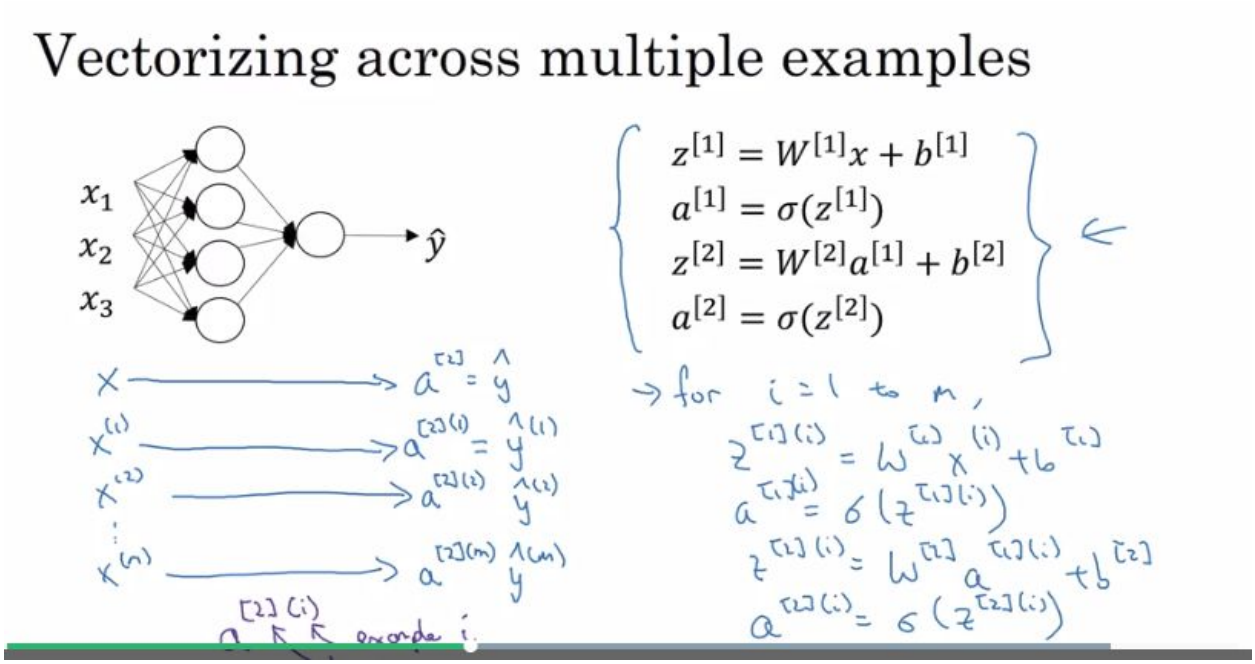
W[2] is a little bit confusing

Basically W[2] is (1,4) since we only have one node in this layer and 4 inputs to this layer.

In the example z[2] and a[2] represent the Output layer

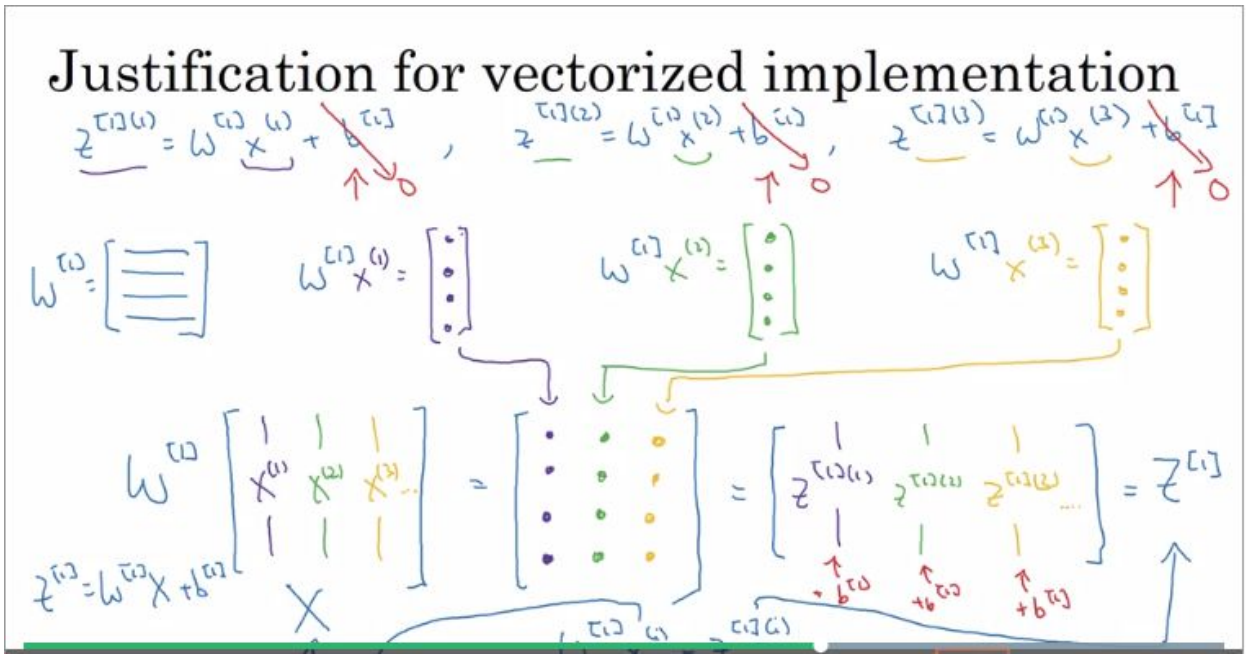
The four instructions above represent code for the above neural network

Vectorization of Layered Neural Networks



All the capital letter Matrices -> each column represents a different training example

-> each row represents a different hidden unit / node / feature / parameter



Activation functions

Example: Sigmoid

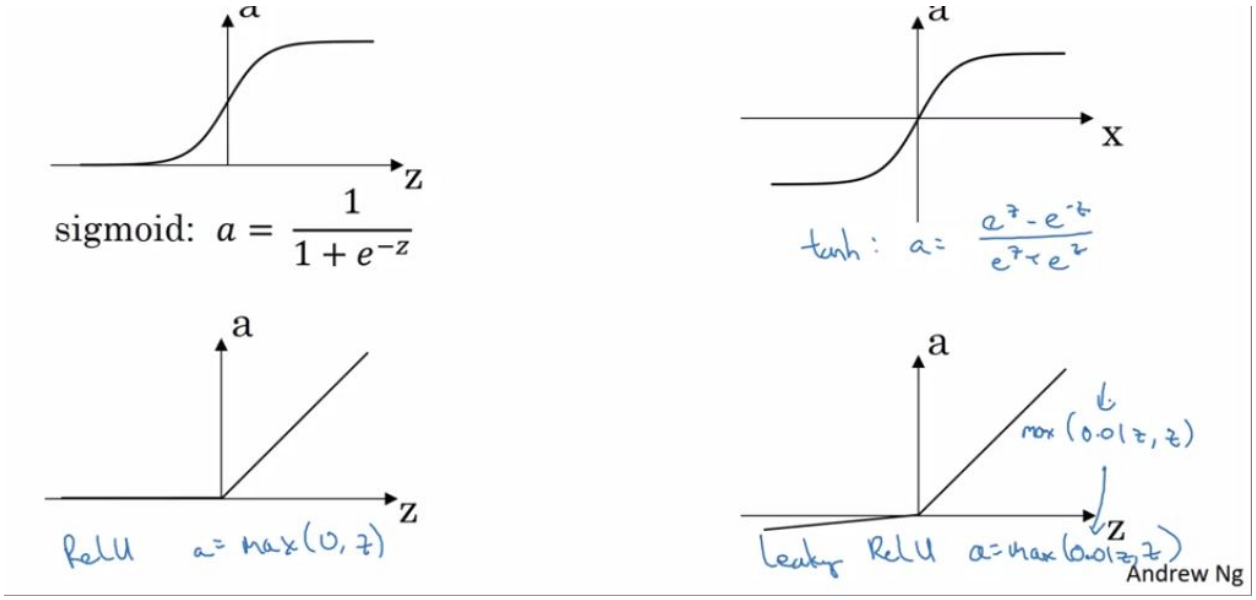
Tanh function almost always works better than the sigmoid function

Makes sure that the data is centered and the mean is 0 makes it easier for the next layer

Exception: Use the sigmoid function if your doing a binary classification problem only for the output layer, since y -> {0.1} so you want 0 <= y_hat <= 1 (y_hat is basically col of A) If

If Z is very large or small, in both tanh and sigmoid function the gradient becomes very slow since the slope decreases (look at visual below)

USE RELU ACTIVATION!!! (Neural Network work will work faster, Slope is greater)

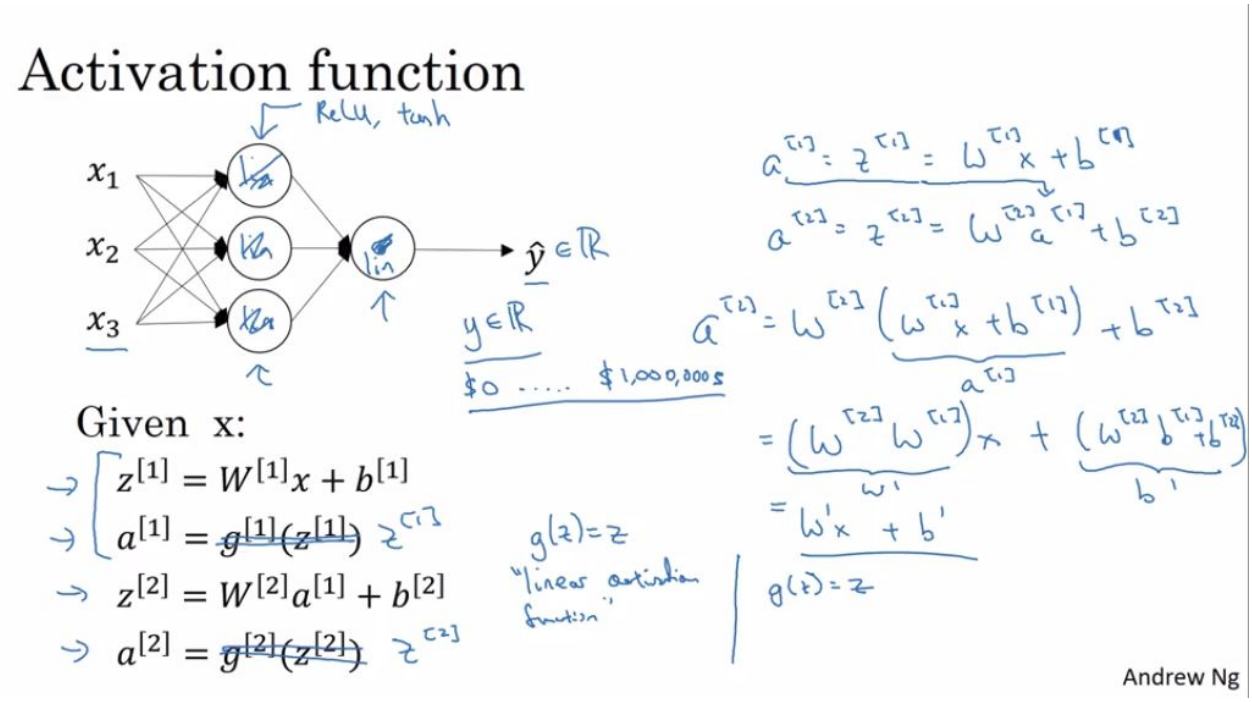


Why do we need a activation function

Because if we don't use a activation function we get a linear function.

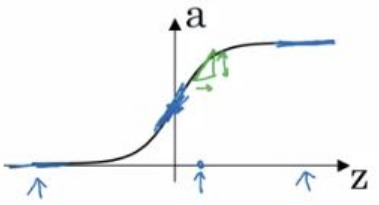
A Linear hidden layer is useless (a layer that has no activation function)

Because compositions of linear functions are linear, not really doing anything interesting



A place where you could use linear activation functions (aka no activation function) is in the output layer if the output you want forexample is a real number from negative …. to a million dollars

Sigmoid activation function



$$g(z) = \frac{1}{1+e^{-z}}$$

$a = g(z) = \frac{1}{1+e^{-z}}$

$g'(z) = \frac{d}{dz} g(z)$ = slope of $g(x)$ at z

$= \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right)$

$= g(z)\left(1 - g(z)\right)$ ⇐

$= a(1-a)$

$g'(z) = a(1-a)$ ↑

$z = 10$. $g(z) \approx 1$

$\frac{d}{dz} g(z) \approx 1\,(1-1) \approx 0$

$z = -10$ $g(z) \approx 0$

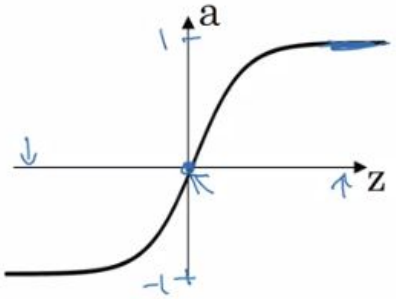
$\frac{d}{dz} g(z) \approx 0 \cdot (1-0) \approx 0$

$z = 0$ $g(z) = \frac{1}{2}$

$\frac{d}{dz} g(z) = \frac{1}{2}\left(1 - \frac{1}{2}\right) = \frac{1}{4}$

Andrew Ng

Tanh activation function



$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$g'(z) = \frac{d}{dz} g(z)$ = slope of $g(z)$ at z

$= 1 - (\tanh(z))^2$ ←

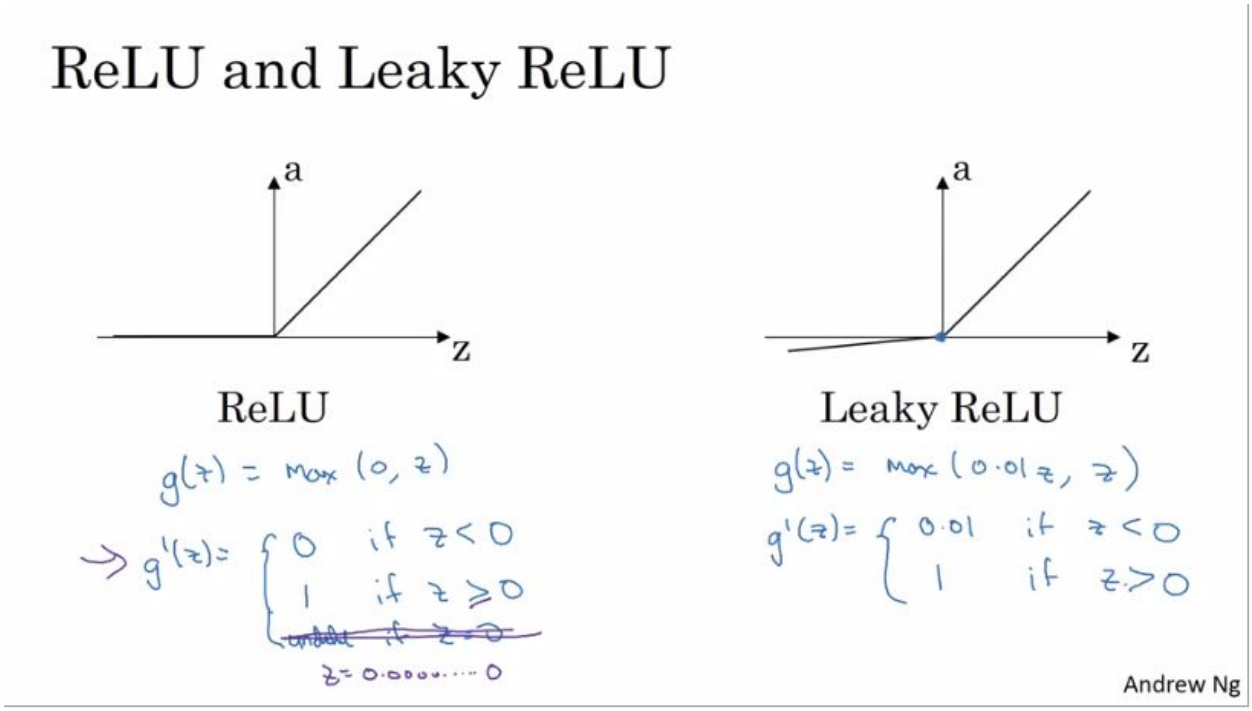
$a = g(z)$, $g'(z) = 1 - a^2$

$z = 10$ $\tanh(z) \approx 1$
$g'(z) \approx 0$

$z = -10$ $\tanh(z) \approx -1$
$g'(z) \approx 0$

$z = 0$ $\tanh(z) = 0$
$g'(z) = 1$

Andrew Ng



Gradient Descent for Neural Networks

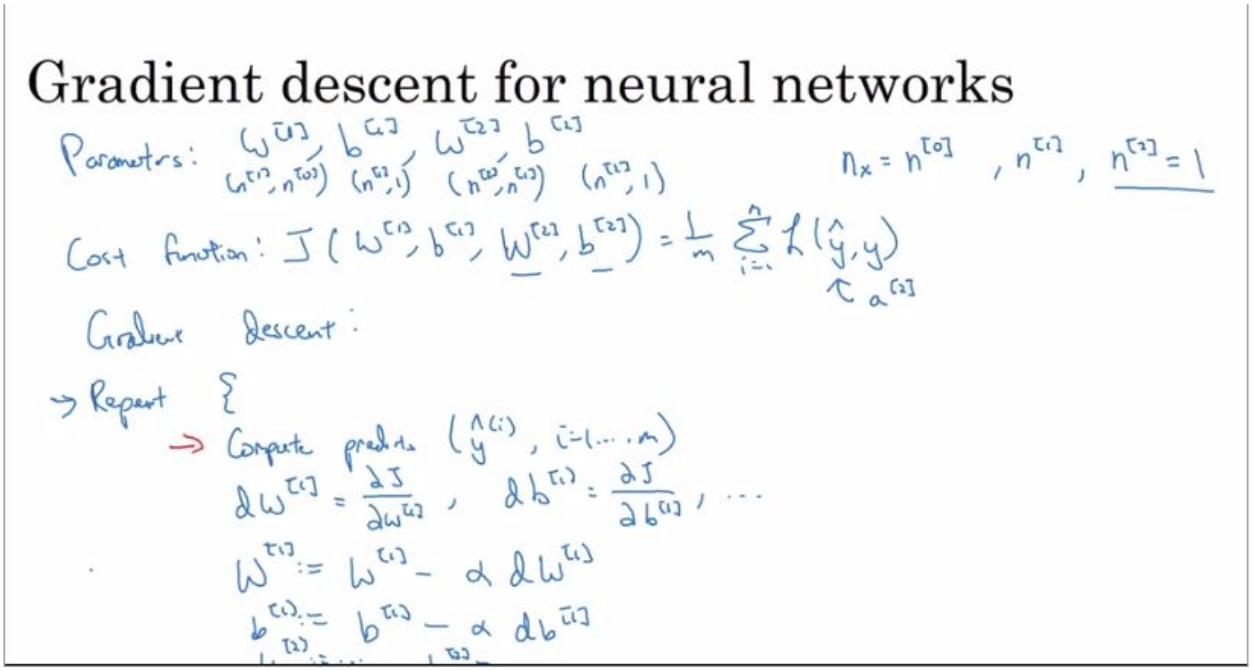
dimension :

N[0] is input

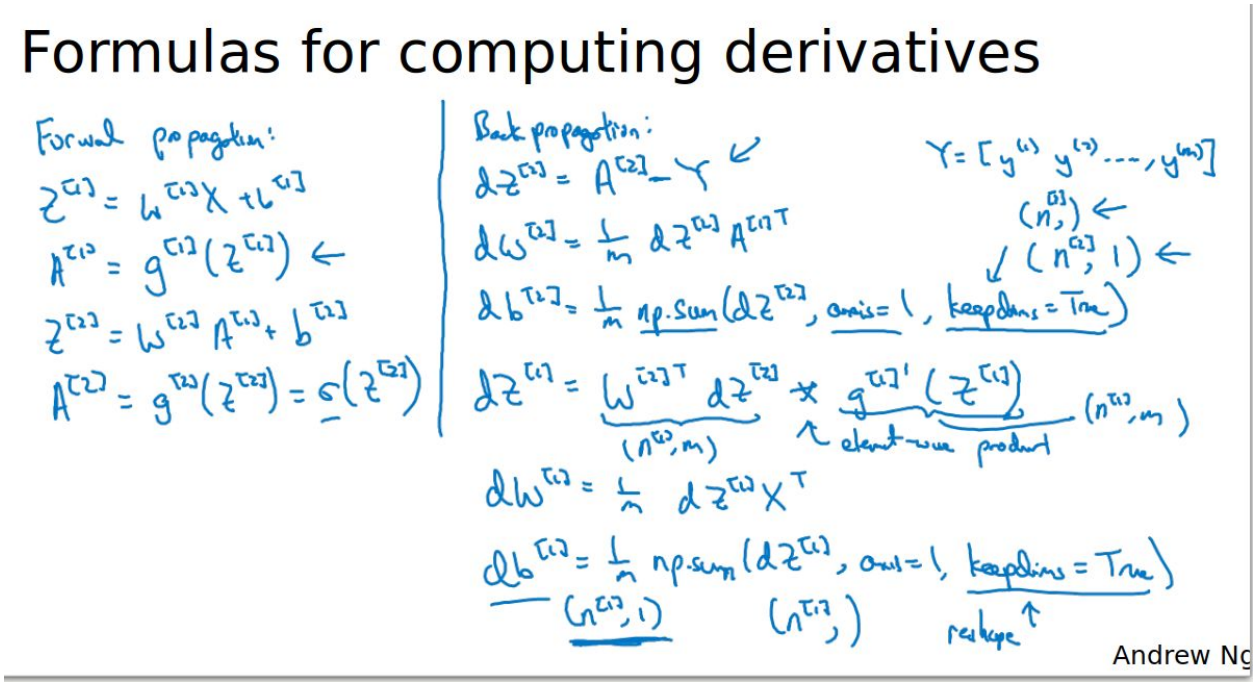
N[1] is hidden layer

N[2] is output

Summary if how gradient descent will work



Derivatives For Gradient Descent



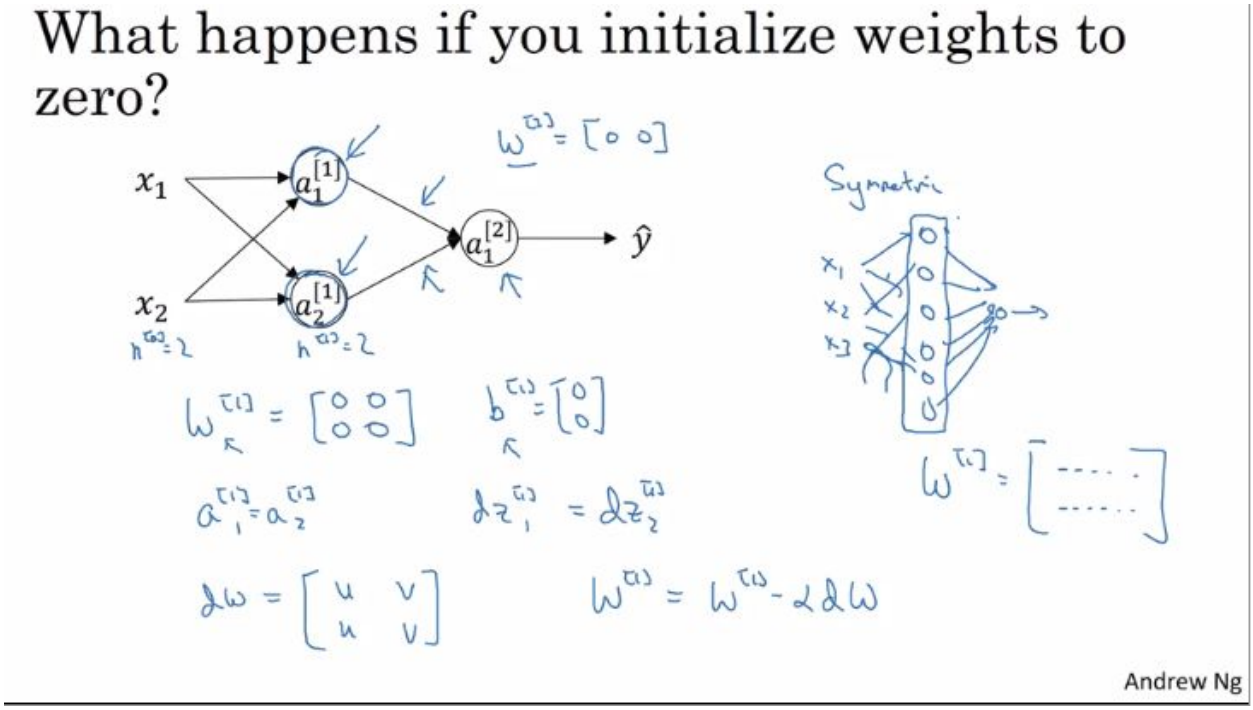
Random Initialization

For Logistic Regression okay to start with values of zero, but for Neural Networks must randomize the w and b

