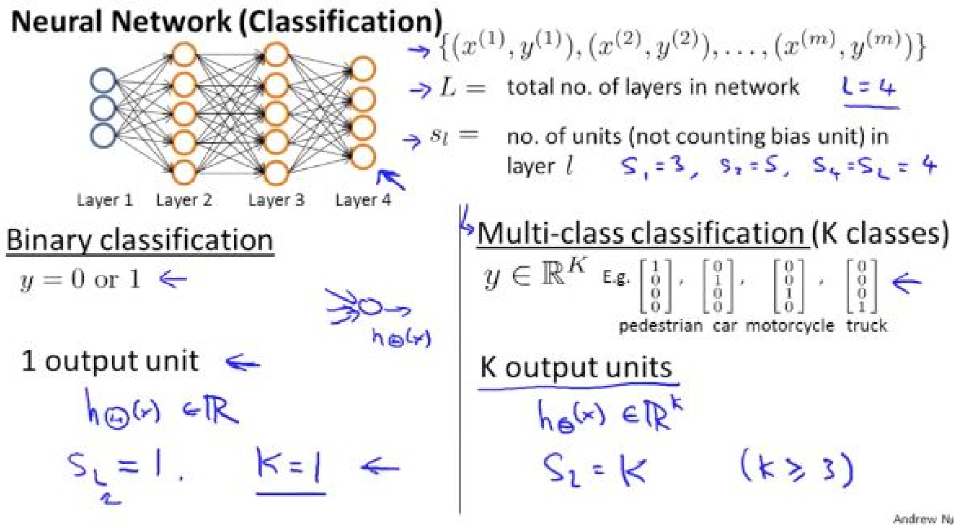
Andrew Ng's Machine Learning Course

Week 5: Neural Networks Learning

Cost Function

- We are going to talk about the application of neural networks to classification problems. First we define L as the total number of layers in a network and s_j as the number of units in a layer. We will focus in 2 classification problems: Binary where the output layer = 1 and multiclass classification (K classes) where there are K outputs and $K \geq 3$.



Neural Network (Classification)

$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

$L =$ total no. of layers in network $L=4$

$s_l =$ no. of units (not counting bias unit) in layer l $s_1=3,\ s_2=5,\ s_4=s_L=4$

Binary classification

$y = 0$ or 1

1 output unit

$h_\Theta(x) \in \mathbb{R}$

$s_L = 1$, $K=1$

$h_\Theta(x)$

Multi-class classification (K classes)

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

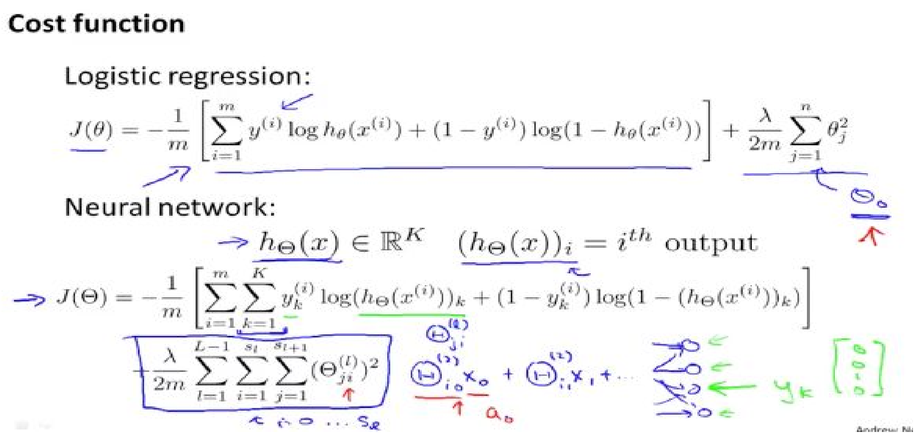
K output units

$h_\Theta(x) \in \mathbb{R}^K$

$s_L = K$ $(K \geq 3)$

Andrew Ng

- The cost function for a neural network is a generalization of the logistic regression regularized cost function. The differences is that first we have two sums where the first sum is over the all the neurons in one layer (sum from K=1 to K) and the regularized term iterates over every neuron except the bias factor neurons. In this case $h_\theta(x)_k$ is the hypothesis that results in the kth output



Cost function

Logistic regression:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1-y^{(i)}) \log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

θ_0

Neural network:

$h_\Theta(x) \in \mathbb{R}^K$ $(h_\Theta(x))_i = i^{th}$ output

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1-y_k^{(i)}) \log(1-(h_\Theta(x^{(i)}))_k)\right]$$
$$+ \boxed{\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2}$$

$i = 0 \ldots s_l$

$\Theta_{ji}^{(l)}$ $\Theta_{i0}^{(2)}x_0 + \Theta_{i1}^{(2)}x_1 + \ldots$ a_0

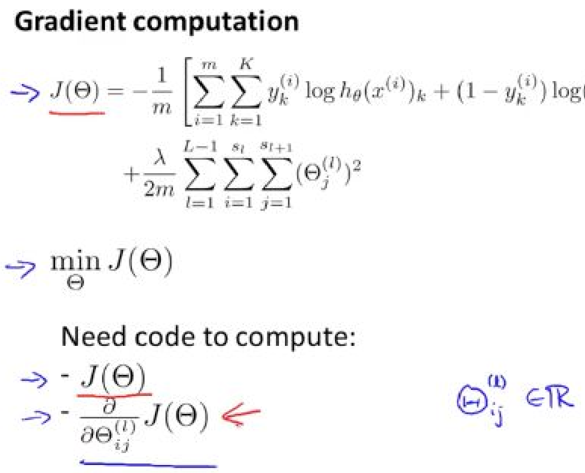
y_k $\begin{bmatrix}0\\0\\0\\0\end{bmatrix}$

Andrew Ng

The double sum simply adds up the logistic regression costs calculated for each cell in the output layer. The triple sum adds up the squares of all the individual θs in the entire network

Backpropagation

- We are going to talk about an algorithm to minimize the cost function: the backpropagation algorithm. What we would like to do is find parameters θ that minimize the $J(\theta)$ cost function shown below. In order to use gradient descent or a more advanced optimization algorithm we need to write code that receives θ and computes $J(\theta)$ and the partial derivatives. We already have the $J(\theta)$ cost function defined so we will concentrate in this class on how to compute the partial derivatives



Gradient computation

$$\rightarrow \underline{J(\Theta)} = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log h_\theta(x^{(i)})_k + (1-y_k^{(i)}) \log(1-h_\theta(x^{(i)})_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j}^{(l)})^2$$

$$\rightarrow \min_\Theta J(\Theta)$$

Need code to compute:

$\rightarrow$ - $\underline{J(\Theta)}$

$\rightarrow$ - $\underline{\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)} \leftarrow$ $\qquad \Theta_{ij}^{(l)} \in \mathbb{R}$

-
- Let's take the example where we only have one training example, an (x, y) pair. Given this one training example we apply forward propagation as shown below. $a^{(1)} = x$, then we move on the 2nd layer which is the sigmoid of the multiplication of the θ matrix * $a^{(1)}$, we then have $z^{(2)} = \theta^{(1)} * a^{(1)}$ and we then apply the sigmoid to get $a^{(2)} = g(z^{(2)})$ (remember to add the bias factor).

Gradient computation

Given one training example $\underline{(x, y)}$:

Forward propagation:

$\underline{a^{(1)}} = \underline{x}$

$\rightarrow z^{(2)} = \Theta^{(1)} a^{(1)}$

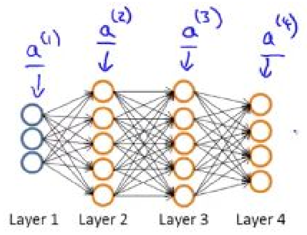
$\rightarrow a^{(2)} = g(z^{(2)})$ $\quad$ (add $\underline{a_0^{(2)}}$)

$\rightarrow z^{(3)} = \Theta^{(2)} a^{(2)}$

$\rightarrow a^{(3)} = g(z^{(3)})$ $\quad$ (add $a_0^{(3)}$)

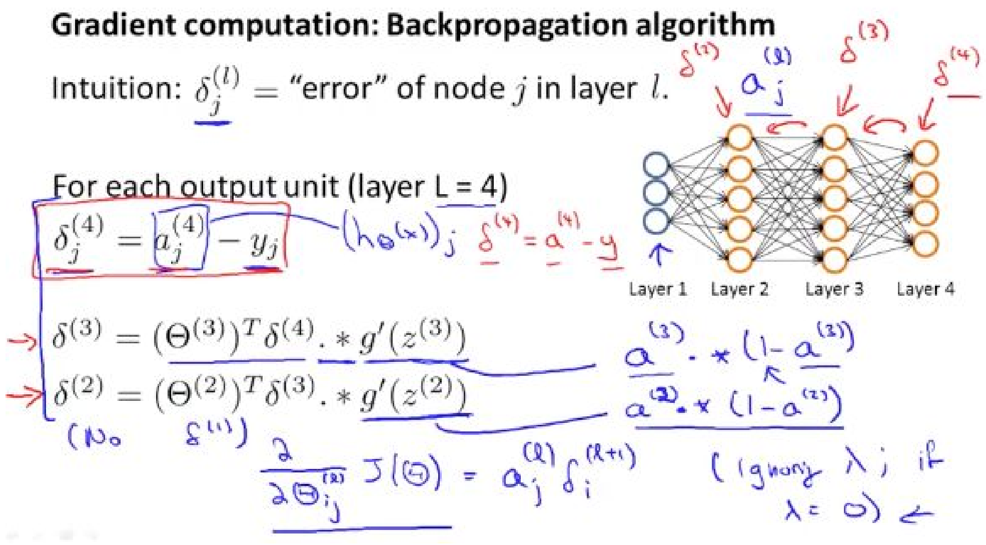
$\rightarrow z^{(4)} = \Theta^{(3)} a^{(3)}$

$\rightarrow a^{(4)} = \underline{h_\Theta(x)} = g(z^{(4)})$



We do the same calculations for the remaining layers and our last layer ends up being $a^{(4)} = h_\theta(x) = g(\theta^{(3)} a^{(3)})$

- We will use backpropagation to compute the derivatives. It is called backpropagation because we start by computing our errors in the output layer and we work backwards by 'propagating' our errors to the previous layers. Our intuition here is that $\delta_j^{(l)} = error\ of\ node\ j\ in\ layer\ l$. For a neural network with 4 layers, like the one shown below, we can start with the output layer and $\delta_j^{(4)} = a_j^{(4)} - y_j$. Remember that $a_j^{(4)} = h_\theta(x)$ and the vectorized implementation is then $\delta^{(4)} = a^{(4)} - y$. Now we have the value for the derivative for the output layer.