Explanation For Diagram Below: Both Units in the Hidden Layer compute the same function, so by symmetry dz[1]1 = dz[1]2, and each row of dw (aka dL/dw) end up being identical, so when you do

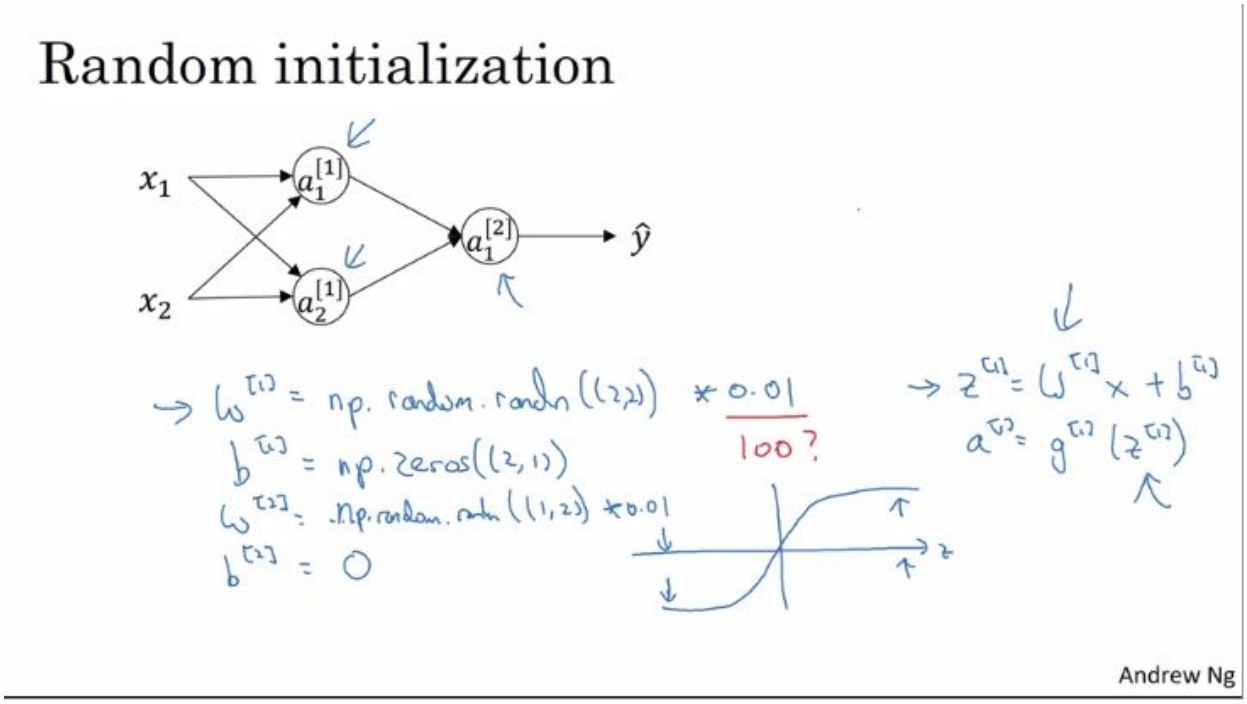
W[1] = W[1] - learning_rate(dw) W[1] gets identical rows

YOU WANT DIFFERENT HIDDEN UNITS TO COMPUTE DIFFERENT FUNCTIONS THUS DIFFERENT PARAMETERS



SO MUST INITIALIZE W RANDOMLY, b doesn't matter since W is already random, so we get random number finally

WE LIKE THE RANDOM INITIALIZATION of numbers to be very small so that's why we divide by 100. This is because if it's very large then for tanh and sigmoid function you end up at the edges were slope is very little



FOR DEEP NEURAL NETWORKS might want to multiply initialization of W with different constant.

SUMMARY OF DERIVATIVES NEURAL NETWORK

Summary of gradient descent

$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]} a^{[1]T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$	$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$
$dW^{[1]} = dz^{[1]} x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

Andrew Ng

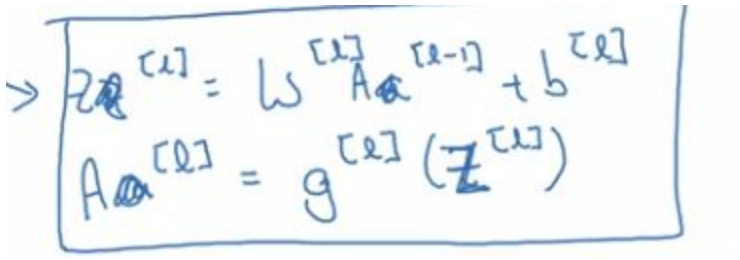
Accuracy is really high compared to Logistic Regression. The model has learnt the leaf patterns of the flower! Neural networks are able to learn even highly non-linear decision boundaries, unlike logistic regression.

Deep Neural Networks

The output layer is denoted as L, and the activation if layer L

a[L] is y_hat, it is the predicted output

Forward Propagation

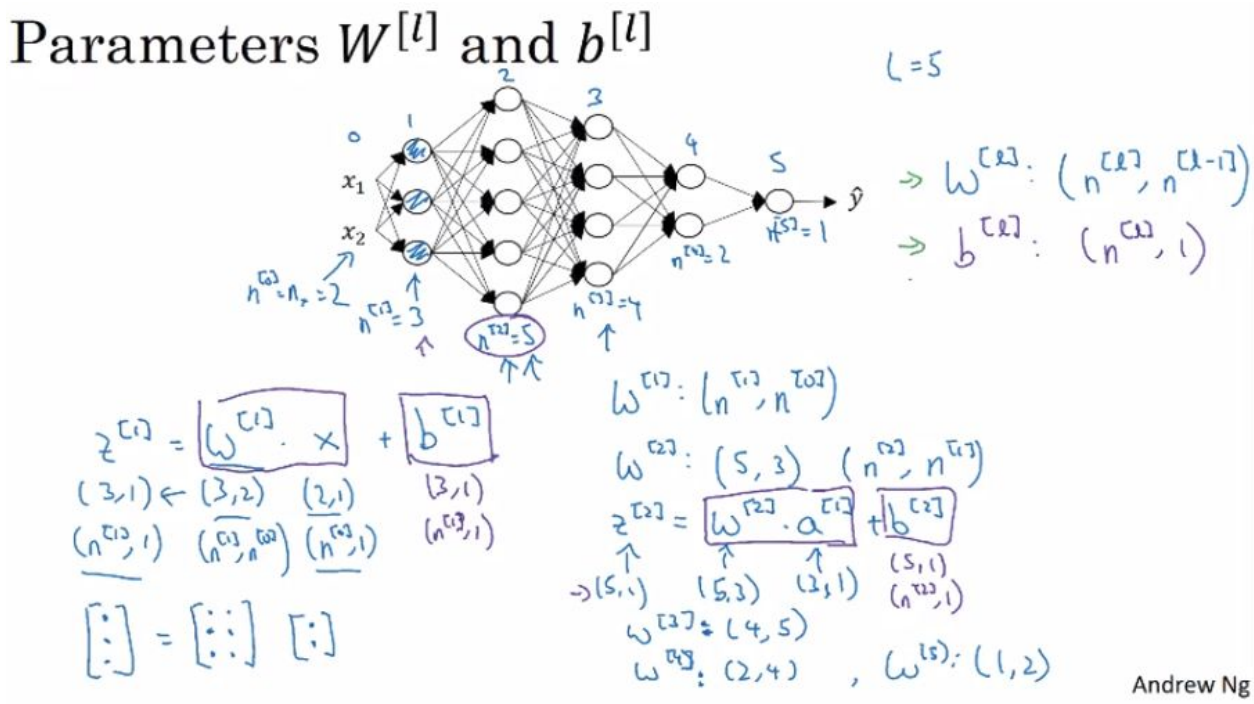


Okay to have a for loop

Sizes of W parameters:

W[L]: {n[L] x n[L-1]}

b[L]: {n[L] x 1}



If making a neural network more shallow, you might need to add more nodes in the hidden layer to make up for the logical thought process.

Cache in implementation is used to store the paramters Wi, Zi, bi so you can use it in backpropagation to calculate derivatives can calualate new Wi and bi's.

Back Propagation Step Summary:

Backward propagation for layer l

→ Input $da^{[l]}$

→ Output $da^{[l-1]}, dW^{[l]}, db^{[l]}$

$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$

$dW^{[l]} = dz^{[l]} \cdot a^{[l-1]}$

$db^{[l]} = dz^{[l]}$

$da^{[l-1]} = W^{[l]T} \cdot dz^{[l]}$

$dz^{[l]} = W^{[l+1]T} dz^{[l+1]} * g^{[l]\prime}(z^{[l]})$

$dz^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$

$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$

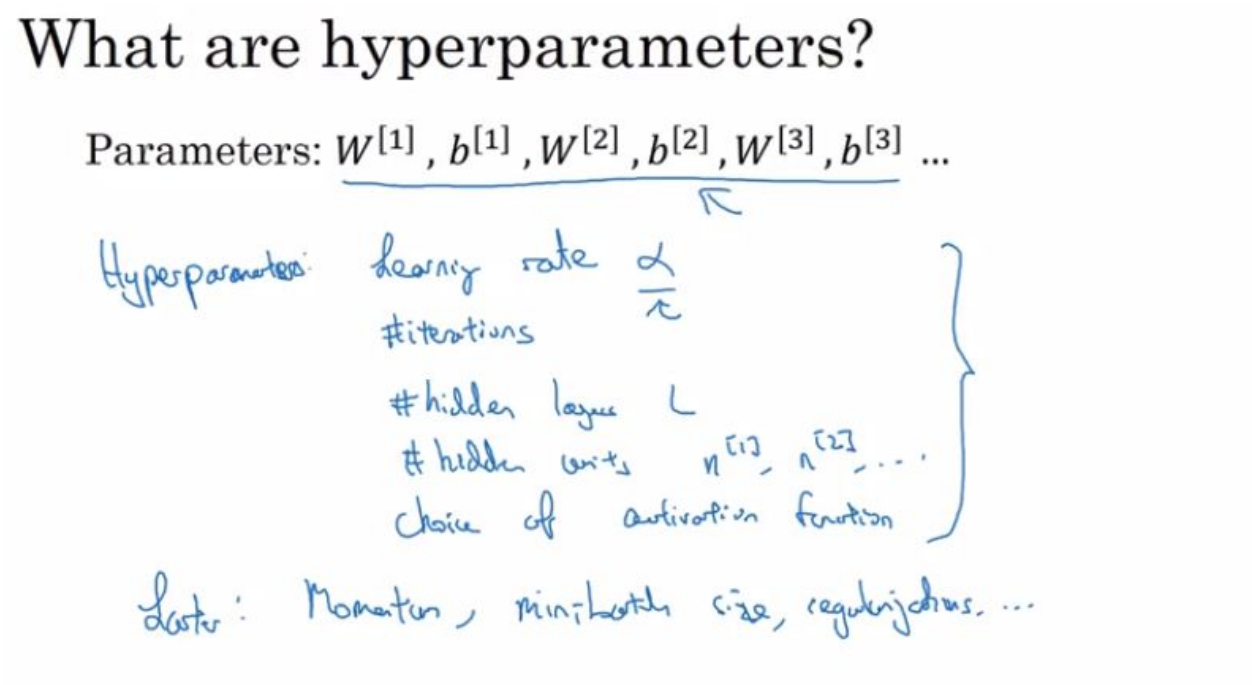
$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis=1, keepdims=True)$

$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$

Andrew Ng

Hyperparamters

Are things that affect the parameters



Try multiple values for hyper parameters

Back Propagation In Practise CODE

```python
def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer (layer l)

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of current layer l)
    cache -- tuple of values (A_prev, W, b) coming from the forward propagation in the current layer

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape as W
    db -- Gradient of the cost with respect to b (current layer l), same shape as b
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    ### START CODE HERE ### (≈ 3 lines of code)
    dW = 1/m * np.dot(dZ, A_prev.T)
    db = 1/m * np.sum(dZ, axis = 1, keepdims = True)
    dA_prev = np.dot(W.T,dZ)
    ### END CODE HERE ###
```

Course 2: Improving Deep Neural Networks, Hyper Parameter Tuning, Regularization adn Optimization

Hyperparameters

Layers

Hidden Units

learning Rates

Activation Functions

Very iterative project idea, experiment, code

Data training/dev/test

Dev set is to try and tune the model

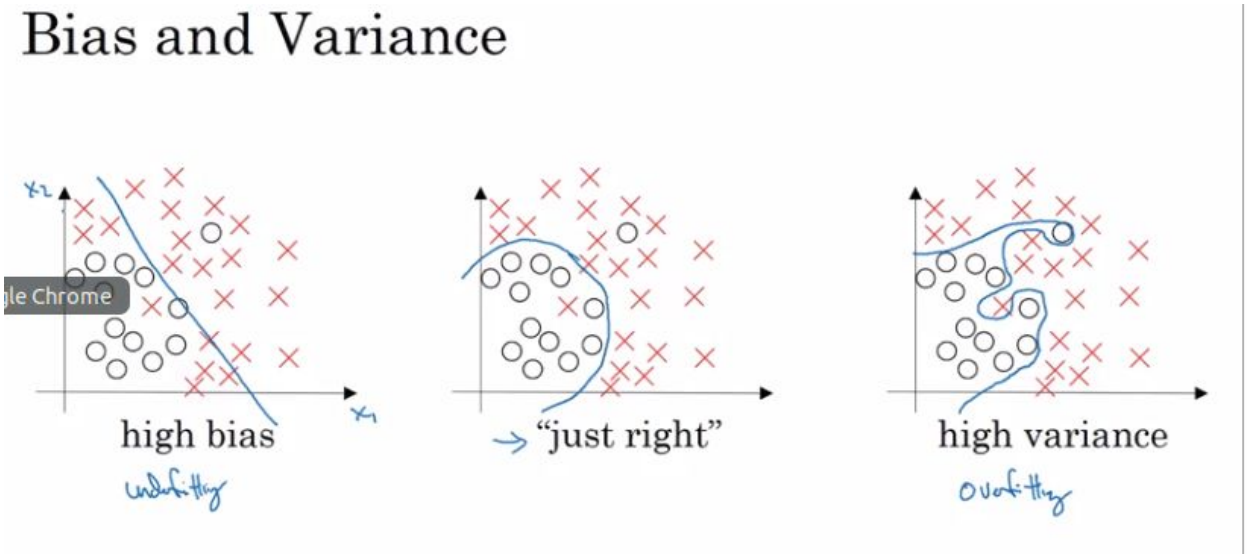
Test set is only required for unbiased estimator

1000, 10000 -> should do 70/30 60/20/20

Big Data -> millions just pick 99/1 98/1/1 (like 10000 tests examples)

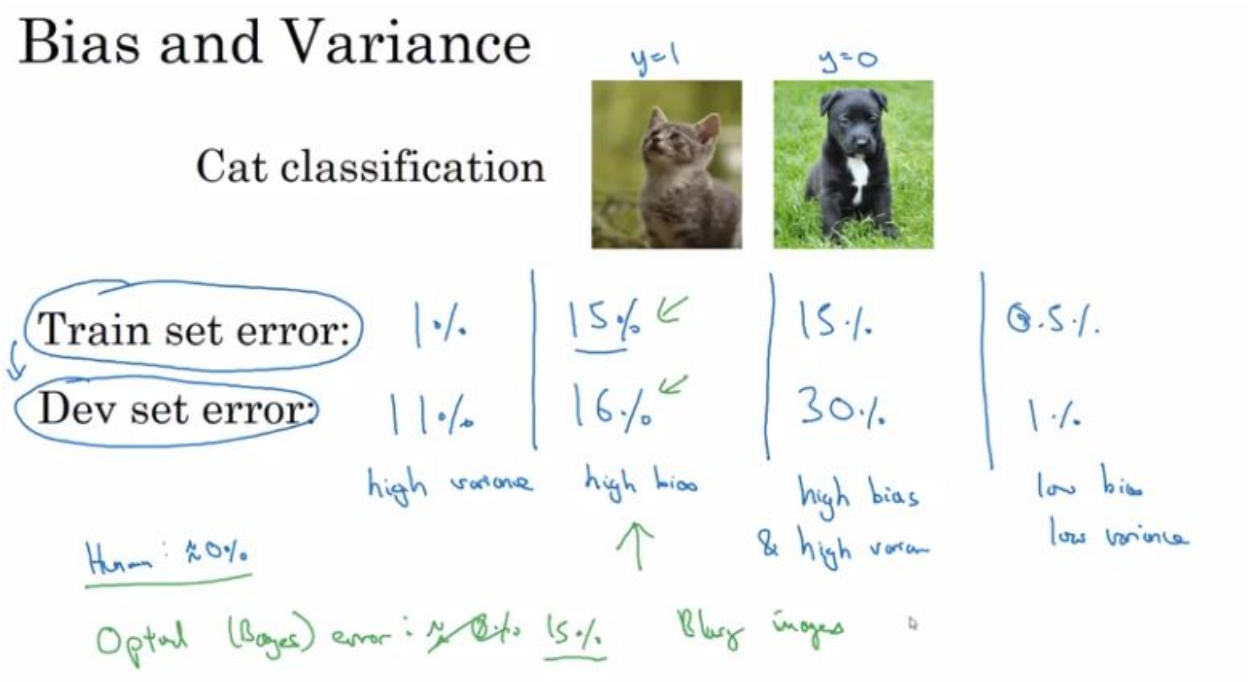
Make sure dev and test sets come from the same distribution

Bias/Variance



High Variance -> Overfitting

High Bias -> Underfitting



Optimal is how much error in terms of human prediction, the bias and variance above is for optimal of 0%, if it was 15% #2 col would be low bias and variance

<u>Recipe To Improve Error Bias/Variance</u>

Training Data

High Bias -> Bigger Network (More Layers) (more computational Time)

-> Train Longer

-> Different Architecture

Dev Data

High Variance -> More Data

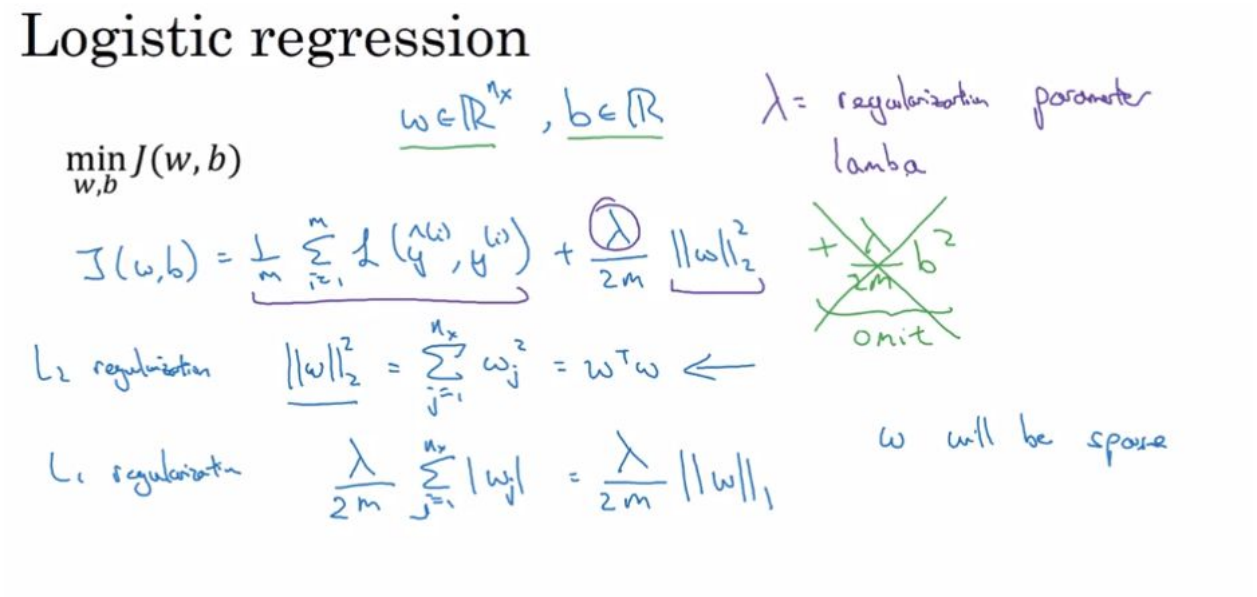
-> Regularization

-> Architecture

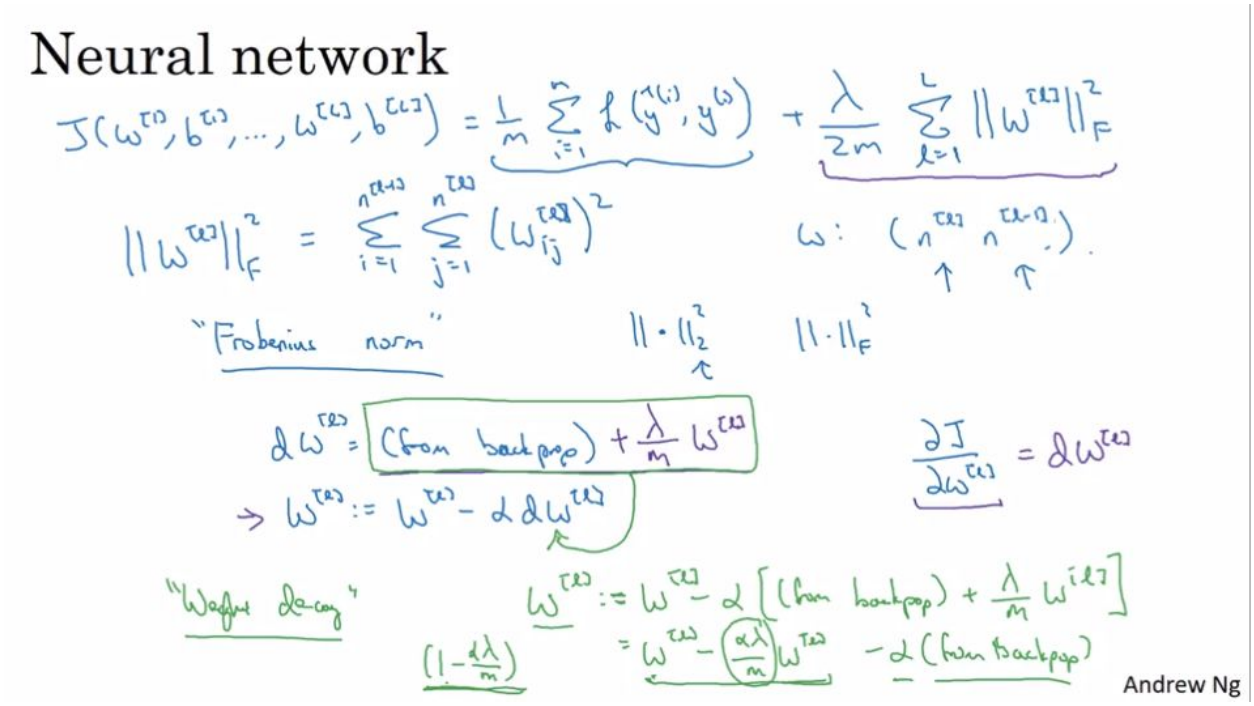
Improvement has been shown with just adding more layers and including more dev data, reduces both bias and variance

Regularization (To reduce overfitting and variance)

Use L2 Norm for logistic regression

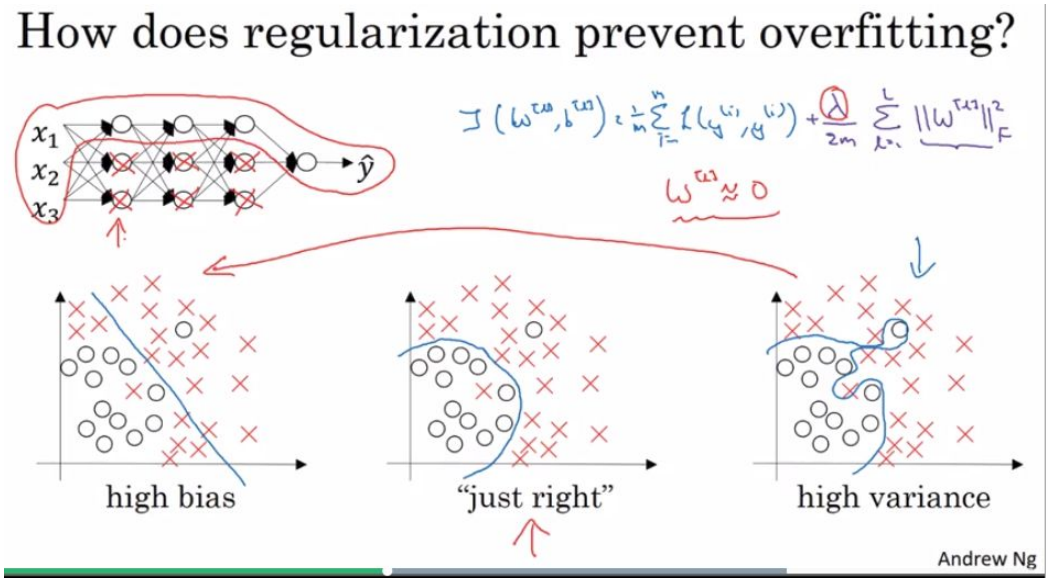


Using L2 / Weight Decay Regularization Technique to make values in W a bit smaller



How does regularization minimize overfitting?

Hidden Units become very close to zero, so it becomes a simple neural network and follows the arrow to being less variance , but increasing bias, there must be an intermediate values for lambda thats just right.

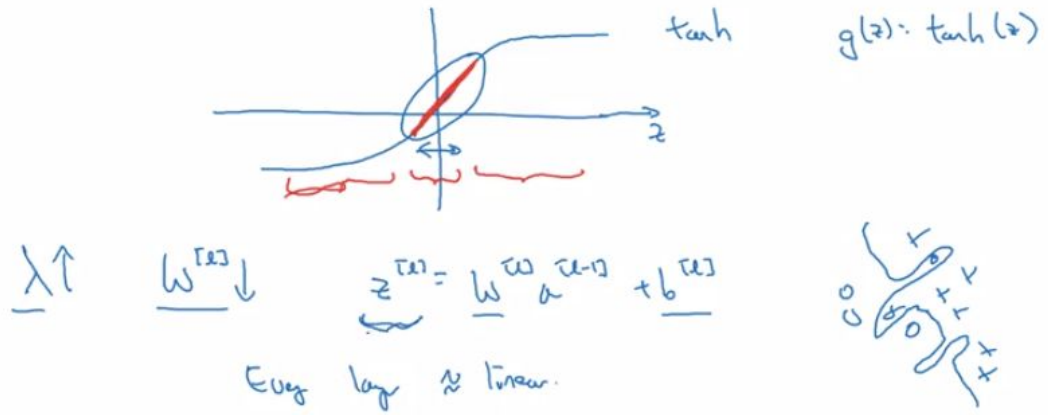


A more intuitive reason to why regularization, prevents overfitting

Lambda is bigger -> W is smaller -> Z has smaller values so in the graph below shows that g(Z) can be only linear. So if every layer is linear, then your whole network is only linear.

Can't get some crazy function like the one shown below to fit the data.

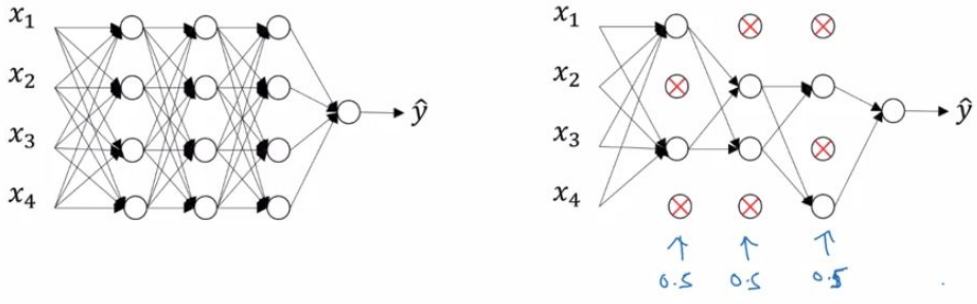
How does regularization prevent overfitting?



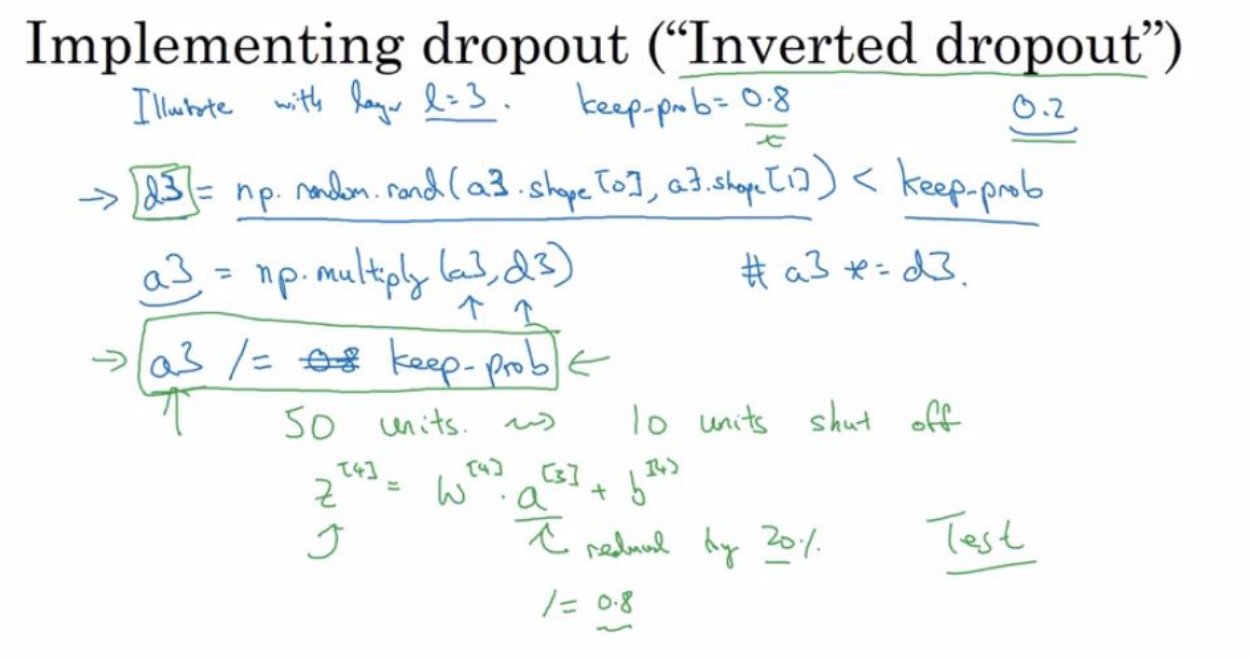
Dropout Regularization

Basically each layer has a probability of dropping some of its hidden units, (not using them), taking the links away

Dropout regularization



Implementing Dropout



Basically On Every Iteration of Gradient Descent You Drop Different Hidden Units Randomly.

During Testing you shouldn't drop any hidden units!!!!!

Can vary Keep_Prob for each hidden layer, Depending on which layer you feel there is overfitting

Make Sure gradient descent is working without dropot, decreasing monotonically, then try dropout

Why Dropout Actually Works

Can't rely on any single feature, this is how overfitting happens, Spread out the weights for each features

Dropout is used greatly used in COMPUTER VISION, since you don't have a lot of data and way to many pixels

Other ways of decreasing overfitting

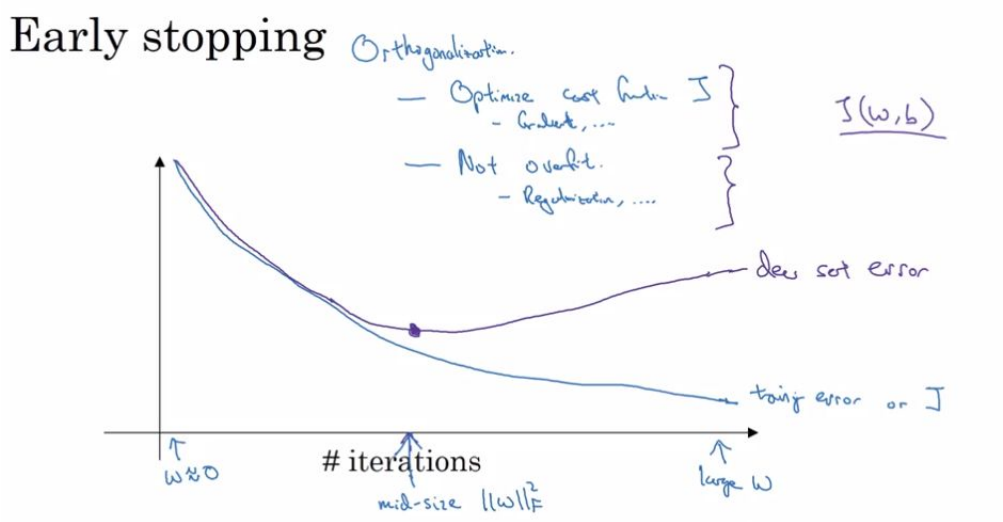
Increasing Data:

Can increase data for computer vision by rotating cat images, and flipping them, , zooming double, triple your data.

Early Stopping:

Stopping gradient descent early since W is smaller usually at the beginning, when W is smaller it minimizes overfitting

Disadvantage: Cannot Optimize the cost function J, should deal with optimization and overfitting separately



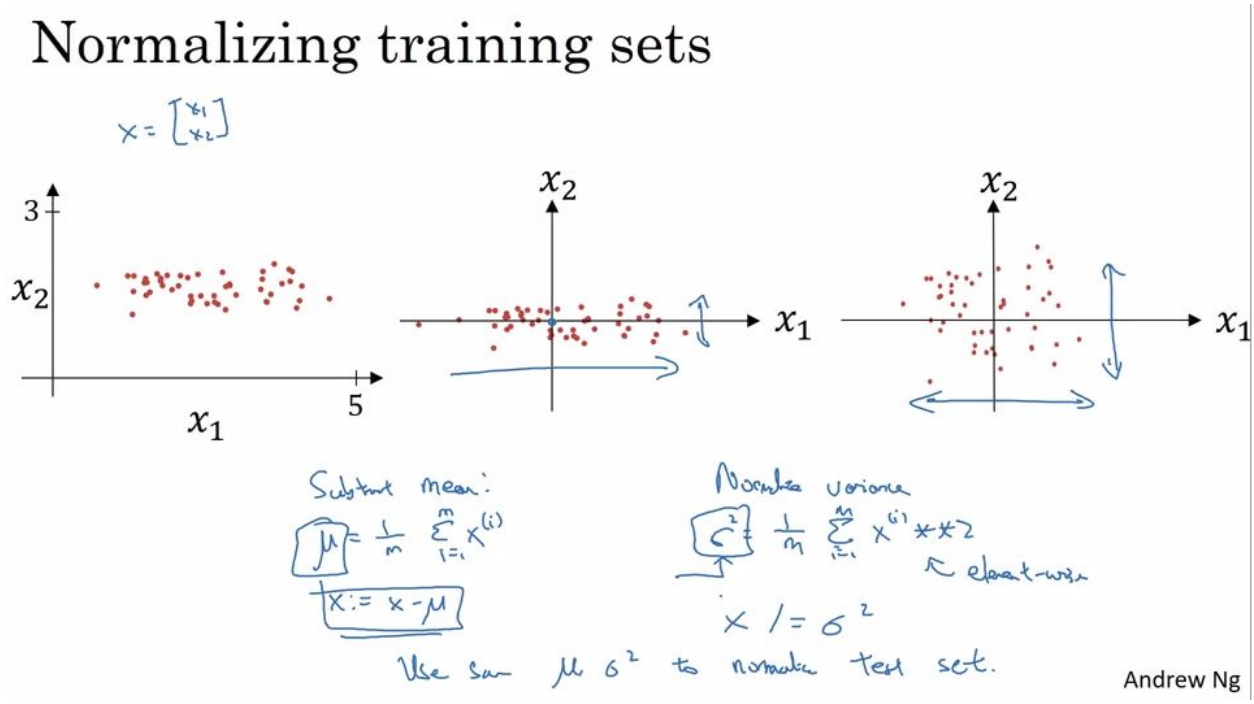
Disadvantage to L2 is the amount of values you have to test for lamda

Normalization

Subtract The Mean to have 0 mean

Normalize the variance

Don’t normalize the test and training set differently, us the same mean and varaince



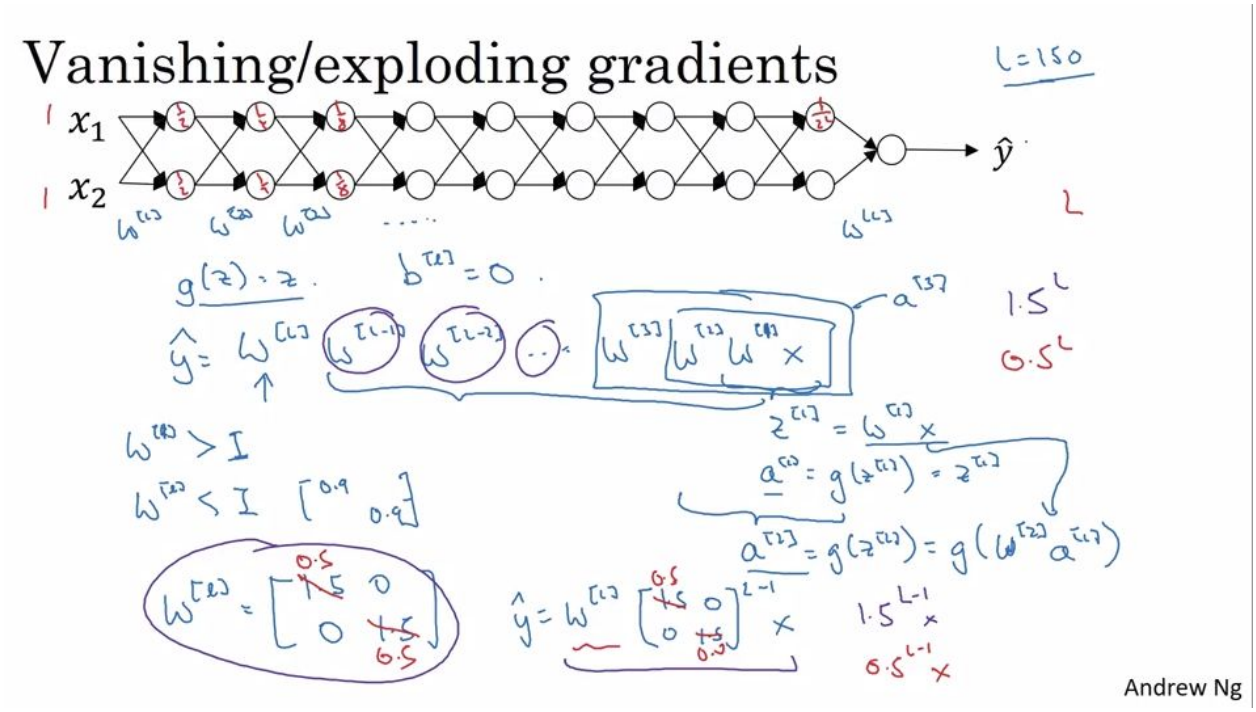
Why to Normalize?

Since features are in different scales, w1 and w2 will have different scale of values, elongated convex bowl gradient descent.

For Unnormalized data the learning needs to be small, since you have to be careful to take small steps to get to the minimum, once normalized, learning rate can be large, since it goes straight to the minimum, less spread in data.

<u>Vanishing / Exploding Gradients</u>

For Deep Networks, The values of the activation functions can increase or decrease exponential, as L gets very large, therefore

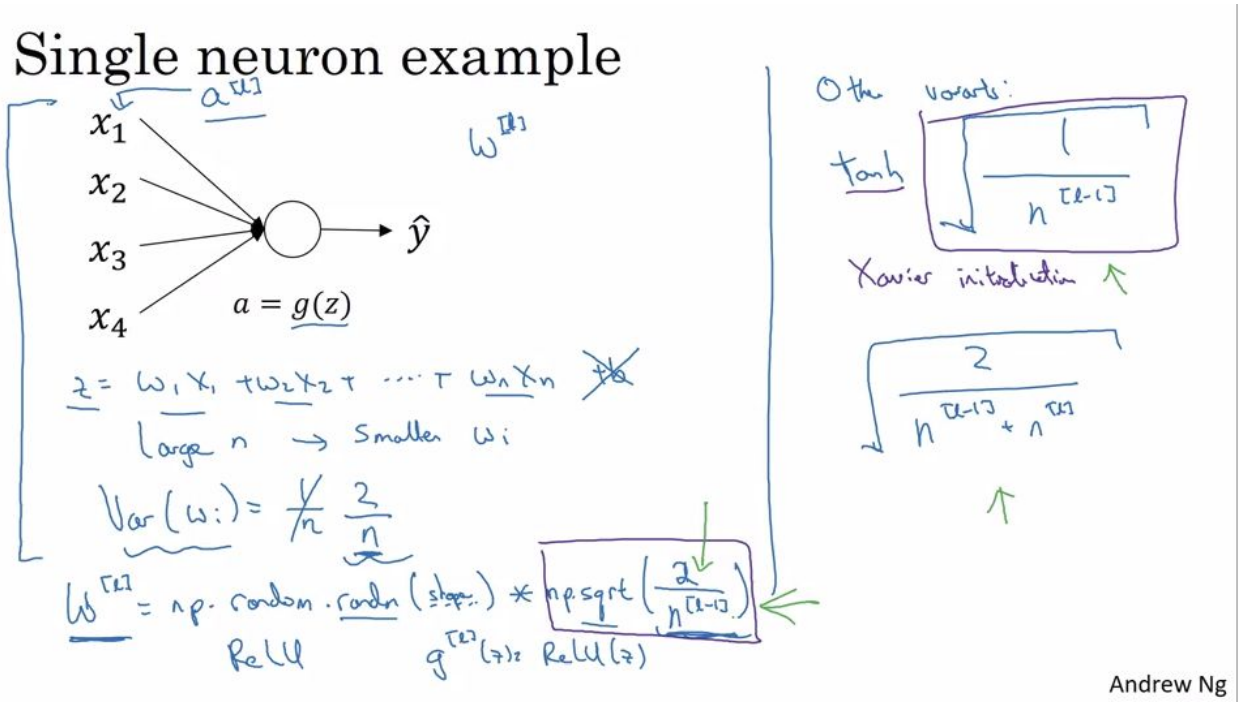


Derivative Values can get very Big Or Very Small, so learning can happen very slowly or too large steps to never reach minimum

<u>Partial Solution</u>

Weight Initialization

Not Too Important to Tune



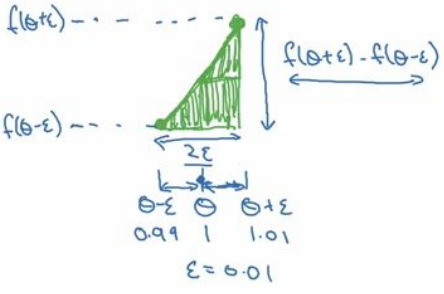
Numerical Approximation Of Gradients

2 sided derivative calculation (f(theta + e) - f(theta - e)) / 2e has a lower approximation error of the derivative compared to the one sided derivative of (f(theta + e) - f(theta)) / e

Where e = epsilion

Checking your derivative computation

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



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

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

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

approx error: 0.0001

(prev slide: 3.0301. error: 0.03)

USE THE TWO SIDED DIFFERENCE

Gradient Checking Debugging

Gradient check for a neural network

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

concatenate

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

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

Gradient checking (Grad check)

$J(\theta) = J(\theta_1, \theta_2, \ldots)$

for each i:

$d\theta_{approx}[i] = \dfrac{J(\theta_1, \theta_2, \ldots, \theta_i + \varepsilon, \ldots) - J(\theta_1, \theta_2, \ldots, \theta_i - \varepsilon, \ldots)}{2\varepsilon}$

$\approx d\theta[i] = \dfrac{\partial J}{\partial \theta_i}$ | $d\theta_{approx} \overset{?}{\approx} d\theta$

Check $\dfrac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$ $\approx$ 10^{-7} — great! 10^{-5}

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

Gradient checking implementation notes

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

Andrew Ng

Initialization of Parameters W

Random Intialization

The model is predicting 0 for every example.