Gradient computation: Backpropagation algorithm

Intuition: $\underline{\delta_j^{(l)}}$ = "error" of node j in layer l.

For each output unit (layer L = 4)

$\boxed{\delta_j^{(4)} = \boxed{a_j^{(4)}} - y_j}$ $(h_\Theta(x))_j$ $\delta^{(4)} = a^{(4)} - y$

$\rightarrow \delta^{(3)} = (\Theta^{(3)})^T\delta^{(4)} .* g'(z^{(3)})$ $a^{(3)} .* (1 - a^{(3)})$

$\rightarrow \delta^{(2)} = (\Theta^{(2)})^T\delta^{(3)} .* g'(z^{(2)})$ $a^{(2)} .* (1 - a^{(2)})$

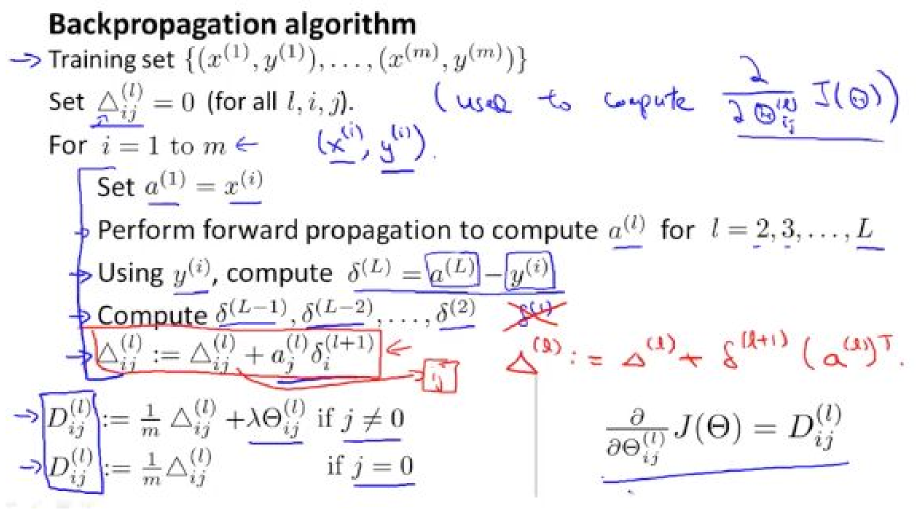
(No $\delta^{(1)}$)

$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$ (ignoring λ; if $\lambda = 0$) $\leftarrow$

Then we go backwards and compute the error/partial derivative for the previous layers. For $\delta^{(3)} = (\theta^{(3)})^T\delta^{(4)} * g'(z^{(3)})$ and we compute $g'\ as\ a^{(3)} .* (1 - a^{(3)})$ .For $\delta^{(2)} = (\theta^{(2)})^T\delta^{(3)} * g'(z^{(2)})$ and $g'\ is\ a^{(2)} .* (1 - a^{(2)})$. We can prove this mathematically (assuming there is no regularization λ or $\lambda = 0$) but that is outside the scope of the class. Finally notice there is no $\delta^{(1)}$ for the input layer because those values are the features we observed in our training set and those do not have any error

- Generalizing for all kinds of training sets we have the following: given a training set of (x,y) value pairs
  - we set $\Delta_{ij}^{(l)} = 0\ (for\ all\ i, j,\ l)$. These are our accumulators
  - We then do a for loop $for\ i = 1..\ m$
    - Set $a^{(1)} = x^{(i)}$
    - Perform forward propagation to compute $a^{(l)}$ for all layers
    - Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
    - Do backpropagation to compute $\delta^{(L-1)}, \delta^{(L-2)} ... \delta^{(2)}$
    - $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(L)}\delta_i^{(L+1)}$
  - Outside the loop we then have

- $D_{ij}^{(L)} := 1/m \;\; \Delta_{ij}^{(l)} + \lambda\theta_{ij}^{(L)}$ if j <> 0
- $D_{ij}^{(L)} := 1/m \;\; \Delta_{ij}^{(l)}$ if j = 0



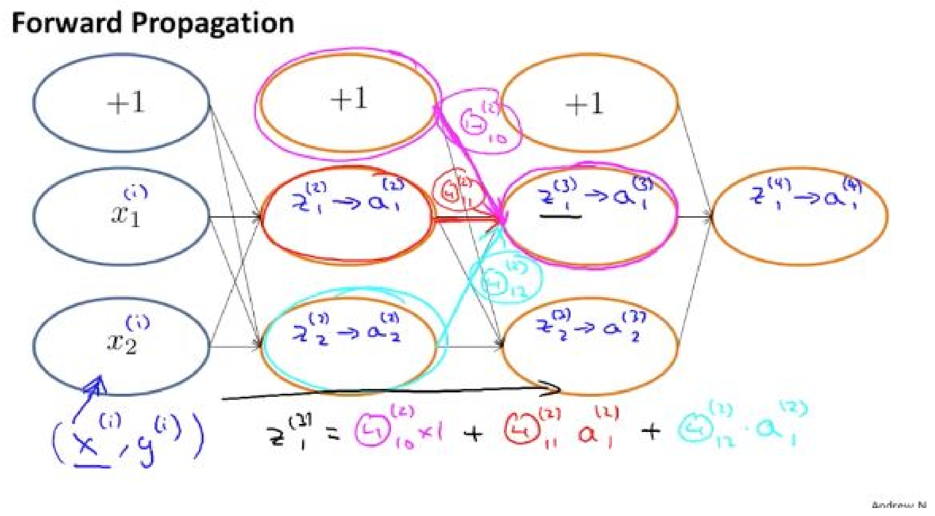
Backpropagation algorithm

```
Training set $\{(x^{(1)},y^{(1)}),\ldots,(x^{(m)},y^{(m)})\}$
Set $\triangle_{ij}^{(l)} = 0$ (for all $l,i,j$).    (used to compute $\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta)$)
For $i = 1$ to $m$ ←    $(x^{(i)}, y^{(i)})$.
    Set $a^{(1)} = x^{(i)}$
    Perform forward propagation to compute $a^{(l)}$ for $l = 2,3,\ldots,L$
    Using $y^{(i)}$, compute $\delta^{(L)} = \boxed{a^{(L)}} - \boxed{y^{(i)}}$
    Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$  $\cancel{\delta^{(1)}}$
    $\boxed{\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)}\delta_i^{(l+1)}}$ ←    $\triangle^{(l)} := \triangle^{(l)} + \delta^{(l+1)}(a^{(l)})^T$.
$\boxed{D_{ij}^{(l)}} := \frac{1}{m}\triangle_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)}$ if $j \neq 0$
$\boxed{D_{ij}^{(l)}} := \frac{1}{m}\triangle_{ij}^{(l)}$ if $j = 0$
```

$$\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta) = D_{ij}^{(l)}$$

Backpropagation Intuition

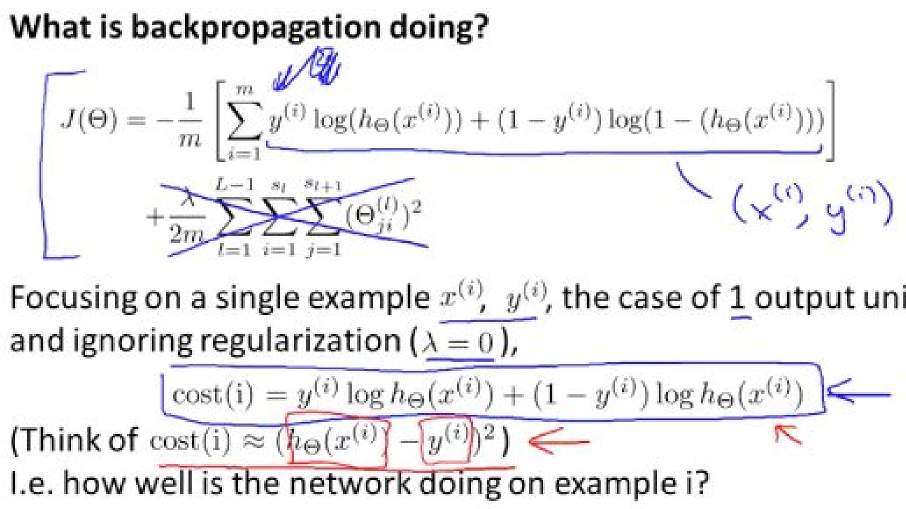
- To get a better intuition and understanding of what backpropagation is doing let's start by reviewing how forward propagation works. In the neural network below we know how to get the values for each node: it is z to which we apply the sigmoid function to get the activation value a with the appropriate indices. Now let's concentrate on the 2nd neuron of the 2nd hidden layer and let's see how we compute the value there.