In general, initializing all the weights to zero results in the network failing to break symmetry. This means that every neuron in each layer will learn the same thing, and you might as well be training a neural network with n[l]=1n[l]=1 for every layer, and the network is no more powerful than a linear classifier such as logistic regression.
What you should remember:

- The weights W[l]W[l] should be initialized randomly to break symmetry.
- It is however okay to initialize the biases b[l]b[l] to zeros. Symmetry is still broken so long as W[l]W[l] is initialized randomly.

The Values of Intialization

- The cost starts very high. This is because with large random-valued weights, the last activation (sigmoid) outputs results that are very close to 0 or 1 (mathematically) for some examples, and when it gets that example wrong it incurs a very high loss for that example (because you're comparing y_hat and y).
- Poor initialization can lead to vanishing/exploding gradients, which also slows down the optimization algorithm.
- If you train this network longer you will see better results, but initializing with overly large random numbers slows down the optimization.

In summary:

- Initializing weights to very large random values does not work well.
- Hopefully intializing with small random values does better. The important question is: how small should be these random values be? Lets find out in the next part!

SUMMARY PARAMETER INITIALIZATION

- Different initializations lead to different results
- Random initialization is used to break symmetry and make sure different hidden units can learn different things
- Don't intialize to values that are too large

- He initialization works well for networks with ReLU activations.
- He =`sqrt(1./layers_dims[l-1]`

Regularization Applied

L2

Forward Propagation:

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

Backward propagation:

Exercise: Implement the changes needed in backward propagation to take into account regularization. The changes only concern dW1, dW2 and dW3. For each, you have to add the regularization term's gradient ($\frac{d}{dW}(\frac{1}{2}\frac{\lambda}{m}W^2) = \frac{\lambda}{m}W$).

SUMMARY L2

Observations:

- The value of λλ is a hyperparameter that you can tune using a dev set.
- L2 regularization makes your decision boundary smoother. If λλ is too large, it is also possible to "oversmooth", resulting in a model with high bias.

What is L2-regularization actually doing?:

L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values.

This happens because greater the lambda the higher the value of the derivative dW and smaller the W.

Smoother model because as described earlier, when W is small Zi = WAi + b, Zi is also small, then the activation function g(Zi) gets value closer to zero, where the slope is linear, so dA tends to be linear, and you get a lot linear functions being multiplied with matrices.

dWi = 1./m * np.dot(dZi, A[i-1]) + (lambd/m)*Wi

So you end up with a linear prediction

It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.

What you should remember -- the implications of L2-regularization on:

- The cost computation:
    - A regularization term is added to the cost
- The backpropagation function:
    - There are extra terms in the gradients with respect to weight matrices
- Weights end up smaller ("weight decay"):
    - Weights are pushed to smaller values.

DROPOUT SUMMARY:

- A **common mistake** when using dropout is to use it both in training and testing. You should use dropout (randomly eliminate nodes) only in training.
- Deep learning frameworks like tensorflow, PaddlePaddle, keras or caffe come with a dropout layer implementation. Don't stress - you will soon learn some of these frameworks.

What you should remember about dropout:

- Dropout is a regularization technique.
- You only use dropout during training. Don't use dropout (randomly eliminate nodes) during test time.
- Apply dropout both during forward and backward propagation.

- During training time, divide each dropout layer by keep_prob to keep the same expected value for the activations. For example, if keep_prob is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when keep_prob is other values than 0.5.

HOW TO IMPLEMENT GRADIENT CHECK

Exercise: Implement gradient_check_n().

Instructions: Here is pseudo-code that will help you implement the gradient check.

For each i in num_parameters:

- To compute `J_plus[i]`:
  1. Set θ^+ to `np.copy(parameters_values)`
  2. Set θ_i^+ to $\theta_i^+ + \varepsilon$
  3. Calculate J_i^+ using to `forward_propagation_n(x, y, vector_to_dictionary(`θ^+ `))`.
- To compute `J_minus[i]`: do the same thing with θ^-
- Compute $gradapprox[i] = \frac{J_i^+ - J_i^-}{2\varepsilon}$

Thus, you get a vector gradapprox, where gradapprox[i] is an approximation of the gradient with respect to `parameter_values[i]`. You can now compare this gradapprox vector to the gradients vector from backpropagation. Just like for the 1D case (Steps 1', 2', 3'), compute:

$$difference = \frac{\|grad - gradapprox\|_2}{\|grad\|_2 + \|gradapprox\|_2} \tag{3}$$

Moving Averages

Number of days to average:

1/1- Beta = Number of Days

Calculate Beta and then plus below to graph Vt

Momentum for gradientv descent with dW tho

Vt = Beta(Vt-1) + (1-Beta)(thetaT)

{thetaT is basically 1 piece of current data} and {Vt-1 is summed up past data}

Bias Stoppage at the begging

The formula is Vt/1 - B^t

RMS is basically squaring the derivative terms since you want to reduce the vertical distance in the step. So dWi or db (any derivative term repersents the vertical distance) so when you square it.

Bigger Number -> Get Very Big

Small Number -> Gets Very Small

Adam = Momentum Optimization + RSM (Root Mean Square) Optimization

ONLY PARAMATER YOU NEED TO TUNE IS is the learning_rate

Alpha = alph0 / (1+ decay_rate * epoc(number of mini batches completed)

Just need to tune decay rate and alpha0

The interview question on why gradient descent is bad for low amounts of data ->

Answer: This is a hige mis concepttion

Based on the number of features

High-> then you could go toward saddle points, not a local optima. To get stuck at a local minimum there is 2^-features probability of that happening.

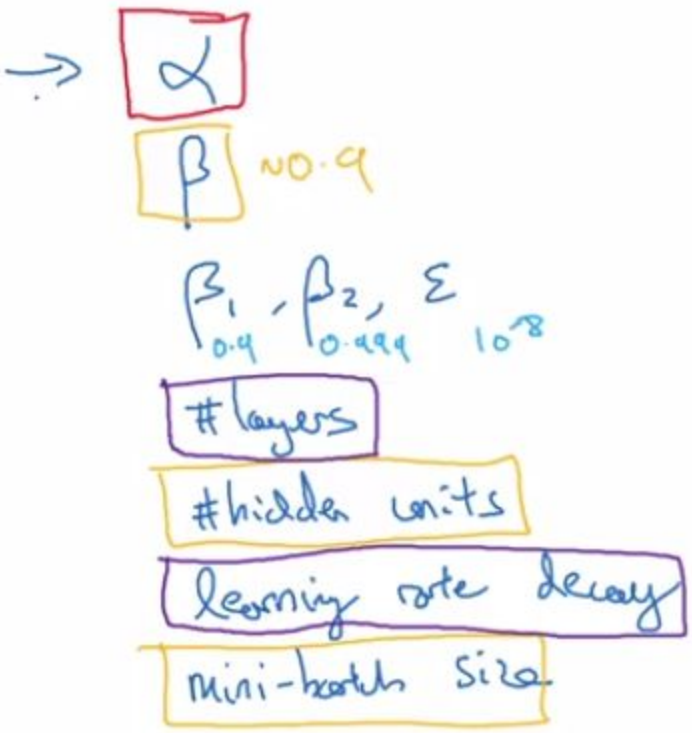
When you low amount of features then probbility increases, but the number of training and test data sets does not affect this

Since your solving problems in such high dimensional spaces, people don't have the intution of what the looks like, so YOU ARE VER VERY UNLUKILY TO GET STUCK IN ON LOCAL MINIMUM

However when you man features, you get alot saddle points, which slow down the learning and make gradient descent really slow

Hyperparameters Tuning Rules

Hyperparameters



Red -> 1st

Yellow -> second

Purple -> 3rd

Not Circled -> like beta1 and beta2 and epsilon are left usually constant

USE RANDOM Sampling, however some variables might need to be scaled like alpha

Alpha -> logarithmic

Beta is also logarithmic

Layers -> Linear

Training HyperParameter Values can

Babysitting the variable or just having many computers testing variables in parallel

Normalizing (Batch Norm) (Quicker Traing and takes away Input Distribution)

Not just normalizing input but also all the layers

Normalize Z[i] before the activation function (there are varaibles gamma and beta)

Beta and Gamma are parameters

{beta[1], gamma[1],.....,beta[n], gamma[n]}

Each beta[i], gamma[i] are used for seperate layer

The parameter b[i] is not needed anymore, since we are subtracting the mean and changing it up by using beta[i] (zero's out the mean)

Input Values Changing Distribution

Normalizing speeds up learning

Batch norm works the same way for each layer but additionally:

Covariate Change: After training the model, there is a shift in the input distribution:

Coloured Cats vs Non-Coloured

Batch Norm keeps this distribution consistent (Mean and Variance Will stay the same)

Earlier Layers can't shift in distribution as much, since mean and variance at a later layer is always consistent, so the each layer applies its independent logic

Has a slight regularization effect:

Since: Each Mini Batch is being scales by the mean and variance of just that mini-batch

So it adds noise just like dropout to each hidden layer -> slight regularization effect

Test Time -> just one mean and variance so used a exponentially weighted average over your training mini-batches

Muti-Class Classification

Softmax regression:

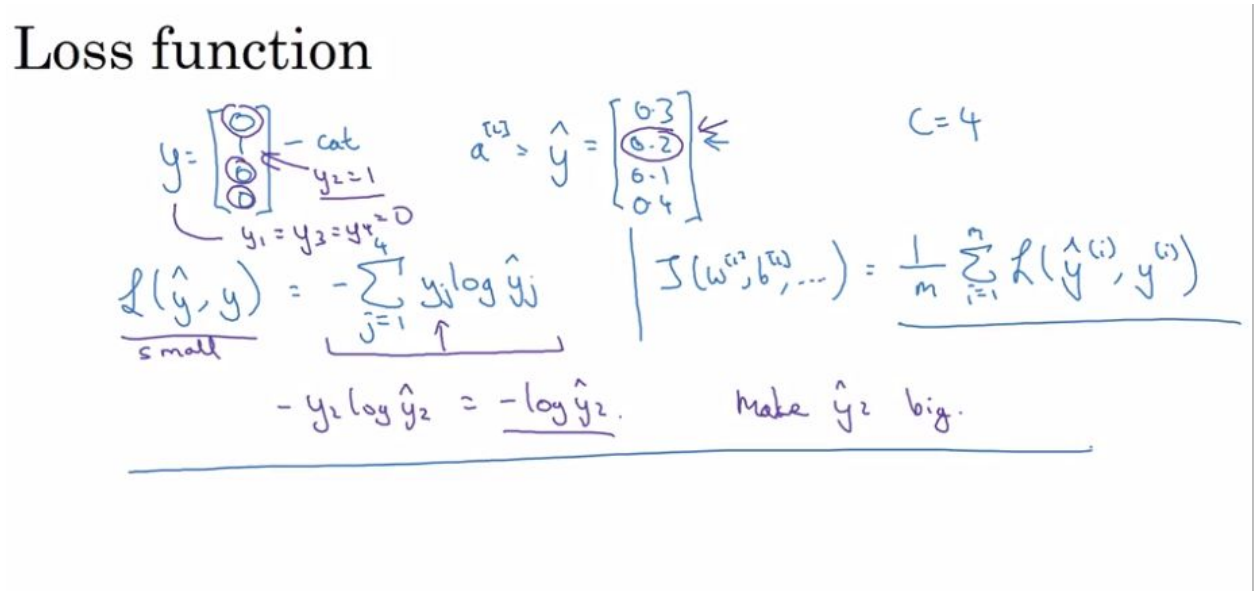
Just Has a Different Activation Function at the end

1) t = e^Z[i]

2) ti / Sum of all ti's

t is same dimension as Z and represents all the classes shape(Number of classes, 1)

MultiClass Loss Function:



Tensorflow does backprop but i have done a lot of the backprop myself using numpy

Tensorflow

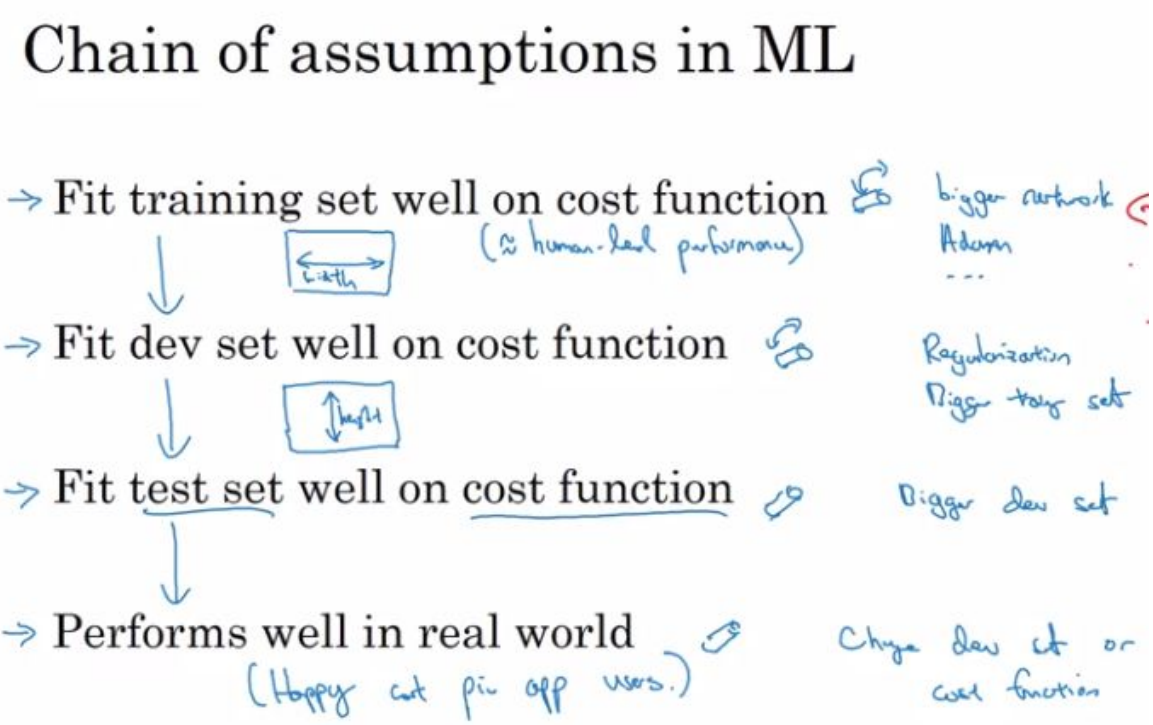
Minimize Cost

Structuring Machine Learning Projects

Orthogonalization:

Design the model so that each control, controls one variable (Car, TV Model)

ML Controls General:



Precision: Of the examples classified as cats, what percentage are cats

Recall: What percentage of the actual cats are recognized

Don't use both separately

get the harmonic mean together (F1 Score) take th best one, done in the dev set

Optimizing and Satisfying

Optimizing: Maximize variable

Satisfying: one or more (Having a threshold) less than 100

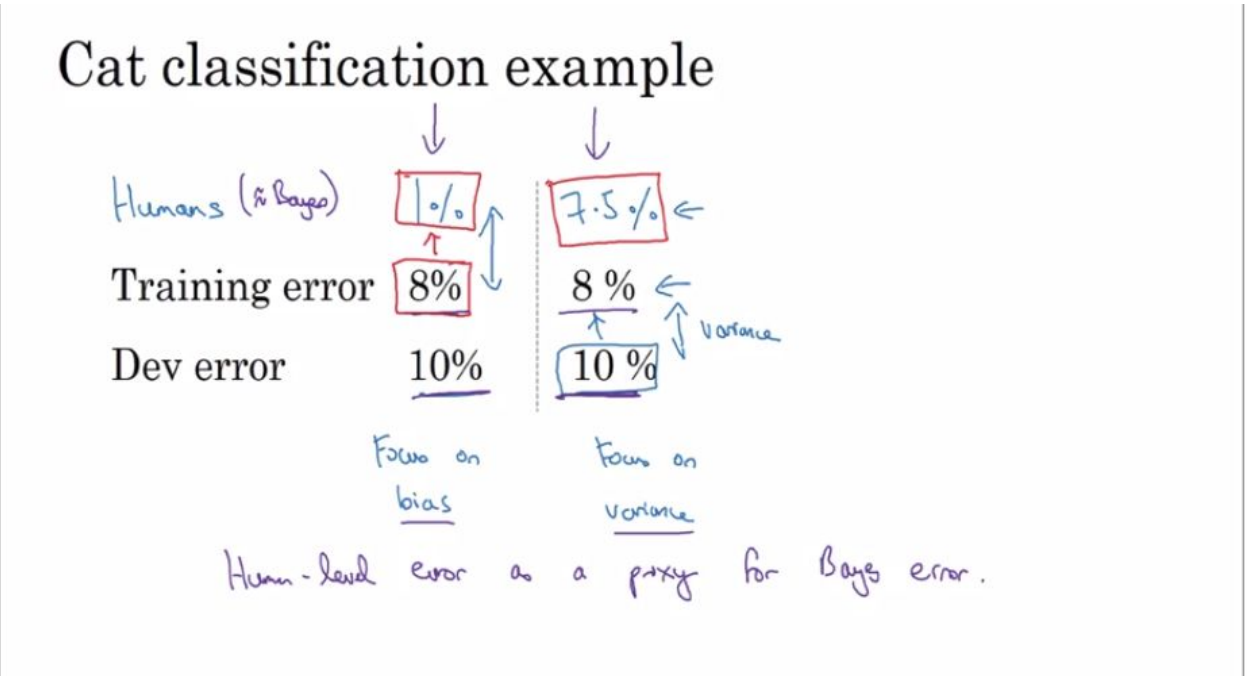
Setting Up Your Training / Dev / Test Sets

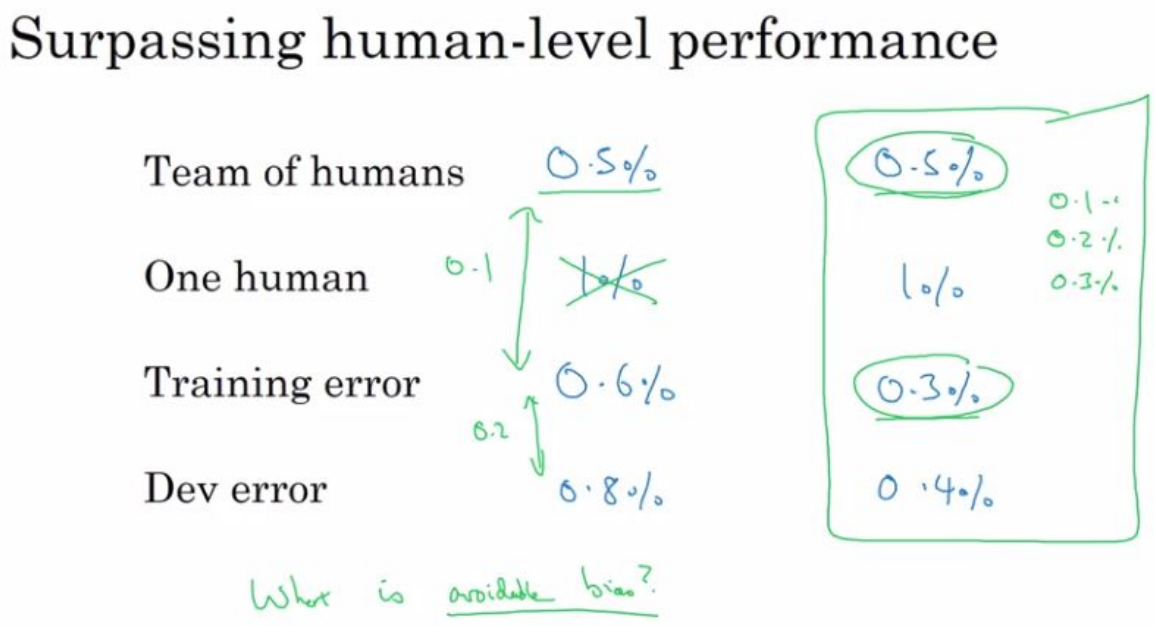
- Make sure your dev and test sets are from the same distribution
- Lots of Data today: So choose like a 1% dev/test set (millions of data)
- Tuning Set is the dev set
- Should have a test set -> no overfitting

How to improve algorithm? (Based on Human Error)

If your training set error is much larger than human error, clearly there is a bias, not training well, need to run gradient descent longer, Bigger Network.