Forward Propagation
+1
$x_1^{(i)}$
$x_2^{(i)}$
$z_1^{(2)} \to a_1^{(2)}$
$z_2^{(2)} \to a_2^{(2)}$
$z_1^{(3)} \to a_1^{(3)}$
$z_2^{(3)} \to a_2^{(3)}$
$z_1^{(4)} \to a_1^{(4)}$
$\Theta_{10}^{(2)}$
$\Theta_{11}^{(2)}$
$\Theta_{12}^{(2)}$
$(x^{(i)}, y^{(i)})$
$z_1^{(3)} = \Theta_{10}^{(2)}\times 1 + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} \cdot a_1^{(2)}$
Andrew N

We know there is a θ vector in between the 2 layers and we know the other input is the appropriate a value from the previous layer so for that neuron we compute its z value as

$z_1^{(3)} = \theta_{1,0}^{(2)} * 1 + \theta_{1,1}^{(2)} * a_1^{(2)} + \theta_{1,2}^{(2)} * a_2^{(2)}$. We did these calculations in a forward way, backpropagation is doing something very similar but from right to left

- What is backpropagation doing? Let's start by examining the cost function for a single training example $(x^{(i)}, y^{(i)})$ with a 1 output unit and with no regularization ($\lambda = 0$). In this case the cost function is $cost(i) = y^{(i)} log(h_\theta(x^{(i)})) + (1 - y^{(i)})\, log(1 - h_\theta(x^{(i)}))$. For intuition purposes, we can also think of the cost as the square error which would give us $cost(i) = (h_\theta(x^{(i)} - y^{(i)})^2$



What is backpropagation doing?

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} \underbrace{y^{(i)}\log(h_\Theta(x^{(i)})) + (1-y^{(i)})\log(1-(h_\Theta(x^{(i)})))}_{(x^{(i)},\, y^{(i)})}\right] \cancel{+ \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2}$$

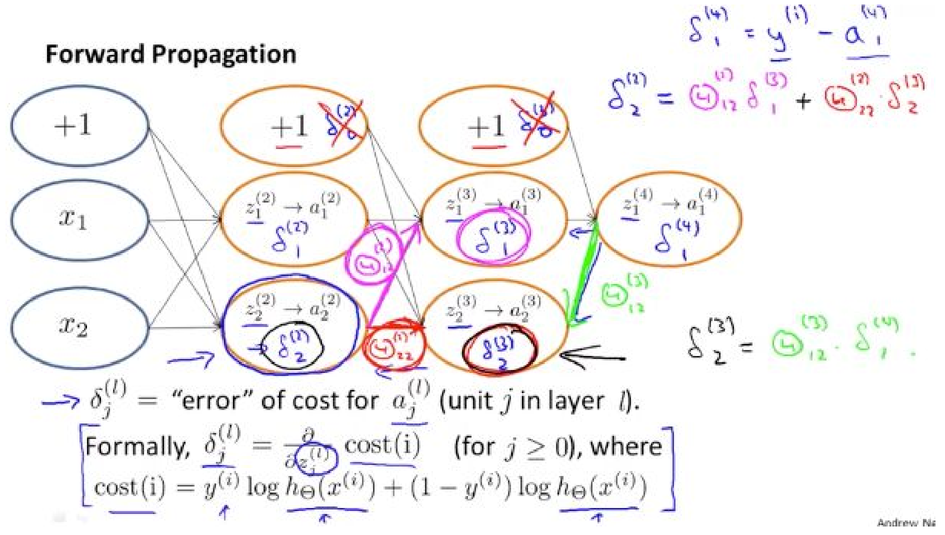
Focusing on a single example $x^{(i)}$, $y^{(i)}$, the case of 1 output unit, and ignoring regularization ($\lambda = 0$),

$$\boxed{\text{cost}(i) = y^{(i)}\log h_\Theta(x^{(i)}) + (1-y^{(i)})\log h_\Theta(x^{(i)})}$$

(Think of $\text{cost}(i) \approx (h_\Theta(x^{(i)}) - y^{(i)})^2$)

I.e. how well is the network doing on example i?

- Now let's take a shot at see how backpropagation works. What we want to do is compute the δ values for each neuron in the network. We start with the output layer (working backwards) and that value is $\delta_1^{(4)} = a_1^{(4)} - y^{(i)}$ which is the cost error for $a_1^{(4)}$.



Forward Propagation

$\delta_1^{(4)} = y^{(i)} - a_1^{(4)}$
$\delta_2^{(2)} = \Theta_{12}^{(2)}\delta_1^{(3)} + \Theta_{22}^{(2)}\delta_2^{(3)}$
$\delta_2^{(3)} = \Theta_{12}^{(3)} \cdot \delta_1^{(4)}$
+1
x_1
x_2
+1 $\cancel{\delta_0^{(2)}}$
$z_1^{(2)} \to a_1^{(2)}$ $\delta_1^{(2)}$
$z_2^{(2)} \to a_2^{(2)}$ $\delta_2^{(2)}$
+1 $\cancel{\delta_0^{(3)}}$
$z_1^{(3)} \to a_1^{(3)}$ $\delta_1^{(3)}$
$z_2^{(3)} \to a_2^{(3)}$ $\delta_2^{(3)}$
$z_1^{(4)} \to a_1^{(4)}$ $\delta_1^{(4)}$
$\Theta_{12}^{(2)}$
$\Theta_{22}^{(2)}$
$\Theta_{12}^{(3)}$
Andrew Ng

$\delta_j^{(l)}$ = "error" of cost for $a_j^{(l)}$ (unit j in layer l).

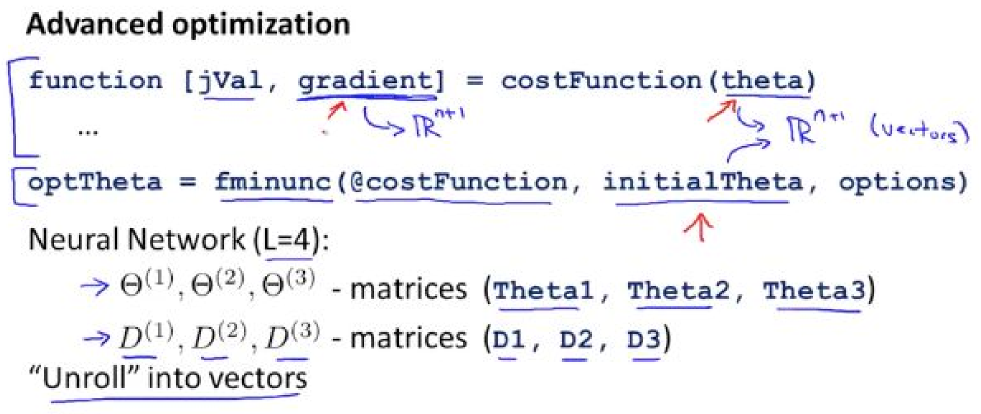
Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}}\text{cost}(i)$ (for $j \geq 0$), where

$\text{cost}(i) = y^{(i)}\log h_\Theta(x^{(i)}) + (1-y^{(i)})\log h_\Theta(x^{(i)})$

Now to compute a δ value for one of the nodes in the hidden layers remember there is a θ matrix in between layers. We are going to use that θ vector and the 'previous' δ values to calculate the cost error for that node. Let's take the 2nd node in the first hidden layer so $\delta_2^{(2)} = \theta_{1,2}^{(2)} * \delta_1^{(3)} + \theta_{2,2}^{(2)} * \delta_2^{(3)}$. Similarly, we can compute the error cost for the 2nd node in the 3rd layer as follows: $\delta_2^{(3)} = \theta_{1,2}^{(3)} * \delta_1^{(4)}$

Unrolling Parameters

- We need to be concerned about an implementation detail: some of the parameters we use in advanced optimization functions expect vectors as inputs and outputs. However, neural networks have matrices: $\theta^{(1)} \ldots \theta^{(n)}$ and $D^{(1)} \ldots D^{(n)}$ so we need to 'unroll' our matrices into vectors so they can be used with these advanced optimization algorithms



Advanced optimization

```
function [jVal, gradient] = costFunction(theta)
   ...
optTheta = fminunc(@costFunction, initialTheta, options)
```

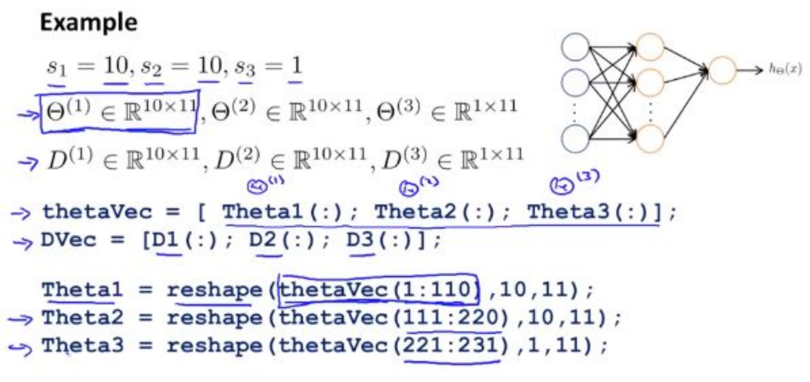
Neural Network (L=4):