If huge gap between dev and training -> regularization





Harder to tell if there is any bias on the right example cause we never know bayes error

Error Analysis

Is it worth reducing the number of dogs being detected as cats

Just go through all images the and calculate the percentage of images that were misrecognized by category and get the error for each category

If the error is huge, then focus on the category that produces the most error

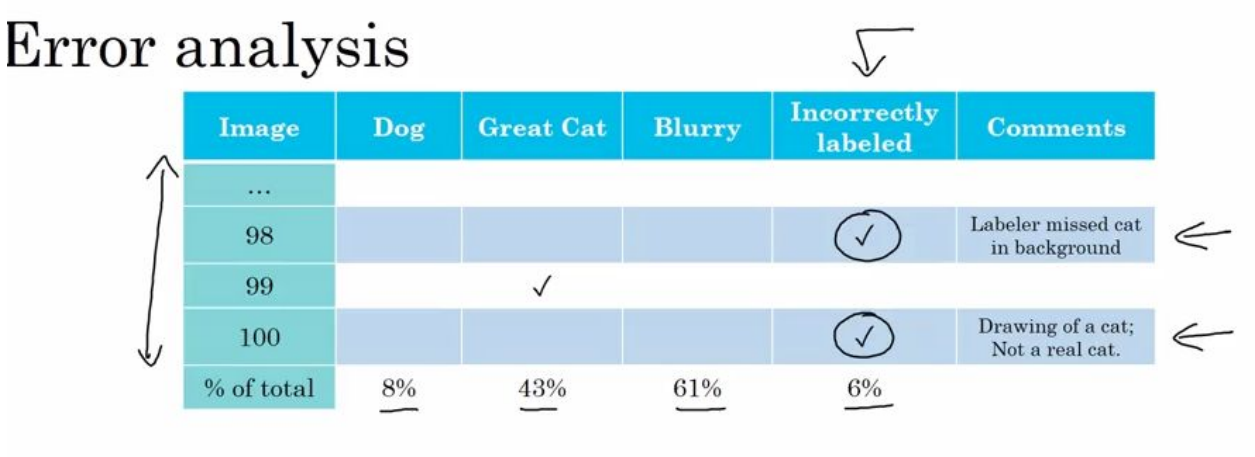
Incorrect Labelled Data

Deep Learning Quite robust to random errors in the training set

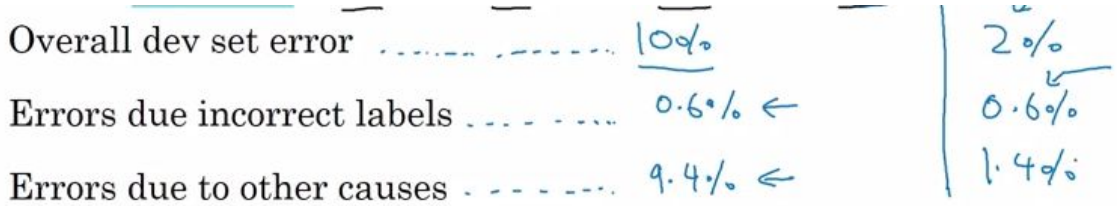
Not systematic errors

For the test and dev sets add another column to error analysis for incorrect labelles

% t



% total states the the % error in that category



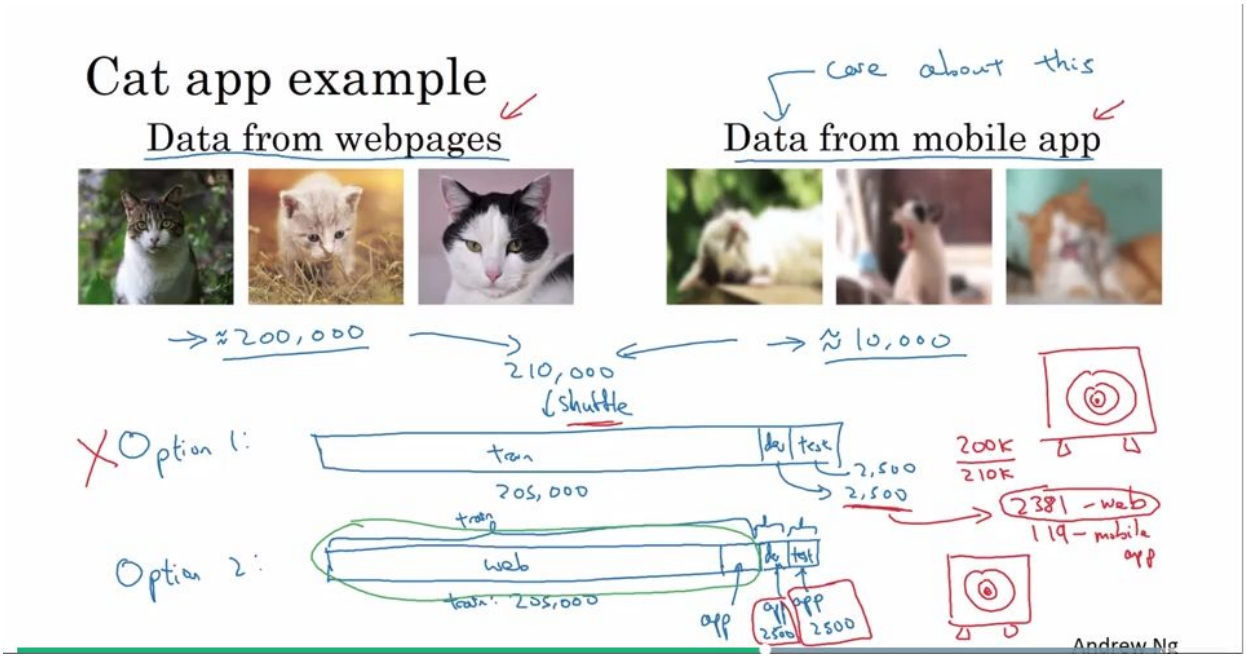
0.6% is nothing not worth fixing up

Correct dev/test sest same time -> continue to come from same distribution

It is okay to have different distributions in train and dev/test data

Build new system quickly then iterate

Training and testing on different distributions



Should add the target dev/test data into the training set, which also has data from a different distribution

Bias and Variance with mismatched data distributions

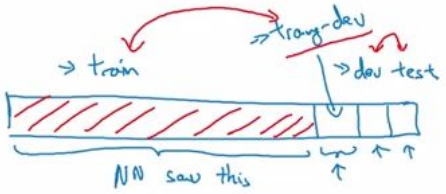
Training set and training-dev set have the same distribution

Cat classifier example

Assume humans get ≈ 0% error.

Training error 1%
Dev error 10%

Training-dev set: Same distribution as training set, but not used for training



Training error	1%	1%
→ Training-dev error	9%	1.5%
→ Dev error	10%	10%

Human error	0%	0%
Training error	10%	10%
Training-dev error	11%	11%
Dev error	12%	20%

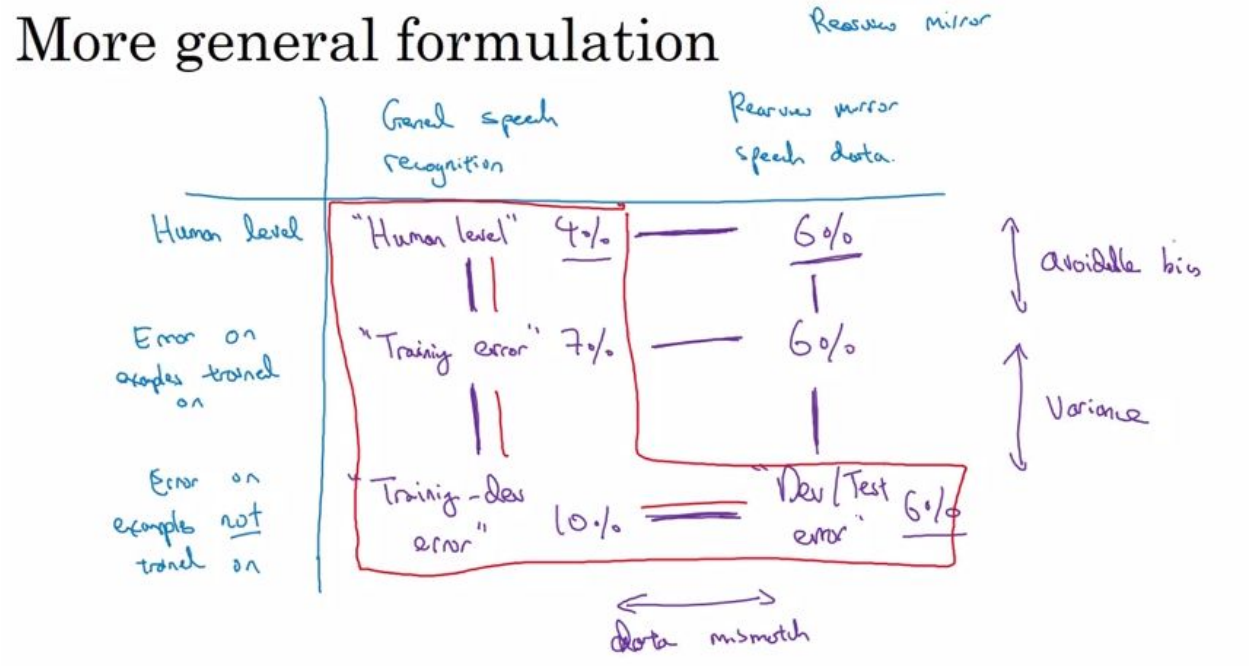
Andrew Ng

Human level
Training set error
Training-dev set error
→ Dev error
→ Test error

4%
7%
10%
12%
12%

avoidable bias
variance
data mismatch
degree of overfitting to dev set.

Purple overfit the dev-set



How to handle data mismatch??

Just collect more data relevant to the dev/test sets

Artificial Data Synthesis (Could Overfit)

Transfer learning used a lot!!!!

PreTraining -> Image Recognition

Fine Tuning -> the last layer and caulating cost

Used when you have a lot of data from the problem you are transferring from and less data for the problem your transferring too.

Image recognition to radiology system

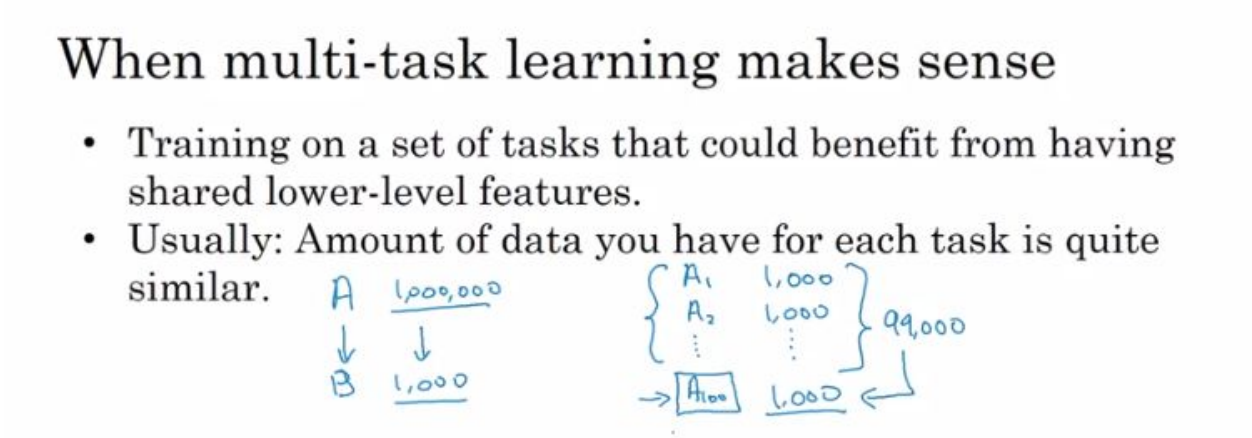
Speech recognition and drive thru communication

Multi-task learning

Multiple labels to one image

Solving multiple problems at the same time

Using one neural network to do multiple tasks (labels) is effective by sharing low level features.



Creates more data

Multitasking hurts performance if neural network is small, otherwise is very helpful

Used more in computer vision

End to End Deep Learning

Need more Data

Worked well for machine translation english to french.

Not always the best idea

face recognition

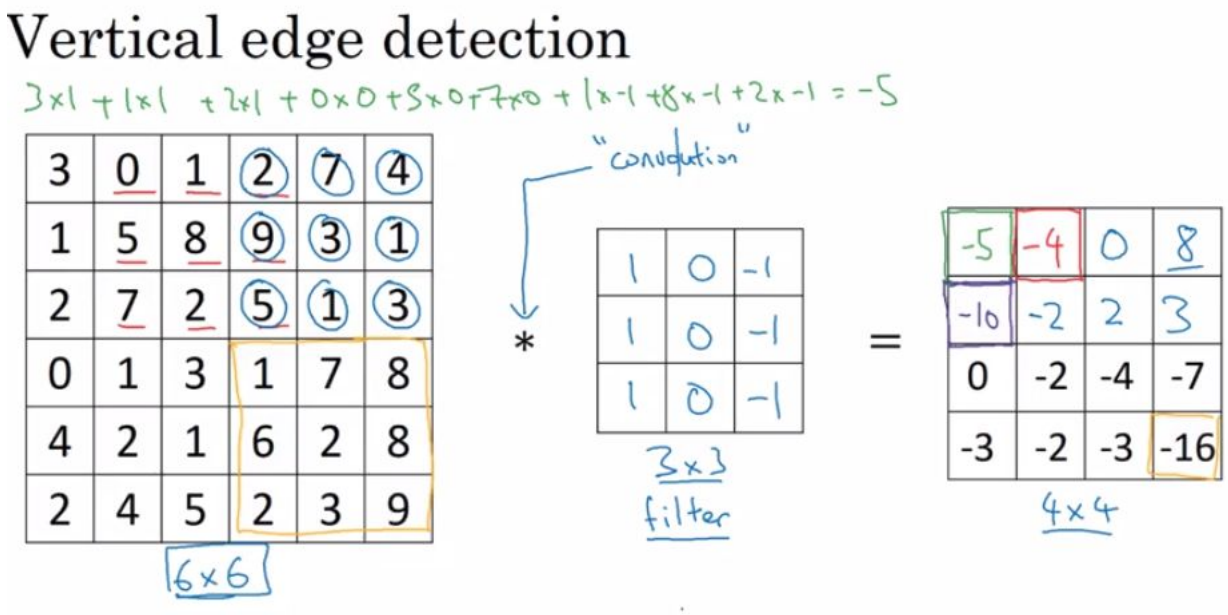
2 step approach is better -> a lot more data for individual problems

Estimation child's age through bone structure

Image -> bones -> age is better than image -> age

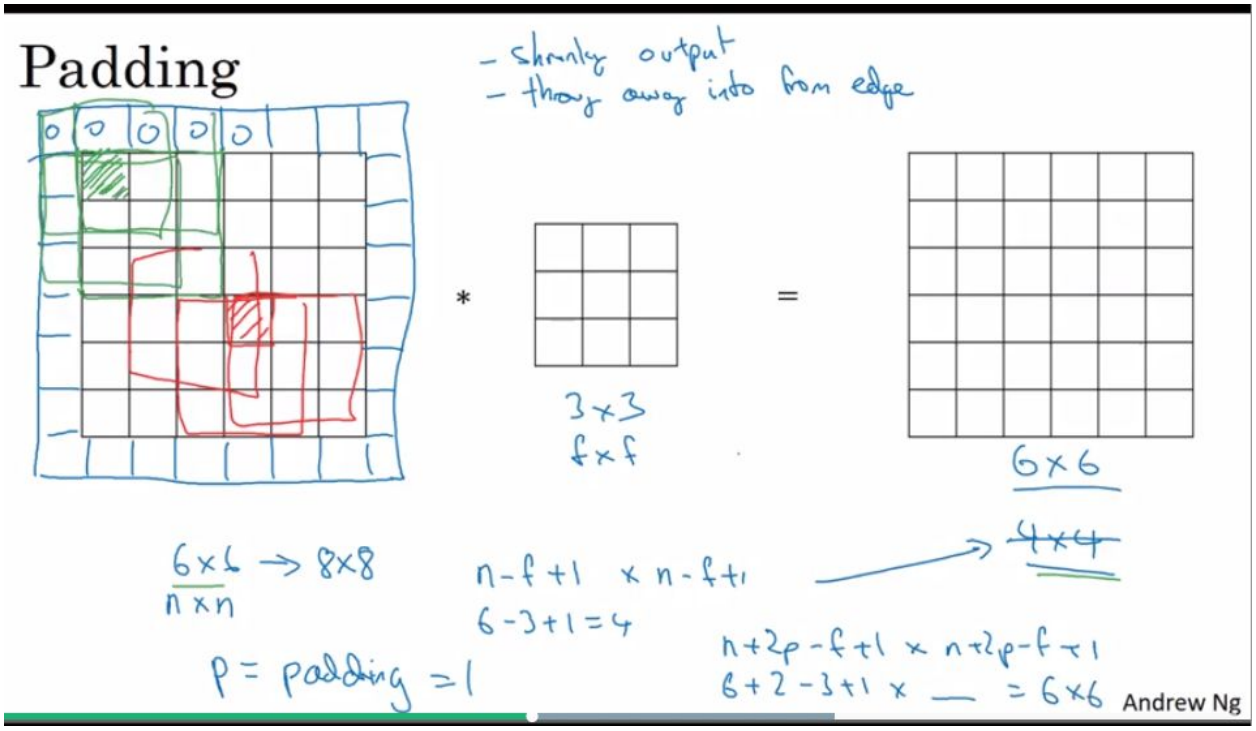
Excludes potentially useful hand-designed components, giving it manual knowledge by breaking it into smaller pieces, but sometimes those manual pieces of knowledge can be flawed like phonemes before words. (where computer is better of learning its own way)

COURSE 4: CONVOLUTION NEURAL NETWORKS



In CNNs, filters are not defined. The value of each filter is learned during the training process.

By being able to learn the values of different filters, CNNs can find more meaning from images that humans and human designed filters might not be able to find.



Padding

Another name Same Convolution (Same size output)

No padding => valid convolutions P = 0

Good for two reasons

1) Just so the corner pixels count the same
2) And the output after the convolution step stays the same size

Done: Adding zeros around the initial image pixels (1 layer on each side)

Padding Size:

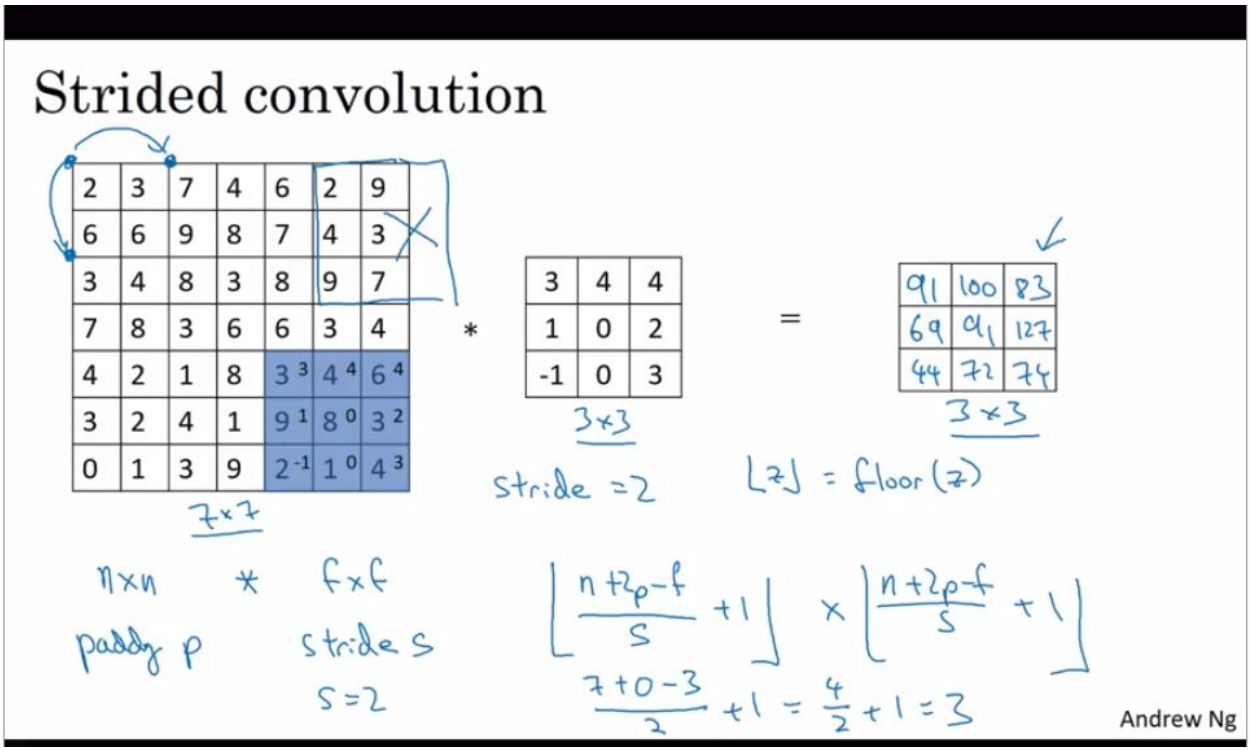
P = (f -1) / 2

f is odd (it is the size of the filter)

P is the padding

Stride Convolution

Skipping the filter more than once



Equation below

Summary of convolutions

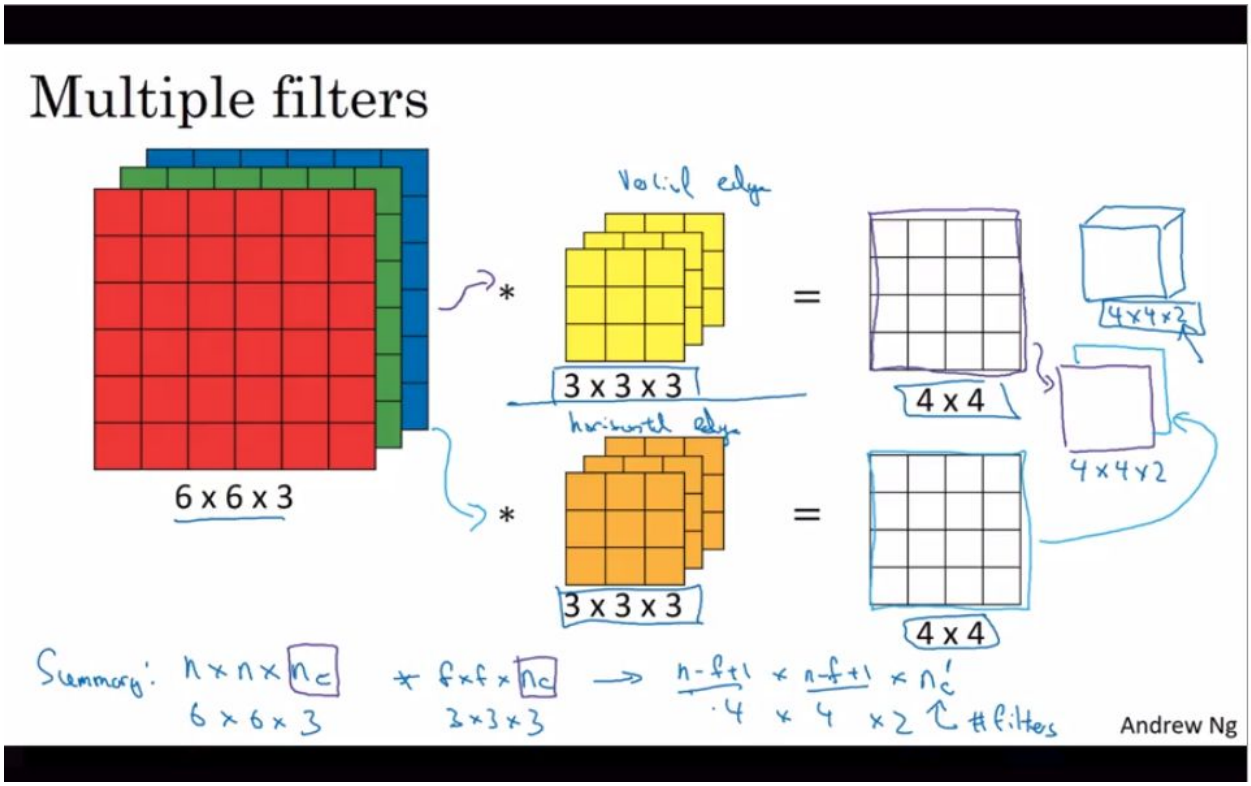
$n \times n$ image $\quad f \times f$ filter

padding p $\quad$ stride s

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

Don’t need to flip the filter: just avoid in neural networks, only used in signal processing