$\rightarrow \Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - matrices (`Theta1, Theta2, Theta3`)

$\rightarrow D^{(1)}, D^{(2)}, D^{(3)}$ - matrices (`D1, D2, D3`)

"Unroll" into vectors

- Let's see how we do this using an example. We have a neural network with 10 nodes per layer and 1 output unit. The dimensions of the θ matrices and the D matrices are shown below. We 'unroll' the theta matrices into vectors using the octave command thetaVec = *[Theta1(:); Theta2(:); Theta3(:)]*; and we bring back the matrices from a vector by using the following commands:
    - *Theta1 = reshape(thetaVec(1:110), 10, 11);*
    - *Theta2 = reshape(thetaVec(111:220), 10,11);*
    - *Theta3 = reshape(thetaVec(221:231), 10, 11);*



Example

$s_1 = 10, s_2 = 10, s_3 = 1$

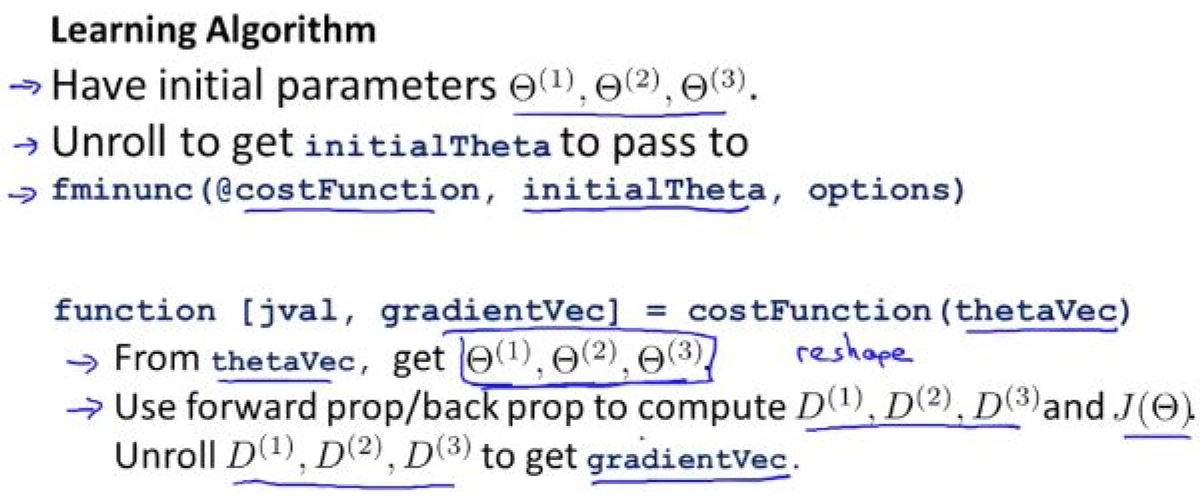
$\rightarrow \boxed{\Theta^{(1)} \in \mathbb{R}^{10\times 11}}, \Theta^{(2)} \in \mathbb{R}^{10\times 11}, \Theta^{(3)} \in \mathbb{R}^{1\times 11}$

$\rightarrow D^{(1)} \in \mathbb{R}^{10\times 11}, D^{(2)} \in \mathbb{R}^{10\times 11}, D^{(3)} \in \mathbb{R}^{1\times 11}$

```
-> thetaVec = [ Theta1(:); Theta2(:); Theta3(:)];
-> DVec = [D1(:); D2(:); D3(:)];

   Theta1 = reshape(thetaVec(1:110),10,11);
-> Theta2 = reshape(thetaVec(111:220),10,11);
-> Theta3 = reshape(thetaVec(221:231),1,11);
```

- Here's how we are going to use these commands in a real situation:
  - We have the initial theta matrices
  - We unroll them into $thetaVec$ so it can be passed to our cost function
  - Inside the cost function I get the individual theta matrices using reshape
  - I use forward and back propagation to compute the D matrices and $J(\theta)$
  - I unroll the D matrices into a $gradientVec$ which is what is returned by the cost function



Learning Algorithm

→ Have initial parameters $\underline{\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}}$.

→ Unroll to get `initialTheta` to pass to

→ `fminunc(@costFunction, initialTheta, options)`

```
function [jval, gradientVec] = costFunction(thetaVec)
```

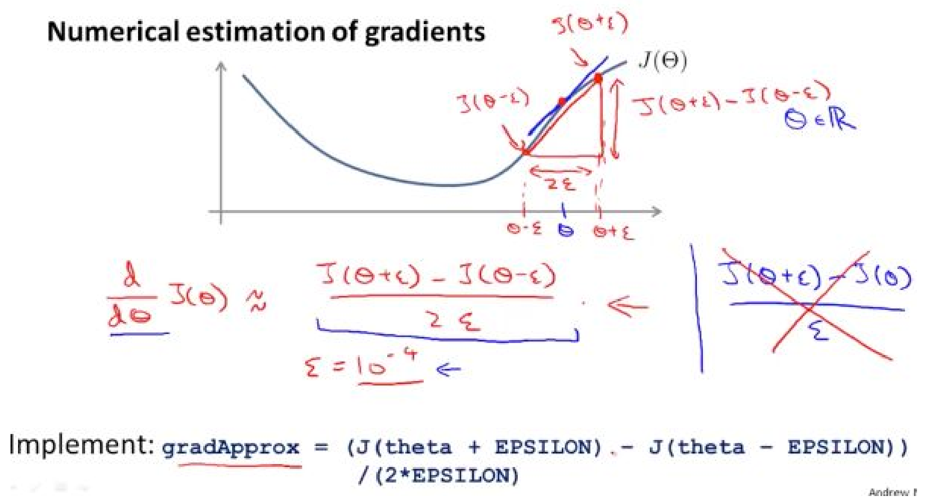
→ From `thetaVec`, get $\boxed{\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}}$ reshape

→ Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$.

Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get `gradientVec`.

Gradient Checking

- Backprop is susceptible to some subtle bugs that are hard to spot and that you might not know about. Your algorithm could be working but you might end up with a bad answer. Because of this, the professor recommends to always implement gradient checking when using backprop
- Concretely this is how we check the gradients. We take a point θ in the $J(\theta)$ curve and create 2 points: $\theta + \varepsilon$ and $\theta - \varepsilon$ and draw a line between the 2; that line is our approximation to the derivative of our cost function. Mathematically, the slope of that line is the height divided by the width The corresponding $J(\theta)$ points for those new theta values are $J(\theta + \varepsilon)$ and $J(\theta - \varepsilon)$. To get the value of slope we then take the height which is $J(\theta + \varepsilon)$ - $J(\theta - \varepsilon)$ divided by the width which is 2ε. My equation for my approximation for the derivative at that θ points is then $(J(\theta + \varepsilon)$ - $J(\theta - \varepsilon))/\, 2\varepsilon$. The ε value is usually very small; the professor uses 10^{-4}. Some people might have seen an alternate equation, the one-sided difference equation which is $(J(\theta + \varepsilon) - J(\theta))/\varepsilon$; ours is called the two-sided difference equation and it gives us a slightly better result so we use that equation. Below we also see the octave code to use for this equation



Numerical estimation of gradients

$$\frac{d}{d\theta} J(\theta) \approx \underbrace{\frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}}$$ ←

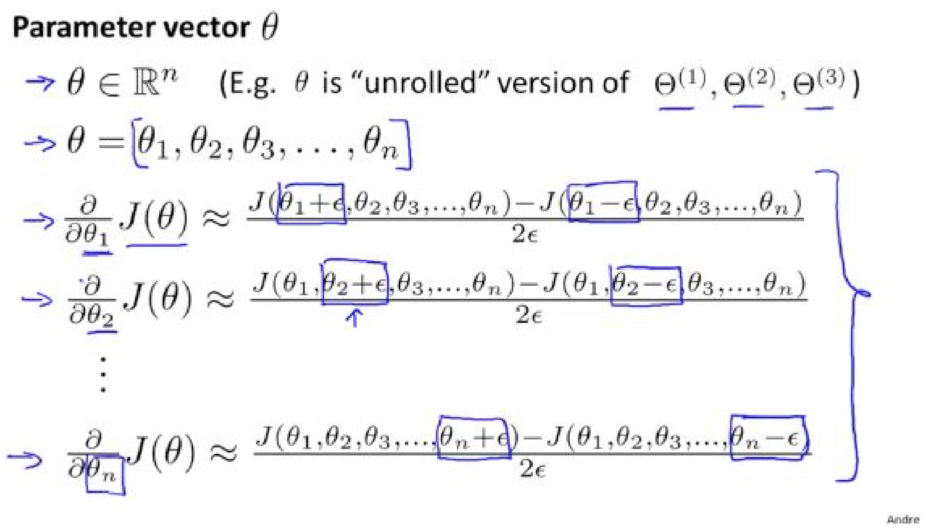
$\varepsilon = 10^{-4}$ ←

~~$\frac{J(\theta+\varepsilon) - J(\theta)}{\varepsilon}$~~

Implement: gradApprox = (J(theta + EPSILON) - J(theta - EPSILON))
/(2*EPSILON)

Andrew N

- The previous example was for the case when θ was a rolled number. Now let's look at the case when θ is a vector of parameters. In this case we use the same logic as in the previous slide, the difference is that here we have a vector so we do it in a one by one basis. For example, to approximate the partial derivative for the cost function with respect to θ_1 we use the formula $J(\theta) \approx J((\theta_1 + \varepsilon, \theta_2, \theta_3, \theta_4 ... \theta_n) - J(\theta_1 - \varepsilon, \theta_2, \theta_3, \theta_4 ... \theta_n) / 2\varepsilon$ and we just repeat the formula for each θ value



Parameter vector θ

→ $\theta \in \mathbb{R}^n$ (E.g. θ is "unrolled" version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$)

→ $\theta = [\theta_1, \theta_2, \theta_3, \ldots, \theta_n]$

$$\frac{\partial}{\partial\theta_1} J(\theta) \approx \frac{J(\boxed{\theta_1+\epsilon}, \theta_2, \theta_3, \ldots, \theta_n) - J(\boxed{\theta_1-\epsilon}, \theta_2, \theta_3, \ldots, \theta_n)}{2\epsilon}$$

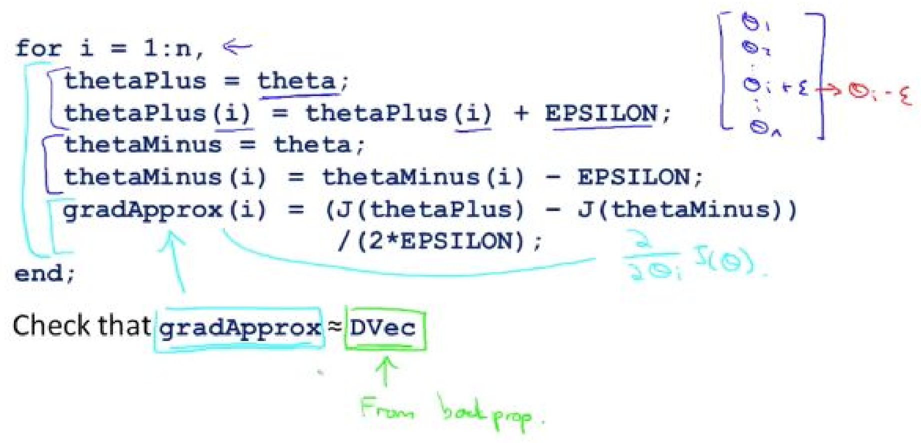
$$\frac{\partial}{\partial\theta_2} J(\theta) \approx \frac{J(\theta_1, \boxed{\theta_2+\epsilon}, \theta_3, \ldots, \theta_n) - J(\theta_1, \boxed{\theta_2-\epsilon}, \theta_3, \ldots, \theta_n)}{2\epsilon}$$

$\vdots$

$$\frac{\partial}{\partial\boxed{\theta_n}} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \ldots, \boxed{\theta_n+\epsilon}) - J(\theta_1, \theta_2, \theta_3, \ldots, \boxed{\theta_n-\epsilon})}{2\epsilon}$$

Andre

- What we actually implement in octave is shown below. At the end of the for loop we have a *gradApprox* value. We can compare that value with the DVec that we got from backprop and if the difference between the two is just a few decimal points I can be confident my backprop algorithm is working correctly

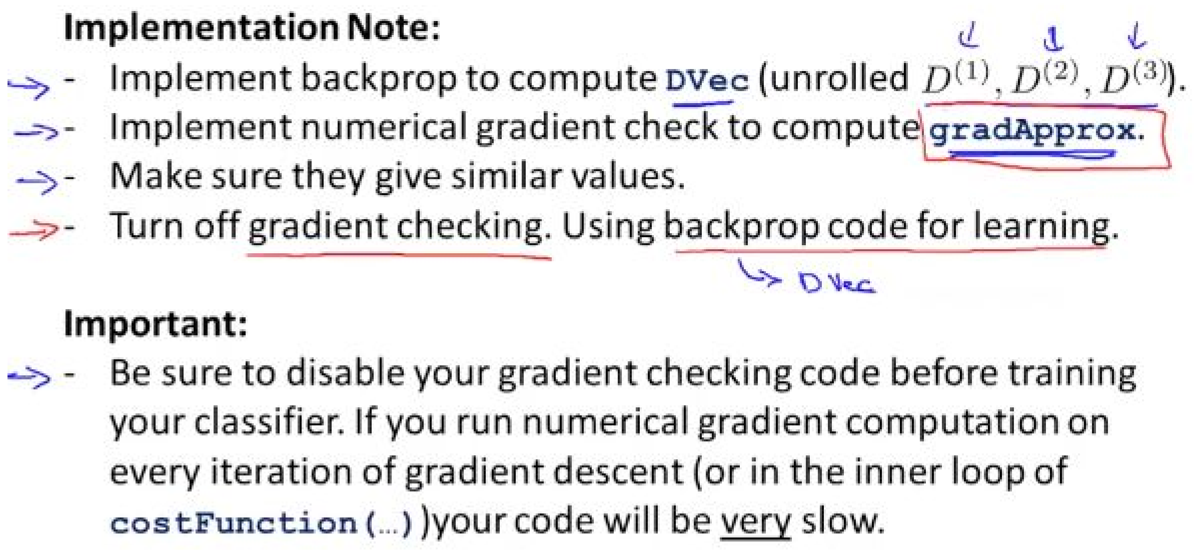


```
for i = 1:n,
    thetaPlus = theta;
    thetaPlus(i) = thetaPlus(i) + EPSILON;
    thetaMinus = theta;
    thetaMinus(i) = thetaMinus(i) - EPSILON;
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                    /(2*EPSILON);
end;
```

Check that gradApprox ≈ DVec

θ_1
θ_2
$\theta_i + \varepsilon \to \theta_i - \varepsilon$
θ_n
$\frac{\partial}{\partial \theta_i} J(\theta)$.
From backprop.

- When implementing this we need to do the following steps:
    - Implement backprop to compute DVec
    - Implement gradient check to compute gradApprox
    - Make sure they give similar values
    - Turn off gradient checking before using backprop code for learning. We do this because backprop is very fast but gradient checking is very slow. If we leave it 'on' our code will be very slow



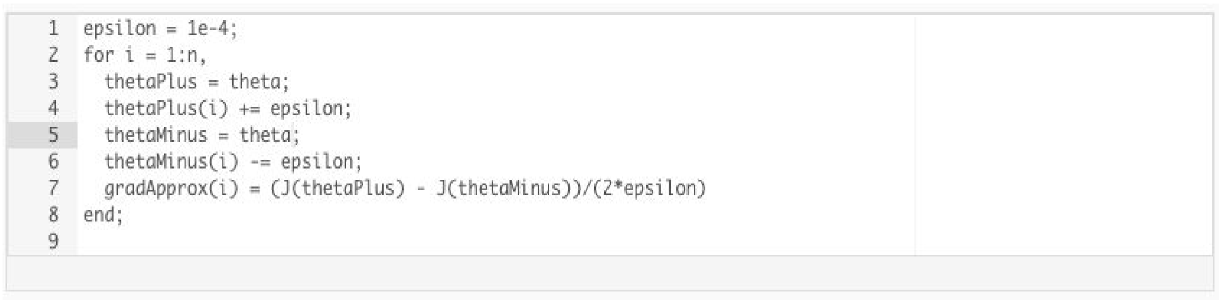
Implementation Note:

- Implement backprop to compute DVec (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$).
- Implement numerical gradient check to compute gradApprox.
- Make sure they give similar values.
- Turn off gradient checking. Using backprop code for learning.

Important:

- Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of costFunction(...))your code will be very slow.

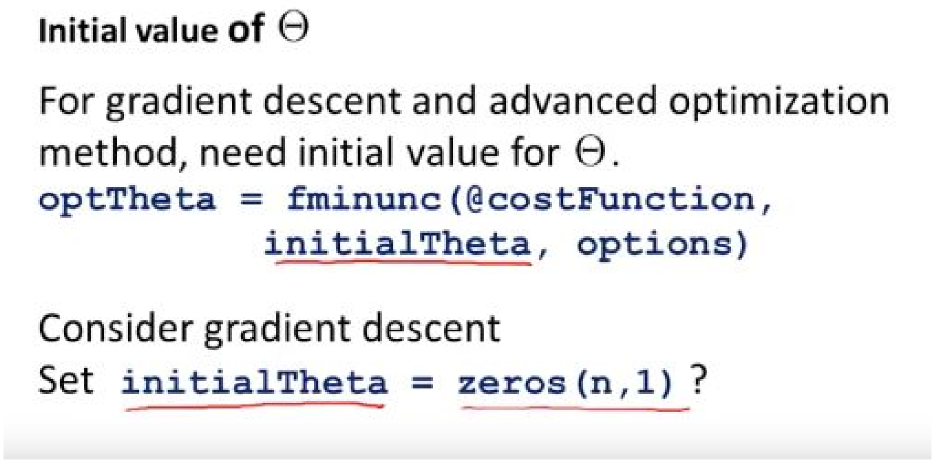
- The octave code that implements gradient checking is shown below for reference.



```
epsilon = 1e-4;
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) += epsilon;
  thetaMinus = theta;
  thetaMinus(i) -= epsilon;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
end;
```