Convolution on Volume:



nc' above is the number of filters

<u>Convolution Networks Connected to Forward Propagation in Deep learning</u>

X: The image pixels

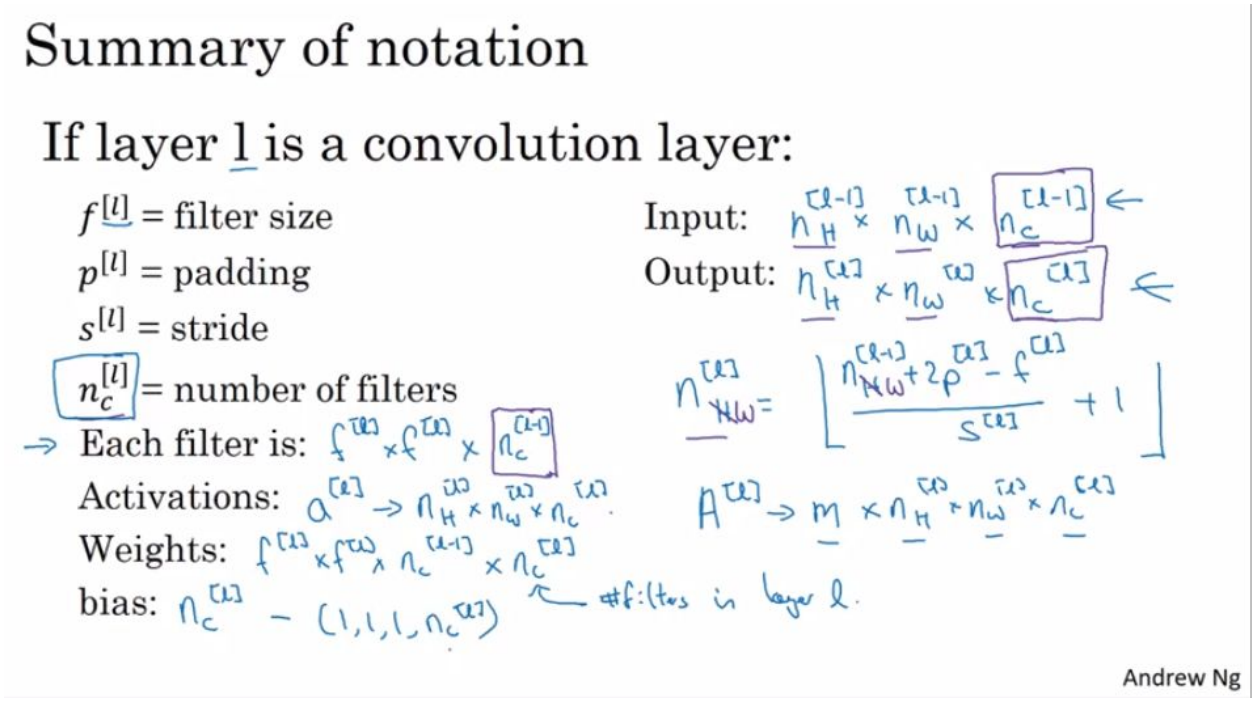
W: Filter

B: constant

Z = ((X * W) + B) ->>> the * is not multiplication but the convolution operation

A = Relu(Z)

Notation Convolution Neural Networks



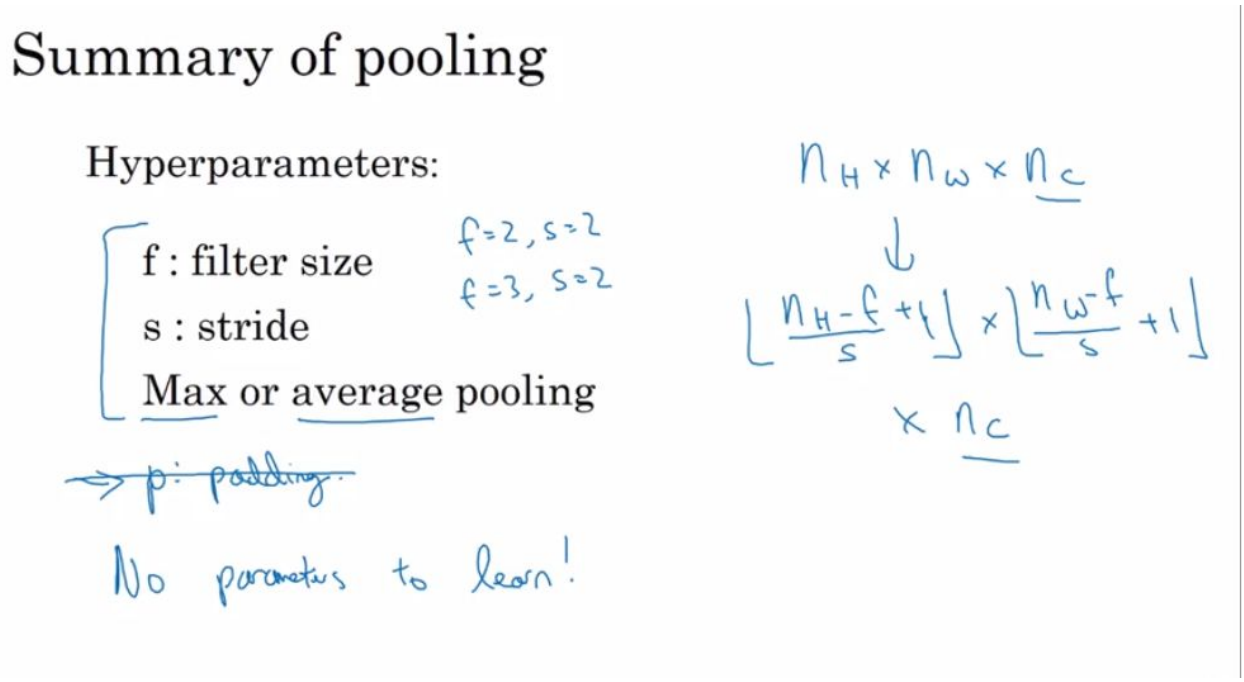
Pooling / Average Pooling

Usually using f = 2 and s = 2

Just finding max number in the filter size f

Same formula

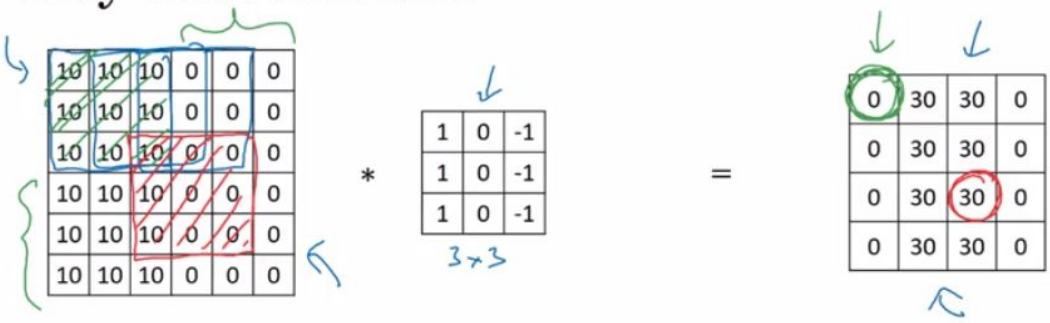
((N + 2p - f) / s) + 1



No parameters to learn in pooling!

Why less parameter

Why convolutions



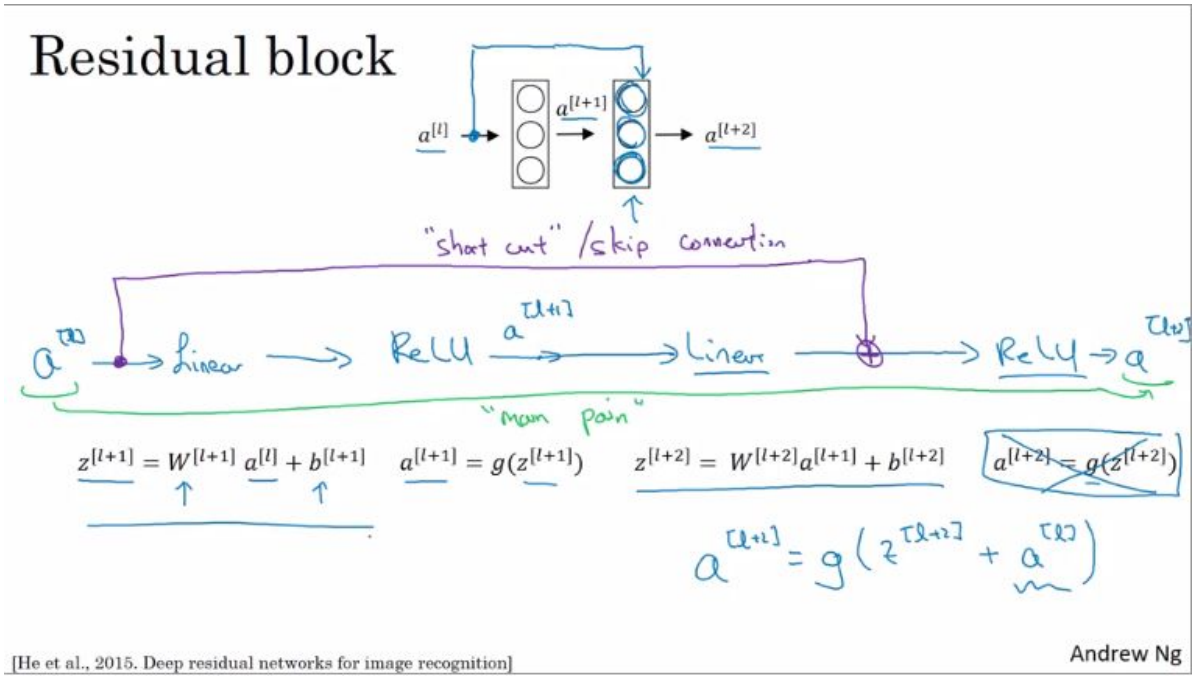
Parameter sharing: A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.

Sparsity of connections: In each layer, each output value depends only on a small number of inputs.

Andrew Ng

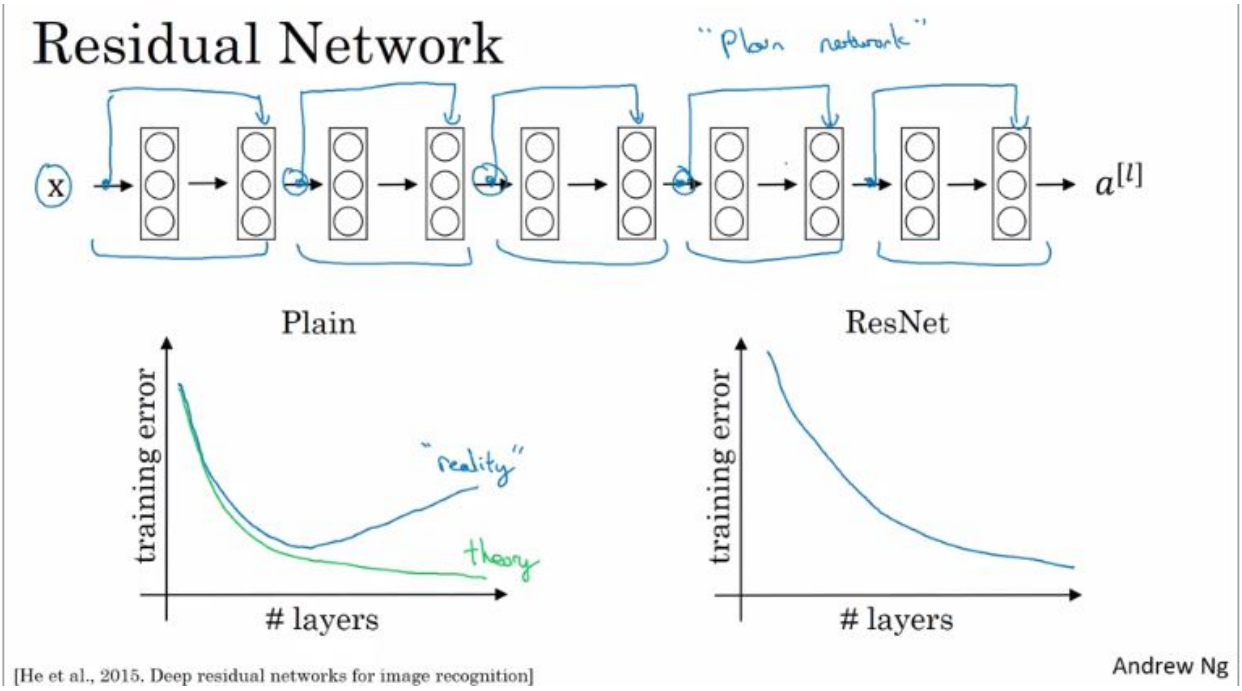
ResNets

Residual Block:



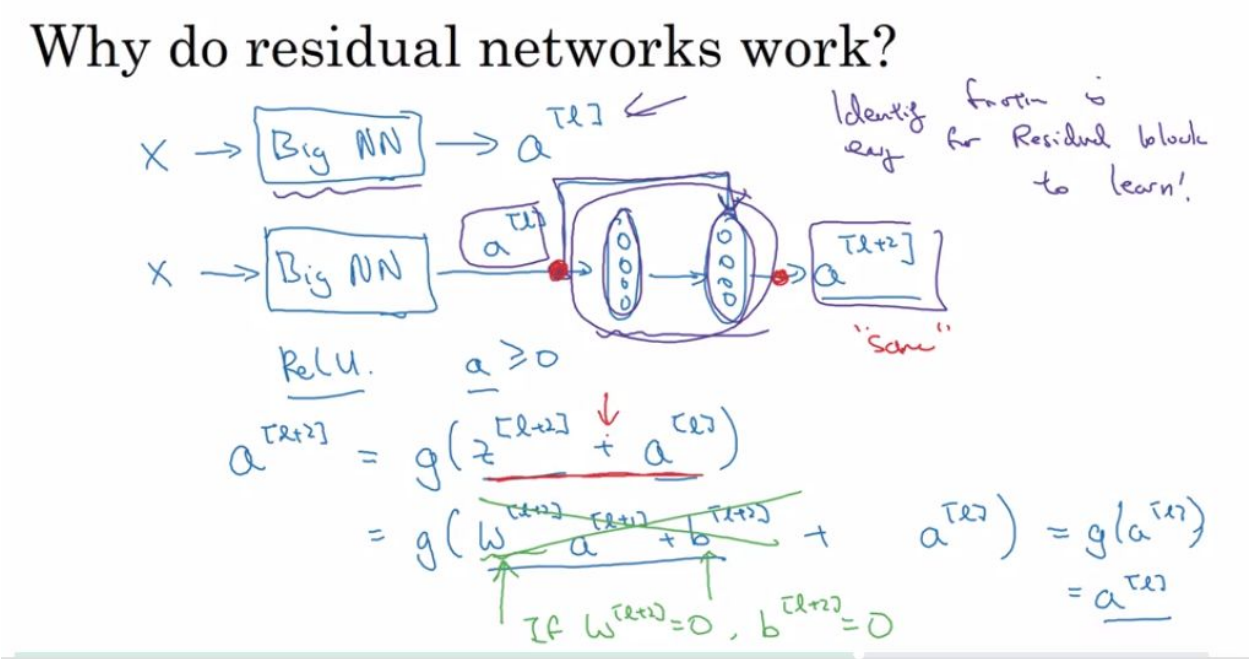
Residual network:

In reality when your network gets too deep your training error follows the blue curve, that's why we add residual blocks. Training acts like theory



Why residual networks work?

Because of l2 regularization/ weight decay W[l+1] and b[l+1] become smaller close to zero which means you just have a[l] left and g(a[l]) is just a[l] so the extra layers don't affect anything in this case.

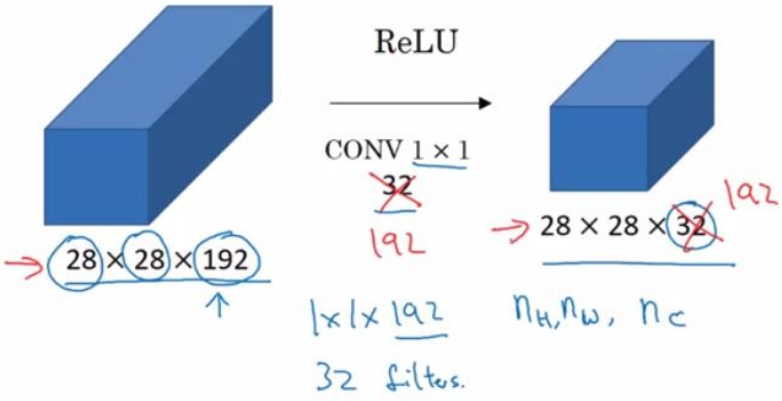


A lot of same convolutions adding padding so g(z[l+2] + a[l]) is possible that you can z and a's have the same matrix shape

1 x 1 convolutions

1 x 1 convolutions help you shrink the number of channels

Using 1×1 convolutions

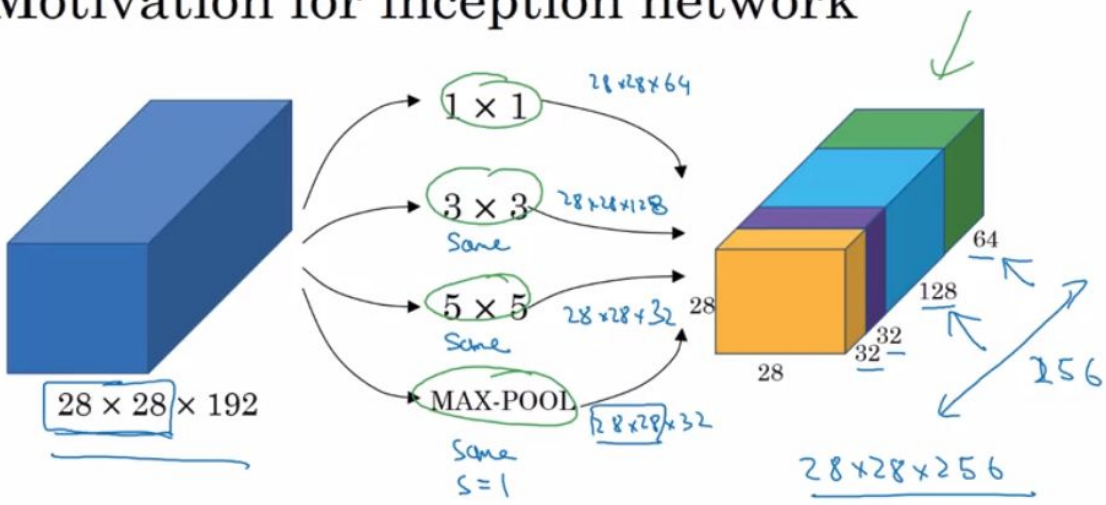


[Lin et al., 2013. Network in network]

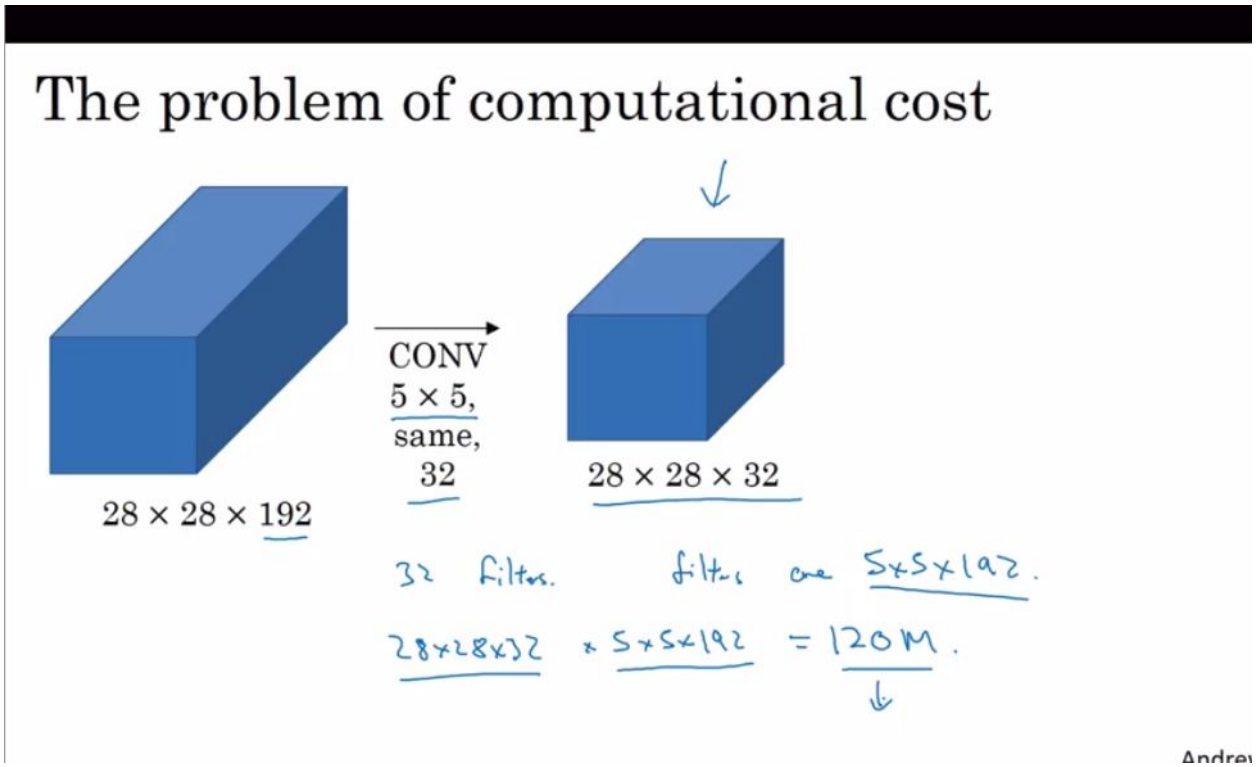
Andrew Ng

Inception Network

Motivation for inception network

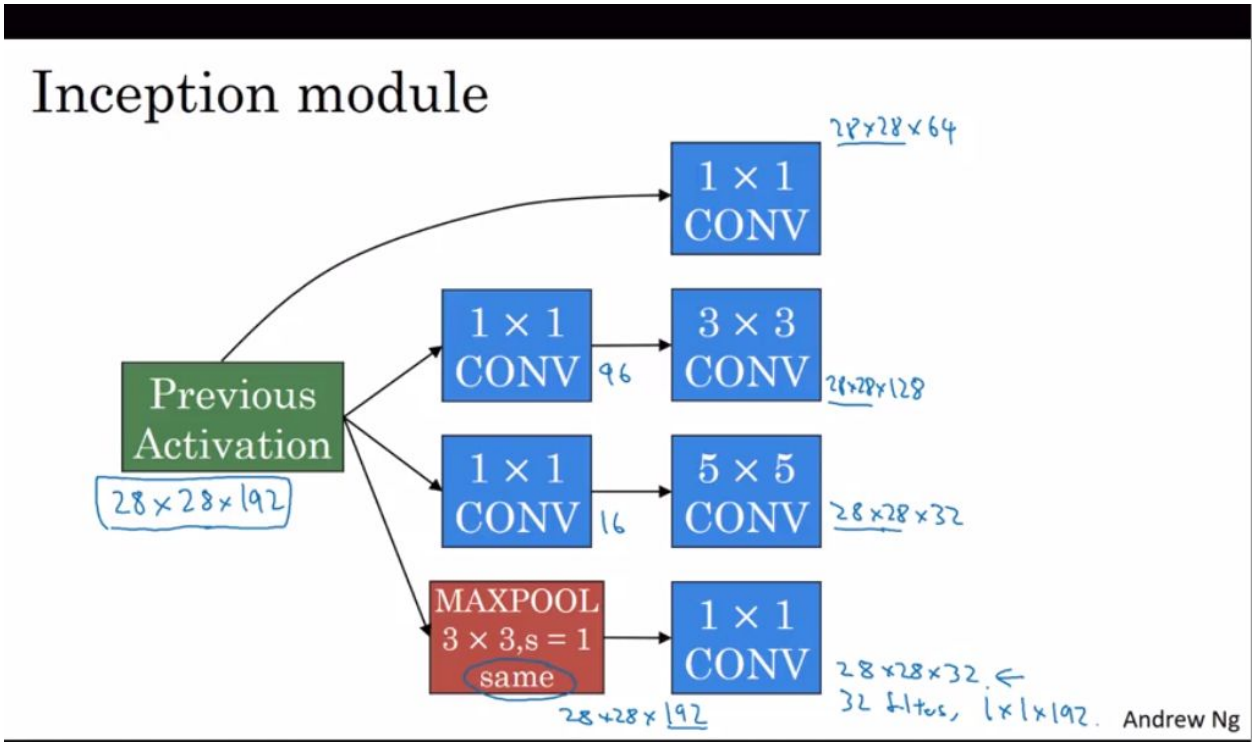


[Szegedy et al. 2014. Going deeper with convolutions]