Random Initialization

- One last detail before putting everything together is to select the initial value of theta. In previous occasions we were fine with initializing theta to all zeros. However, that does not work for neural networks



Initial value of Θ

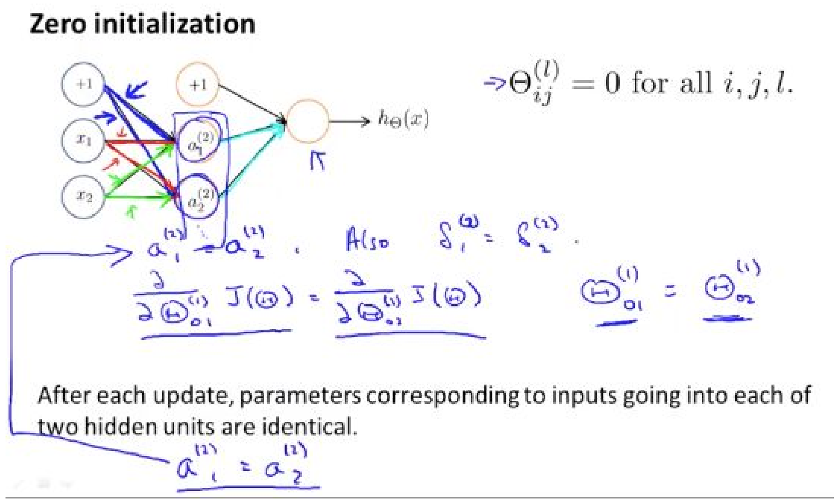
For gradient descent and advanced optimization method, need initial value for Θ.

```
optTheta = fminunc(@costFunction,
           initialTheta, options)
```

Consider gradient descent

Set `initialTheta = zeros(n,1)` ?

- Why is this the case? Because what ends up happening is that the parameters corresponding to inputs into two hidden units are identical and we end up with not only say $a_1^{(2)} = a_2^{(2)}$ but also with $\delta_1^{(2)} = \delta_2^{(2)}$. In practice this fact means our neural network is not computing interesting calculations and therefore we end up with a not very interesting hypothesis $h_\theta(x)$. This problem is also known as symmetric weights



Zero initialization

$\rightarrow \Theta_{ij}^{(l)} = 0$ for all i, j, l.

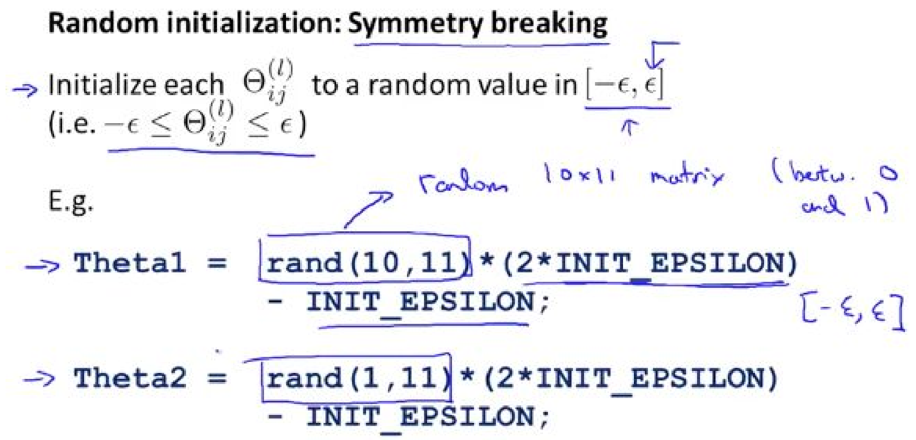
$a_1^{(2)} = a_2^{(2)}$. Also $\delta_1^{(2)} = \delta_2^{(2)}$.

$\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$ $\qquad \Theta_{01}^{(1)} = \Theta_{02}^{(1)}$

After each update, parameters corresponding to inputs going into each of two hidden units are identical.

$a_1^{(2)} = a_2^{(2)}$

- In order to get around this problem we initialize our parameters with random initialization. What we do is we will initialize our theta values to a random value between $-\varepsilon$ *and* ε. This epsilon value is different from the epsilon value we were using in gradient checking.



Random initialization: Symmetry breaking

$\rightarrow$ Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$
(i.e. $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$)

E.g. random 10x11 matrix (betw. 0 and 1)

```
-> Theta1 = rand(10,11)*(2*INIT_EPSILON)
            - INIT_EPSILON;
```

$[-\epsilon, \epsilon]$

```
-> Theta2 = rand(1,11)*(2*INIT_EPSILON)
            - INIT_EPSILON;
```

The octave code to do this is shown above. We start a matrix of random numbers between 0 and 1 and then multiply it by $(2 * \textit{init epsilon}) - \textit{initi epsilon}$ and we do the same for all thetas and that's how we end up with all thetas with values between $[-\varepsilon, \varepsilon]$

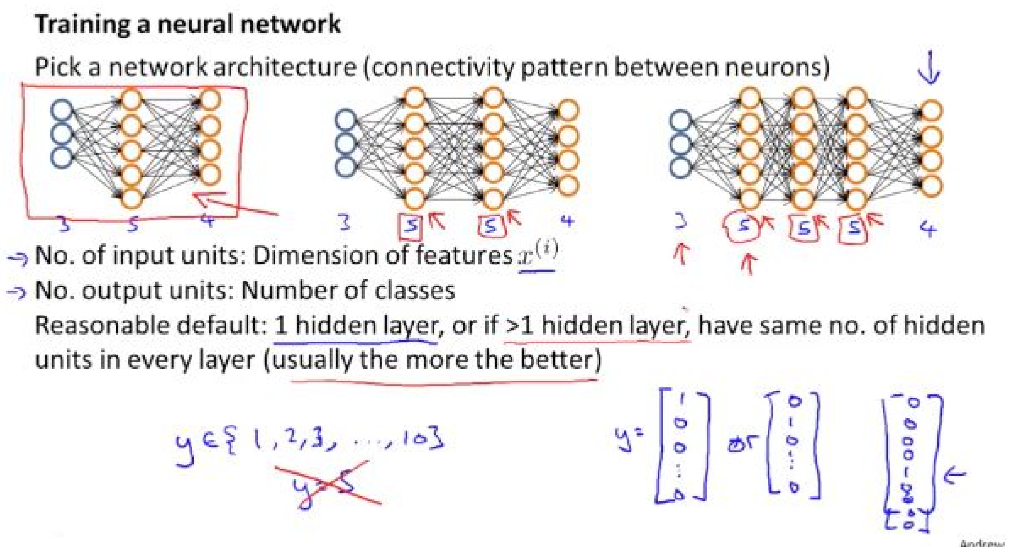
- Summarizing, to train a neural network we should:
    - Initialize the weights to small values close to 0, $[-\varepsilon, \varepsilon]$
    - Implement back propagation
    - Do gradient checking
    - Use gradient descent (or another advanced optimization method) to train to minimize $J(\theta)$ as a function of θ
- Below is some sample octave code we can use to experiment with this method

```
If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11.

Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta3 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
```

Putting it all together

- One of the first steps in training a neural network is to pick a network architecture. You have to make a choice on the number of input & output units, the number of hidden layers and the number of units per hidden layer. In general we can say
  - No of input units is defined by the number of features, i.e. dimension of features
  - No of output units is the number of classes
  - A reasonable default number is 1 hidden layer. If we have to have > 1 hidden layer then have the same number of units in each hidden layer as shown in the sample architectures below



Training a neural network

Pick a network architecture (connectivity pattern between neurons)

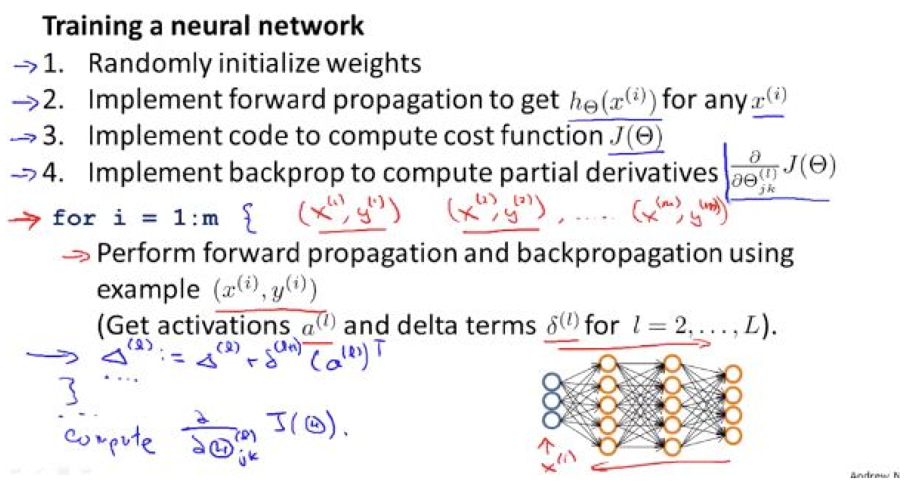
→ No. of input units: Dimension of features $x^{(i)}$

→ No. output units: Number of classes

Reasonable default: 1 hidden layer, or if >1 hidden layer, have same no. of hidden units in every layer (usually the more the better)

3
5
4
3
5
5
4
3
5
5
5
4
$y \in \{1, 2, 3, \dots, 10\}$
$\cancel{y = 5}$
$y = \begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix}$ or $\begin{bmatrix}0\\1\\0\\\vdots\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\\0\\0\\1\\0\\\vdots\\0\end{bmatrix}$
Andrew