1 x 1 filter can resolve the crazy amount of calculations we need to do

So that for inception model you can do a 1 x 1 conv between (bottleneck) the actual f x f calculation the the 1 x 1 x channel helps you reduce the the channels in the original volume



Inception network / googleLeNet

Use Open Source Projects

- to get the architecture for your project
- And to get the pretrained weights (transfer learning)

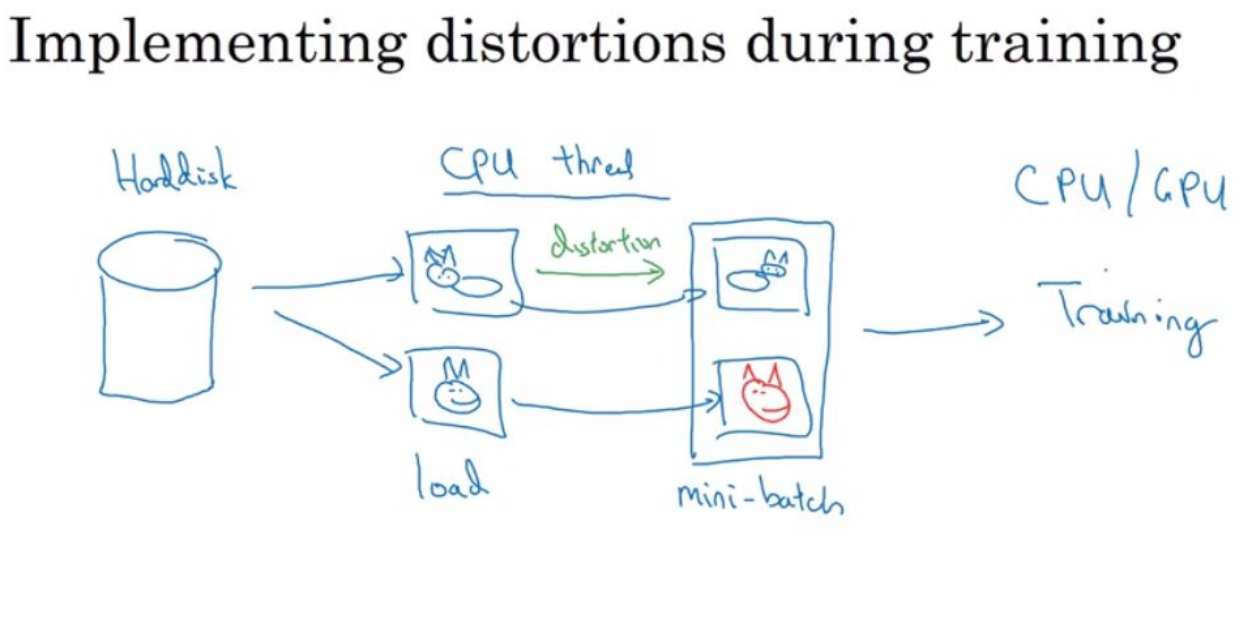
For Computer Vision Use Transfer Learning, already pre trained models on huge data sets

<u>Data Augmentation</u>

Very useful for computer vision

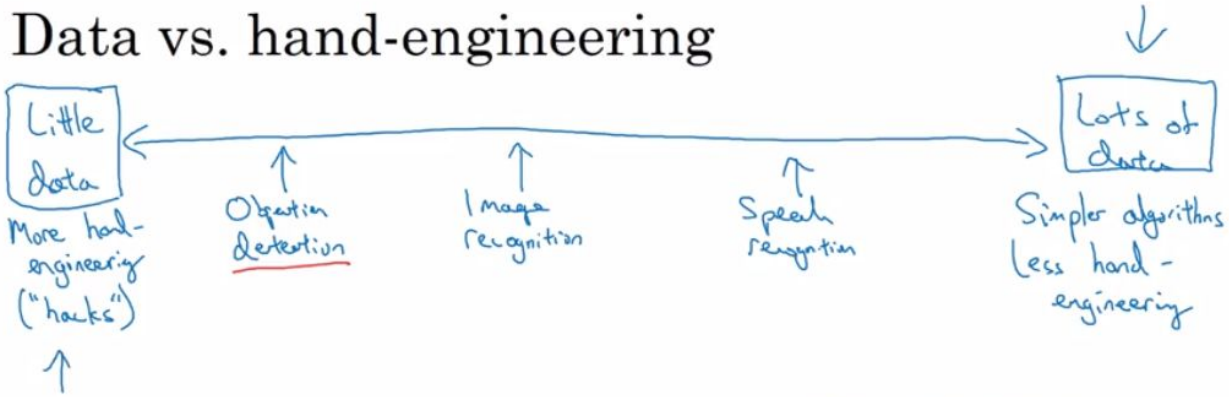
- Mirroring
- Random Cropping
- Shearing
- Rotation

- Colour shifting (Changing the RGB with constants)
-



CPU thread continually brings in images and then the cpu/gpu and the cpu thread run in parallel

Data vs. hand-engineering



Two sources of knowledge

- Labeled data (x, y)
- Hand engineered features/network architecture/other components

Andrew Ng

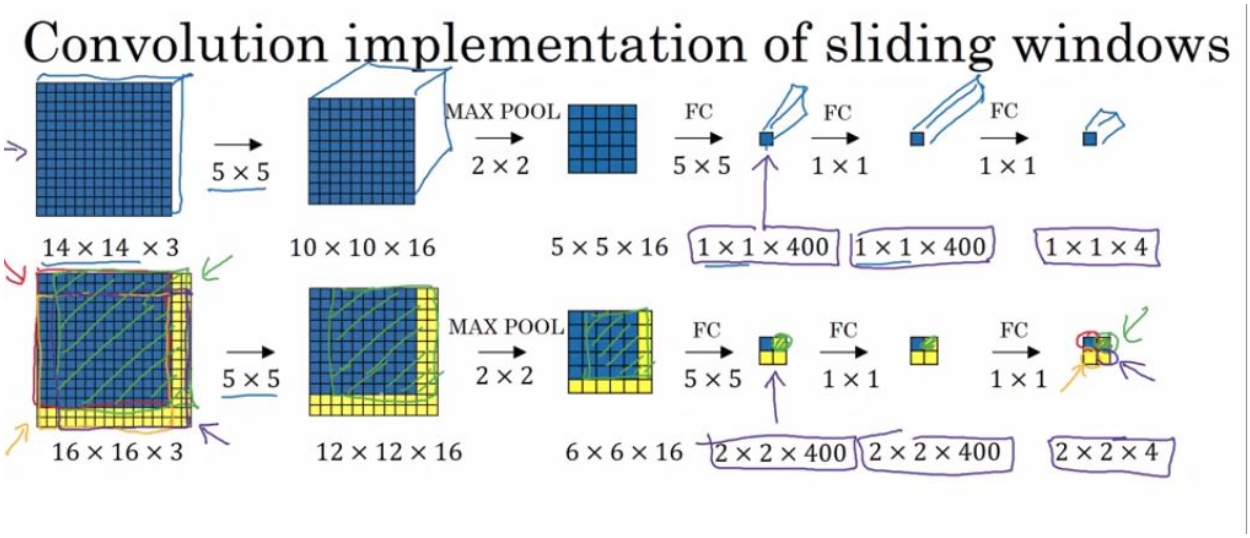
Use open source code

- Use architectures of networks published in the literature
- Use open source implementations if possible
- Use pretrained models and fine-tune on your dataset

Localization -> Detecting where object is in the picture

BEST ALGORITHM IS THE YOLO algorithm for detection

Sliding Windows

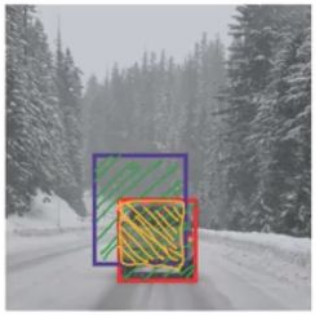


The above shows how to make computational cost lower, can just run the convolution layers on the image, and the final result (2D) will tell your where the object is

Since the 14 x 14 x 3 input gives you a 1 x 1 output in the original convolution layers, any image you put has to be bigger than 14 x 14 x 3 and the sliding window size now is 14 x 14 x 3

Disadvantage the sliding window is fixed size

Evaluating object localization



Intersection over Union (IoU)

$$= \frac{\text{size of } \square}{\text{size of } \square}$$

"Correct" if IoU $\geq$ 0.5 ←

0.6 ←

More generally, IoU is a measure of the overlap between two bounding boxes.

non -Max Suppression Algo

Non-max suppression algorithm

19× 19

Each output prediction is: p_c, b_x, b_y, b_h, b_w

Discard all boxes with $p_c \leq 0.6$

While there are any remaining boxes:

- Pick the box with the largest p_c Output that as a prediction.
- Discard any remaining box with IoU ≥ 0.5 with the box output in the previous step

Andrew Ng

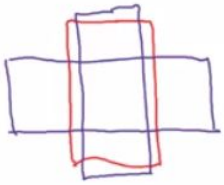
Anchor Box For Multiple Object Detection

Anchor box algorithm

Previously:

Each object in training image is assigned to grid cell that contains that object's midpoint.

Output y: $3 \times 3 \times 8$



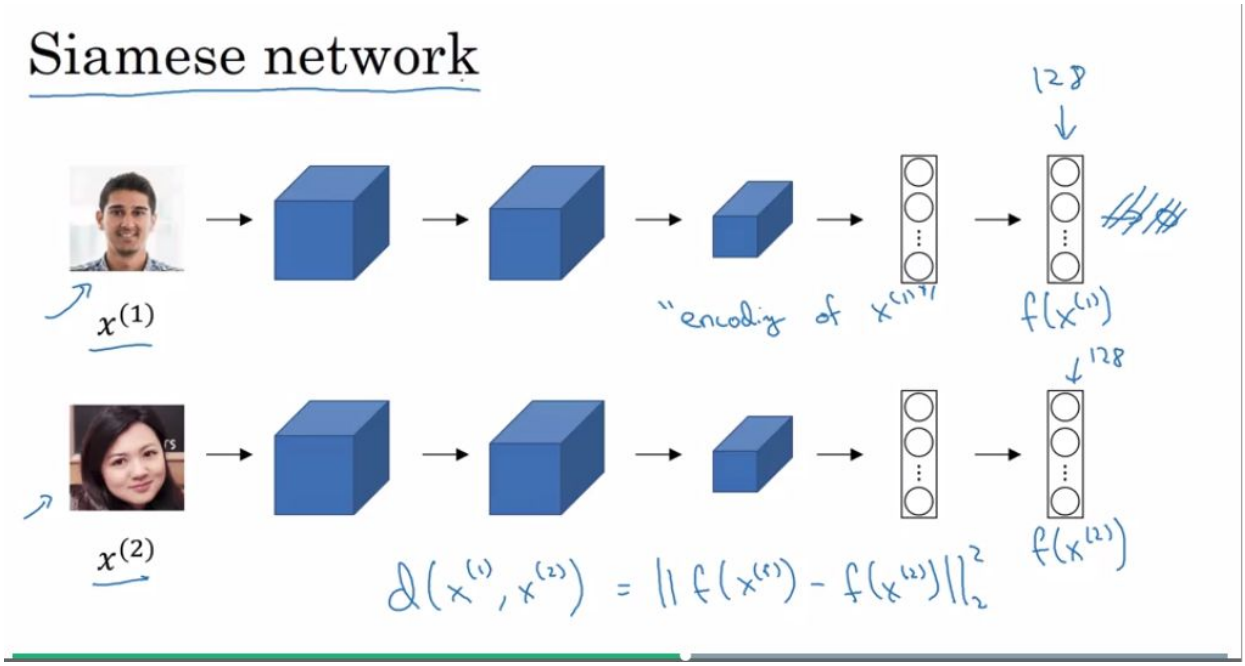
With two anchor boxes:

Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU.

(grid cell, anchor box.)

Andrew Ng

FACE RECOGNITION



A way determining how similar the faces are using a one shot approach. (only one picture available)

The 128 dimension output are the ecodings

Triplet Loss Function:

Basically comparing a positive match and a negative match.

Getting a difference function for both positive and negative match.

Then applying alpha to make sure there is a certain margin of difference between them

Need to make sure that the negative match also has a small difference function, so you can train harder pictures and get a better result

Face Verification and Binary Classification

Just compare two cnn output from two different images and label the images a 0 or 1. (Logistic Regression)

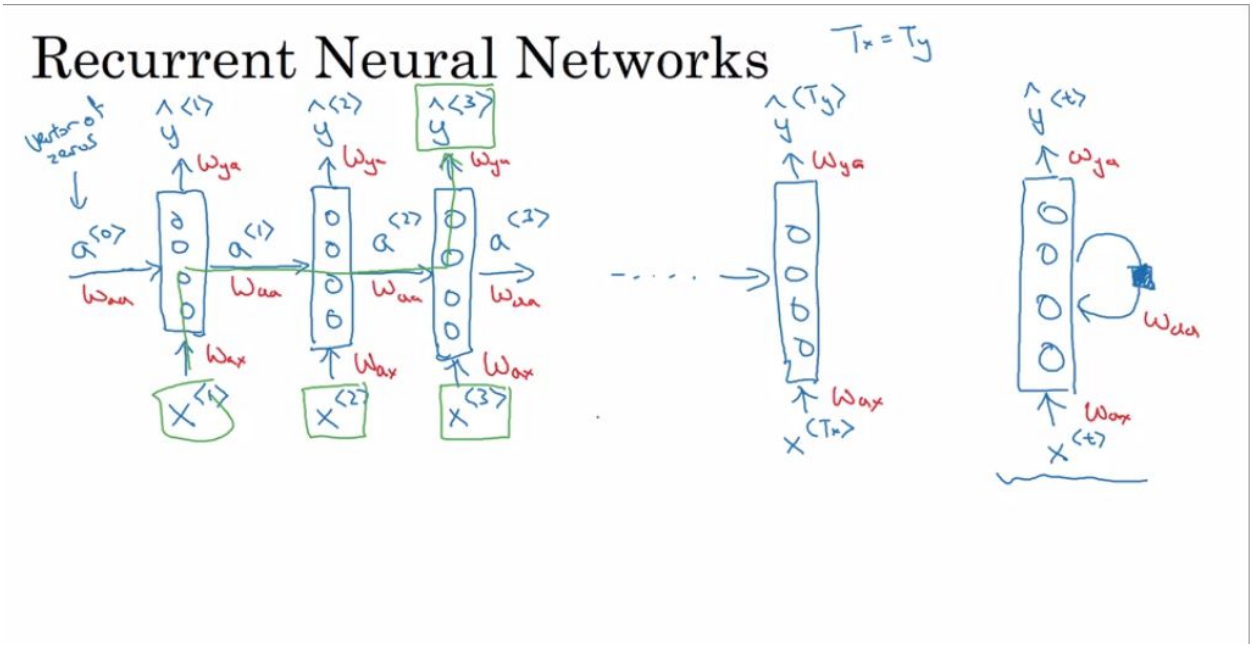
What is neural style transfer?

Combining one picture's content and another picture style

Content is just the difference between the activations for a layer in content picture and the final picture

Style has a G style matrix for picture S the style picture and G the final picture and figures out the their difference for the cost function

Sequence Models



Can have many relationships depending on the size of input and output

One to One

One to Many (Music Generation)

Many to One (Movie Rating)

Many to Many (2 situations equal length and different size lengths)

Exploding Gradients:

Gradient Clipping: Just rescale

Vanishing Gradients Problem

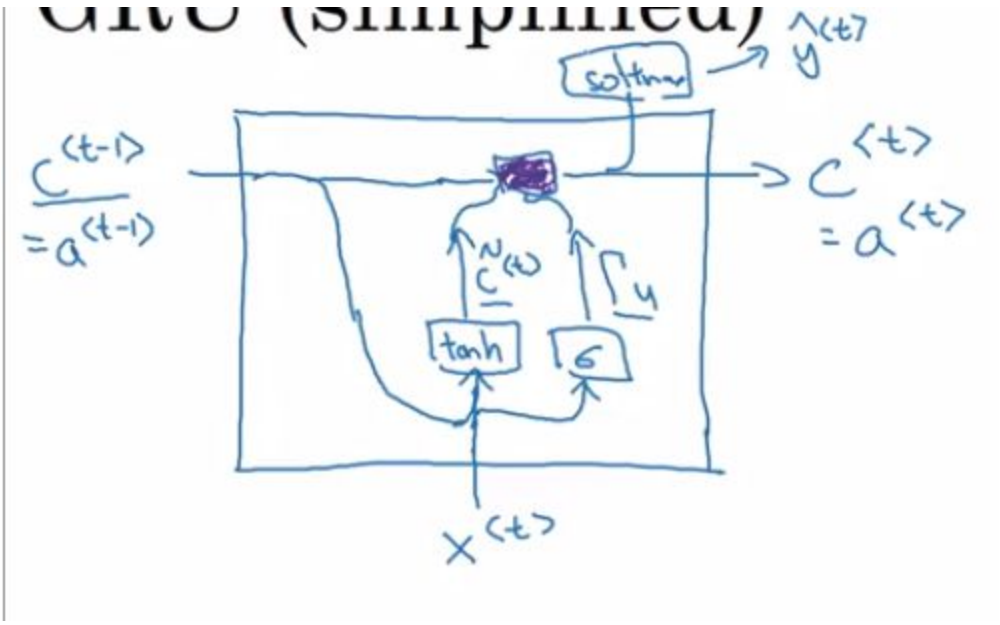
RNN is not very good at catching long term dependencies

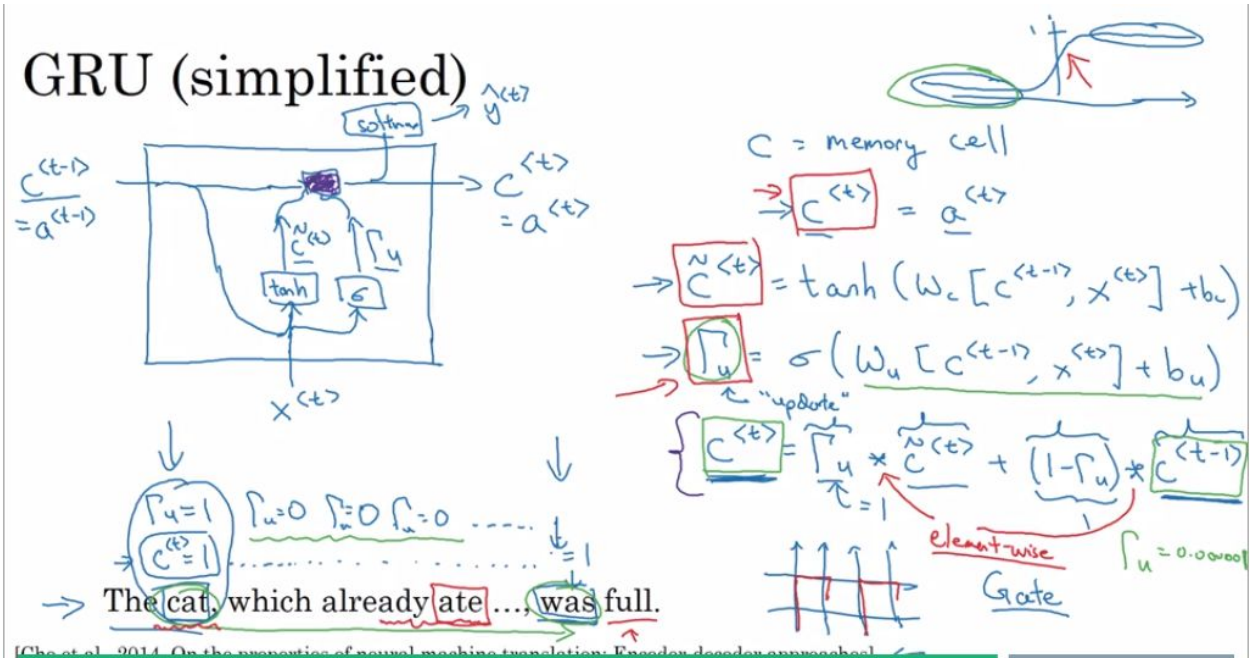
Cats …….. Were

Cats …….. Was

Many local layers highly effect the the output values (Because its very hard for an error to backprogragate long distances away. (long distances mean lots of layers)

Solution: Gated Recurrent Unit (GRU)





Ct.Gamma and Ct~, is actually a vector with multiple bits representing different facts about a sentence

Ct~ is for what facts the current words adds to the sentence, the gate filters out what info is important what is not okay to update

GRU

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[\, c^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_r = \sigma(W_r[\, c^{<t-1>}, x^{<t>}] + b_r)$$

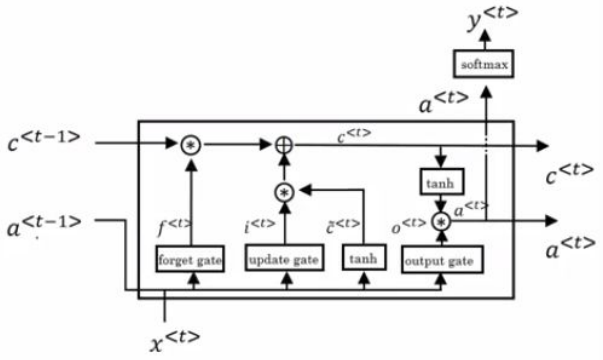
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$

$$a^{<t>} = c^{<t>}$$

Another Solution: LSTM

LSTM in pictures

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$
$$\Gamma_u = \sigma(W_u[\, a^{<t-1>}, x^{<t>}] + b_u)$$
$$\Gamma_f = \sigma(W_f[\, a^{<t-1>}, x^{<t>}] + b_f)$$
$$\Gamma_o = \sigma(W_o[\, a^{<t-1>}, x^{<t>}] + b_o)$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$
$$a^{<t>} = \Gamma_o * \tanh c^{<t>}$$



Bidirectional RNN

NLP: Uses Bidirectional RNN + LSTM

However for real time conversation you would have to wait until the person stops talking to start the algo, not very effective!