- Once we have a network architecture the next step is to train the neural network. There are 6 steps to train the neural network:
  - Randomly initialize weights with values close to, but not quite, zero
  - Implement forward propagation to get $h_\theta(x^{(i)})$ for any $x^{(i)}$
  - Implement code to compute the cost function
  - Implement code to compute the partial derivative of the cost function
    - This code takes the form of a for loop where we perform forward and back propagation for a given $(x^{(i)}, y^{(i)})$
    - We get all the $a^{(l)}$ and all the $\delta^{(l)}$ and we add them to our D accumulators
  - Outside the for loop we compute the partial derivatives for $J(\theta)$

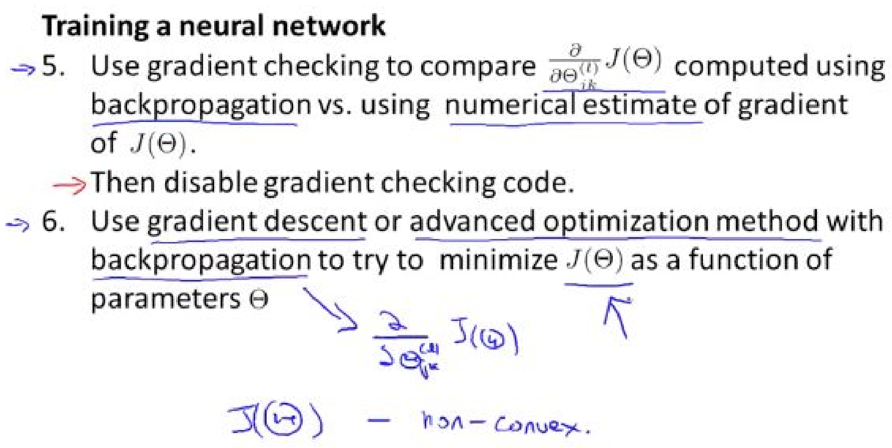


Training a neural network

1. Randomly initialize weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement code to compute cost function $J(\Theta)$
4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial\Theta^{(l)}_{jk}}J(\Theta)$

```
for i = 1:m {   (x^(1), y^(1))   (x^(2), y^(2)), ....  (x^(m), y^(m))
    Perform forward propagation and backpropagation using
    example (x^(i), y^(i))
    (Get activations a^(l) and delta terms δ^(l) for l = 2, ..., L).
    Δ^(l) := Δ^(l) + δ^(l+1) (a^(l))^T
    ....
}
...
compute ∂/∂Θ^(l)_jk J(Θ).
```

- The next 2 steps are
    - Use gradient checking to make sure back prop is correct. We do this by comparing the partial derivatives of $J(\theta)$ vs the numerical estimate of gradient of $J(\theta)$
        - We have to disable the gradient checking code because it is slow
    - Use gradient descent (or another advanced optimization method) to minimize $J(\theta)$



Training a neural network

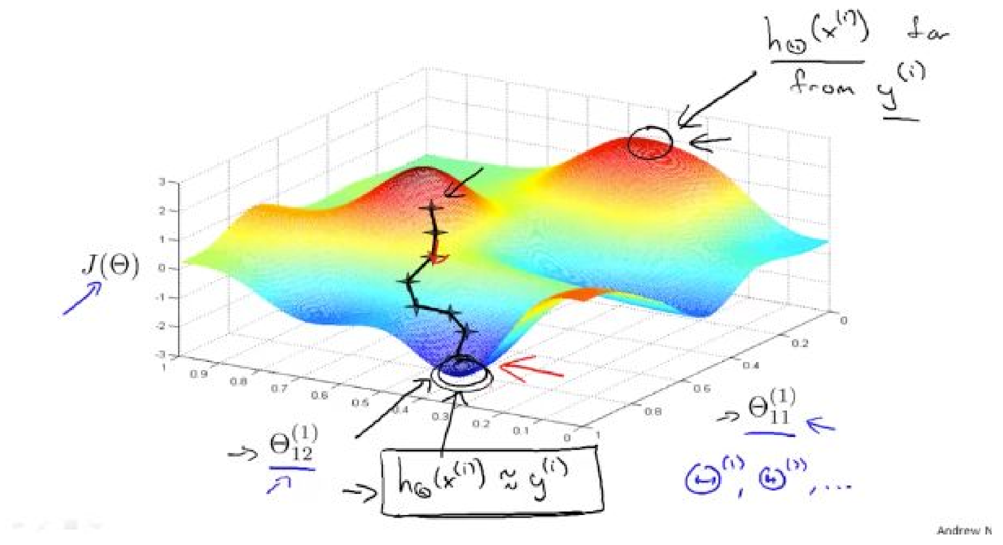
5. Use gradient checking to compare $\frac{\partial}{\partial\Theta^{(l)}_{jk}}J(\Theta)$ computed using backpropagation vs. using numerical estimate of gradient of $J(\Theta)$.
   Then disable gradient checking code.
6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ

$\frac{\partial}{\partial\Theta^{(l)}_{jk}}J(\Theta)$

$J(\Theta)$ — non-convex.

Finally, $J(\theta)$ can end up being non convex and this, as you might remember from previous classes, might cause gradient descent not to converge on the global minimum. In practice this is not a big problem

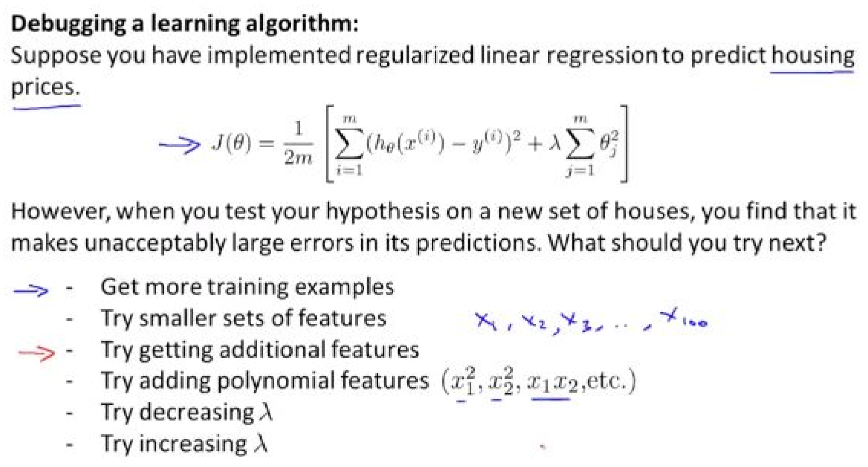
- As a refresher, gradient descent with backpropagation is trying to start from one point and then it is computing the direction or path so we can end up at a point where the difference between the hypothesis $h_\theta(x^{(i)}) \approx y^{(i)}$. In the example below that would be the point circled in black. In contrast, a point atop of one of the red mountains is one where $h_\theta(x^{(i)})$ is far different from $y^{(i)}$



Week 6: Neural Networks Learning

Advice for Applying Machine Learning

- Now we are going to learn some practical advice on how to apply the learning algorithms we have learned so far.
- Suppose we have a learning algorithm like regularized linear regression and we apply it a new set of data and we find out there are very large errors on predictions, what can we do then? There are a few choices (shown below)



Debugging a learning algorithm:

Suppose you have implemented regularized linear regression to predict housing prices.

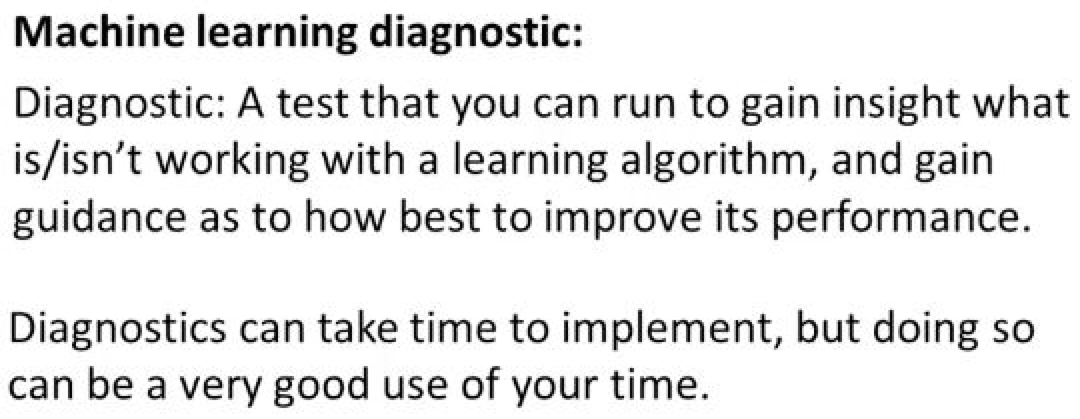
$$\rightarrow J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{m}\theta_j^2\right]$$

However, when you test your hypothesis on a new set of houses, you find that it makes unacceptably large errors in its predictions. What should you try next?

- $\rightarrow$ Get more training examples
- Try smaller sets of features $x_1, x_2, x_3, \dots, x_{100}$
- $\rightarrow$ Try getting additional features
- Try adding polynomial features $(x_1^2, x_2^2, x_1x_2, \text{etc.})$
- Try decreasing λ
- Try increasing λ

The problem is that usually people do not have a method to pick from this menu and go by gut feeling. To make it even worse, some of the choices above could be very long projects and we might end up wasting months and not get any better predictions. So how do we choose from the choices above? Turns out there is a set of techniques that can help us eliminate at least half of the choices; this method is….