Deep RNN

Not many layer as CNN

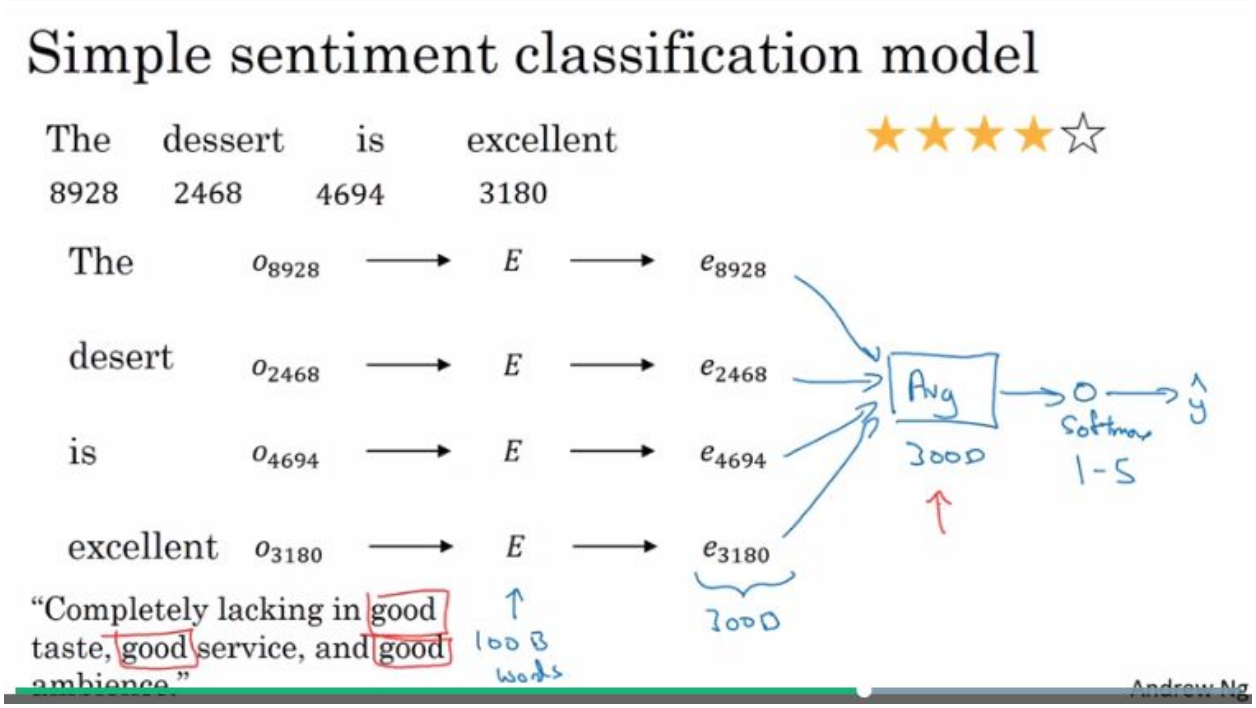
Word Embedding and Transfer Learning

Can be useful for transfer learning when you have a small data set

Application of Sentiment Classification Model

Word embeddings

“Can’t just take the average of all your word embeddings”,



Apply RNN “not good” does not mean “good” a sequence comes to affect here

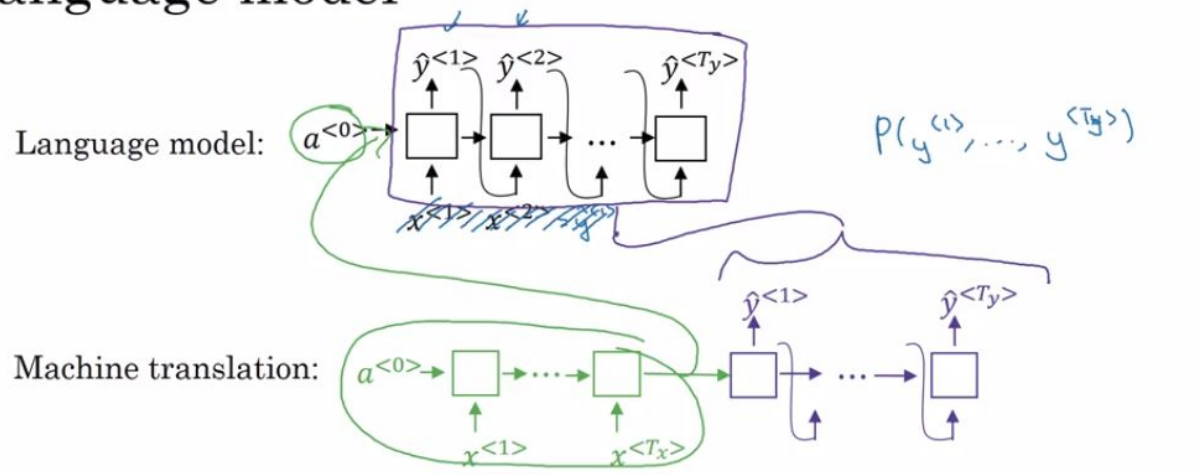
Emojify/Sentiment Analysis

All about using a Embedding vectors, training them if large data set, otherwise use transfer learning and applying it to a sentiment analysis. Using LSTM to keep a sequence words actual meaning.

Sequence to Sequence Models

Image to to caption -> google images

Machine translation as building a conditional language model



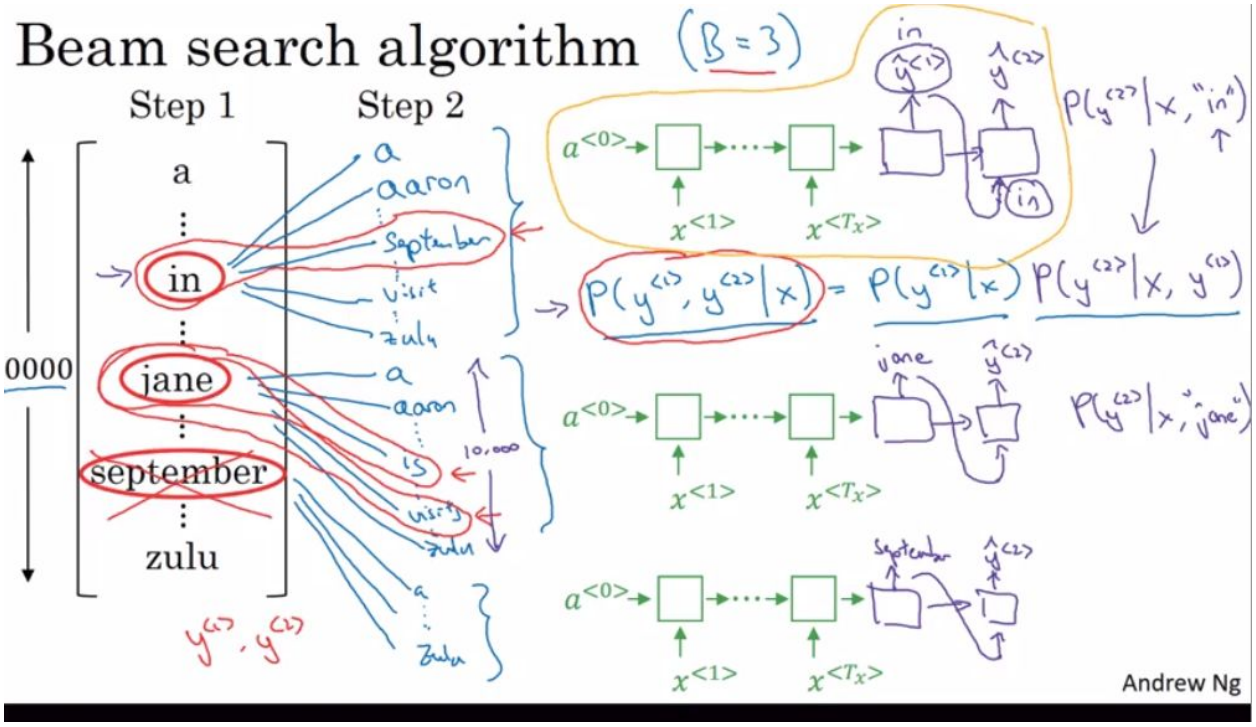
Machine translation is a conditional language model P(y1...yTy|x1….xTx)

Greedy Approach doesn't work, since some words might be used more more, but another word makes sense in the sequence

<u>Beam Search Algorithm</u>

Beam = # of groups you can select being mostly likely

1. Select beam number of words most likely for first word
2. Then for each beam # of words look at at the next possible word options
3. select beam # of groups of 2 words that end up with greatest probability
4. Then repeat again for the beam # of groups to look at at the next possible word options



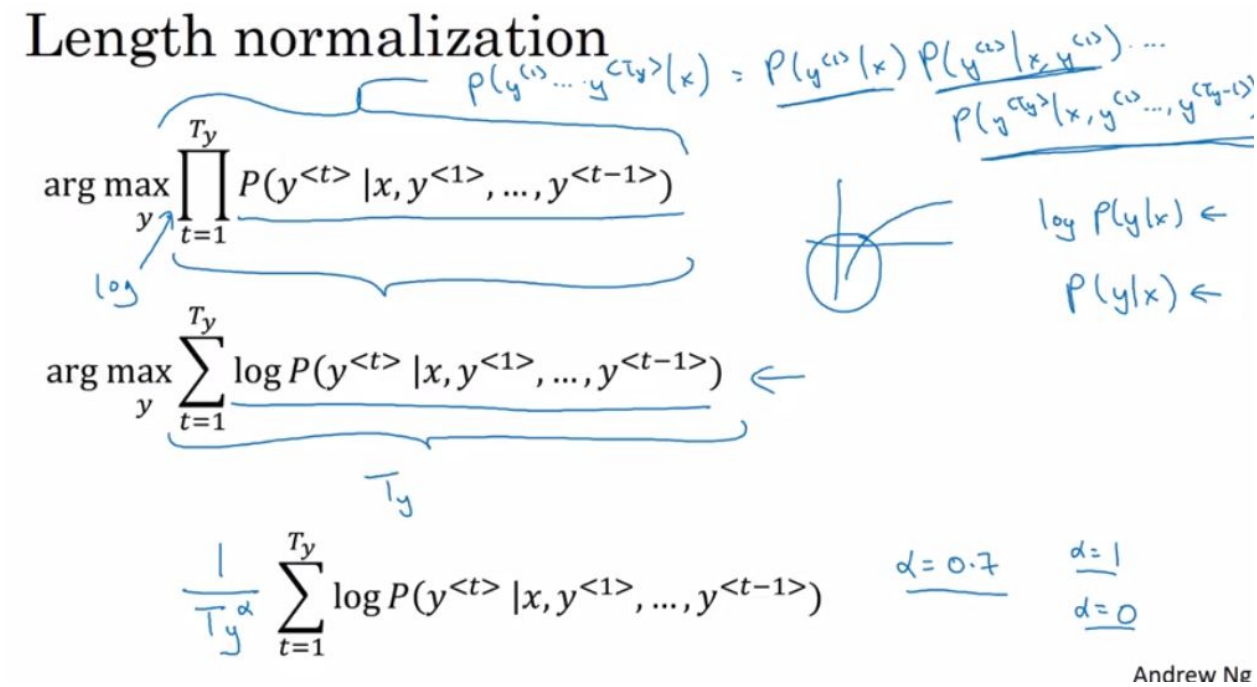
If beam # = 1 -> that's just greedy search algo

Fixing Issues in Beam Algo Underflow

Since multiplying very small numbers various times for the probabilities could result in an underflow, we must log each probability

Beam Algo likes shorter setences since probability is higher

Just normalize it, divide by the number of words in the sequence



Larger Beam -> Slower -> better outcome tho

Small Beam -> Faster -> worst outcome

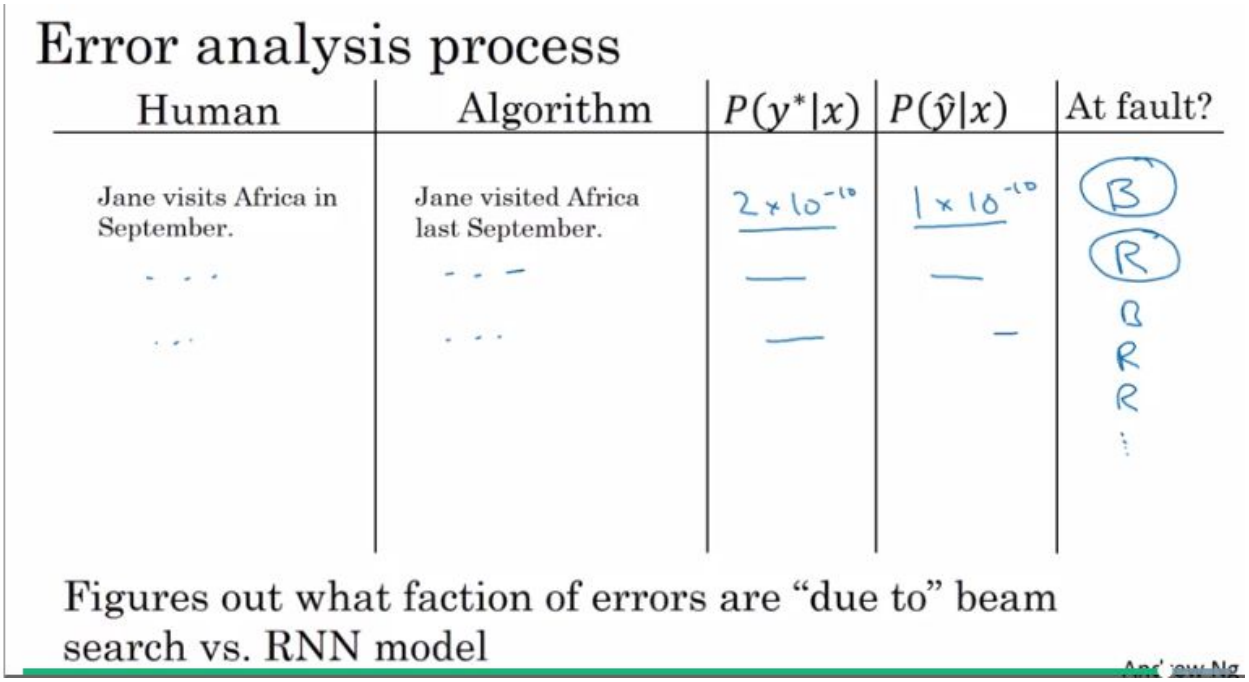
Error analysis in beam search

I beam is at fault -> increase beam number

If RNN is at fault -> add regularization, change architect, more data

y* being the human output

y^ being the machine output



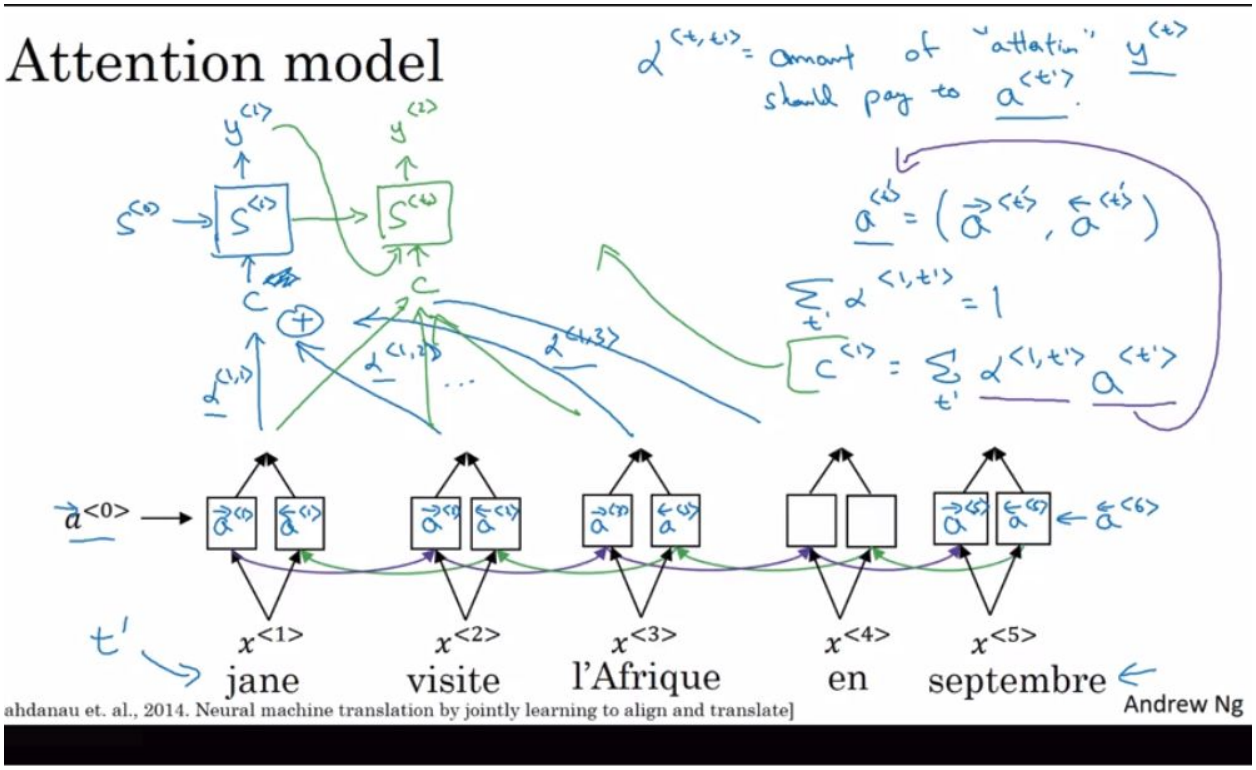
Evaluating how good the machine output is compared to the human output

Called the Blue Score

BP penalizes shorter length sentences, cause easier to get a high blue score

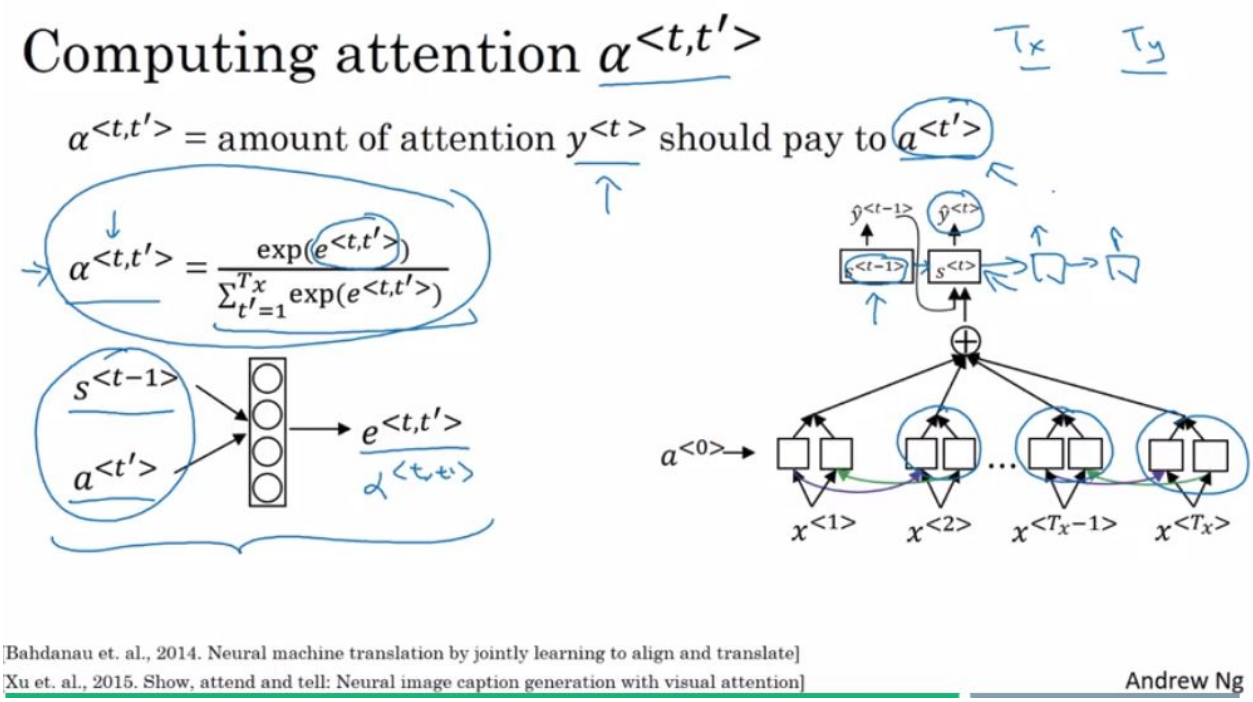
Networks have a hard time memorizing long sentences so we use the attention model

Attention Model: Audio Data



ahdanau et. al., 2014. Neural machine translation by jointly learning to align and translate]

- Bidirectional RNN's at the bottom
- The alpha represents how much attention each output should give to each word.
- Alpha's sum to 1 for each output



[Bahdanau et. al., 2014. Neural machine translation by jointly learning to align and translate]
[Xu et. al., 2015. Show, attend and tell: Neural image caption generation with visual attention]

Andrew Ng

Quadratic Cost

Speech Recognition

- Trigger Words are just a long RNN that output 0's and 1's