- These techniques are known as Machine Learning Diagnostic. A diagnostic is a test that we can run to understand what is/is not working with an algorithm and we can gain guidance on how to improve its performance.



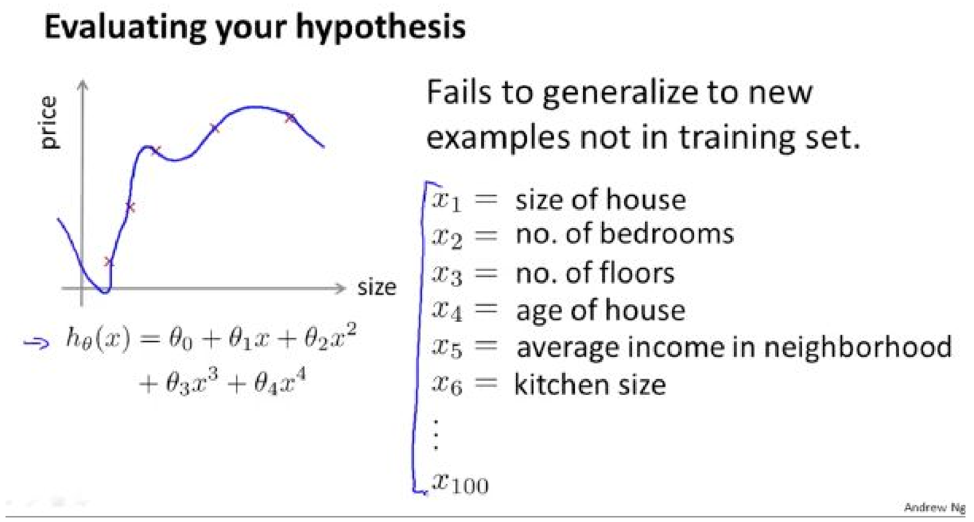
Machine learning diagnostic:

Diagnostic: A test that you can run to gain insight what is/isn't working with a learning algorithm, and gain guidance as to how best to improve its performance.

Diagnostics can take time to implement, but doing so can be a very good use of your time.

Evaluating a Hypothesis

- One way to evaluate a hypothesis is to plot it against the training set as we have been doing. However, that only works for problems where the number of features is small. It is difficult to plot if we have hundreds of features



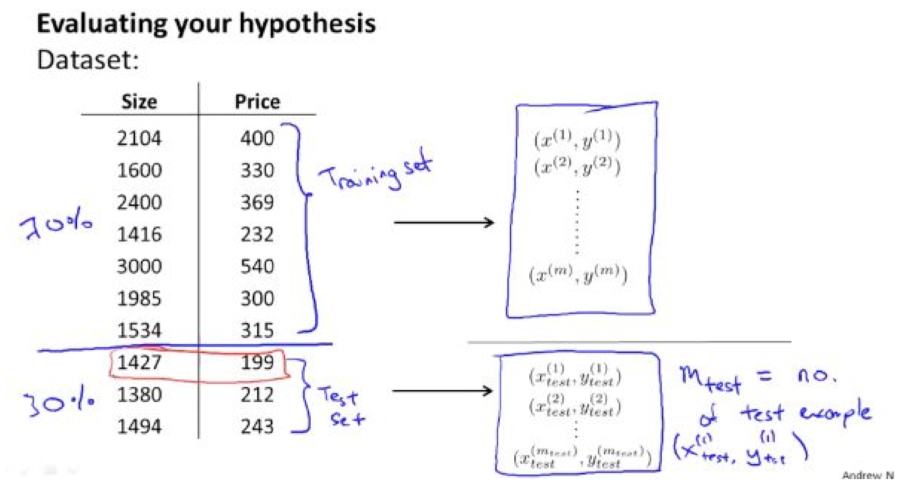
Evaluating your hypothesis

$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

Fails to generalize to new examples not in training set.

$x_1 =$ size of house
$x_2 =$ no. of bedrooms
$x_3 =$ no. of floors
$x_4 =$ age of house
$x_5 =$ average income in neighborhood
$x_6 =$ kitchen size
$\vdots$
x_{100}

- The standard way of evaluating a hypothesis is to take the data we have and assign a certain percentage as a training set (say 70%) and the remaining 30% is our test set. $m_{test} = number\ of\ test\ examples$.



Evaluating your hypothesis

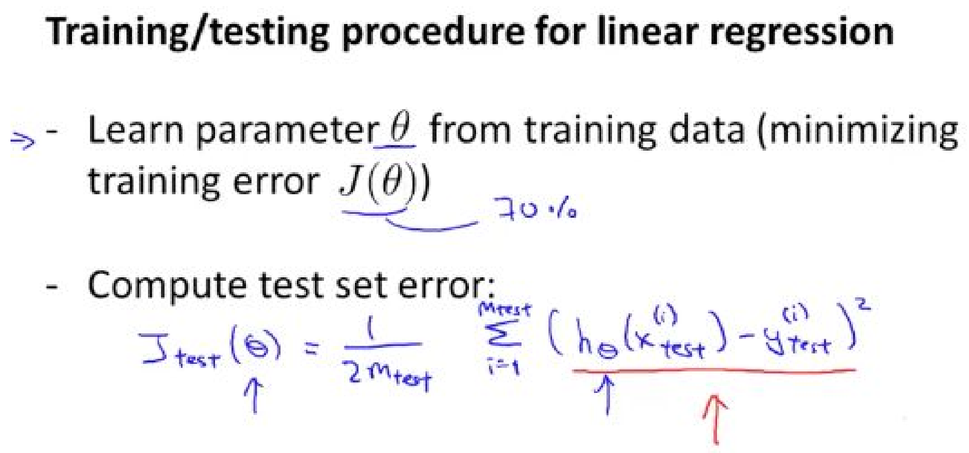
Dataset:

Size	Price
2104	400
1600	330
2400	369
1416	232
3000	540
1985	300
1534	315
1427	199
1380	212
1494	243

Training set
70%
30%
Test set
$(x^{(1)}, y^{(1)})$
$(x^{(2)}, y^{(2)})$
$(x^{(m)}, y^{(m)})$
$(x^{(1)}_{test}, y^{(1)}_{test})$
$(x^{(2)}_{test}, y^{(2)}_{test})$
$(x^{(m_{test})}_{test}, y^{(m_{test})}_{test})$
m_{test} = no. of test example
$(x^{(i)}_{test}, y^{(i)}_{test})$
Andrew N

It is also desirable to randomize the order of the training set so we can use a random 70% as the training set and a random 30% as the test set. In the example above we just used the first 70% but it's better if we make sure it is a random 70% set

- The procedure then for linear regression is to:
  - Learn θ from training data
  - Compute test set error which is expressed as $J(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m} (h_\theta(x^{(i)}_{test} - y^{(i)}_{test})^2$



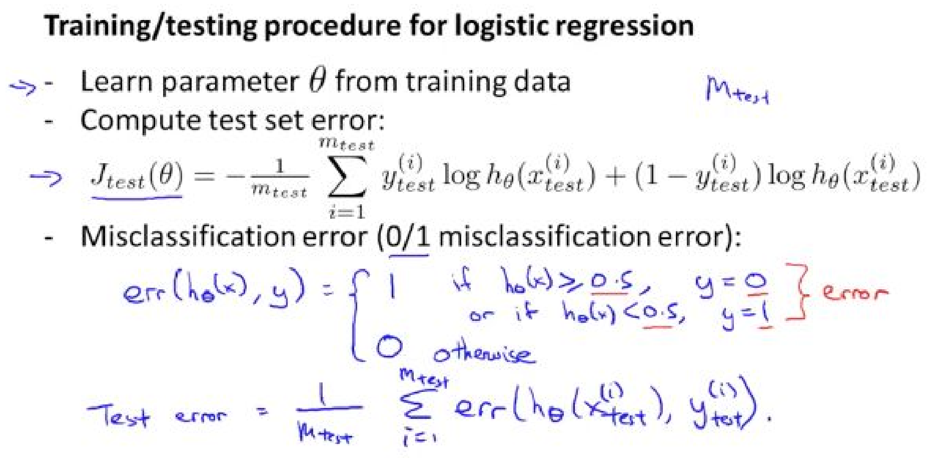
Training/testing procedure for linear regression

- Learn parameter θ from training data (minimizing training error $J(\theta)$)
- Compute test set error:

$$J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} \left(h_\theta(x^{(i)}_{test}) - y^{(i)}_{test} \right)^2$$

- The same process for logistic regression is similar: we learn θ from training set and then compute the error of the test set using the logistic regression function but expressed with m_{test} as follows $J_{test}(\theta) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} y^{(i)}_{test} log(h_\theta(x^{(i)}_{test}) - (1 - y^{(i)}_{test})\, log(h_\theta(x^{(i)}_{test})$.

  In this case sometimes we can use another test metric called the misclassification error (also known as the 0/1 misclassification error) and is defined below. Basically it is showing the number of examples on my test set that the hypothesis got wrong



Training/testing procedure for logistic regression

$\rightarrow$ - Learn parameter θ from training data $\quad m_{test}$

- Compute test set error:

$$\rightarrow \underline{J_{test}(\theta)} = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} y_{test}^{(i)} \log h_\theta(x_{test}^{(i)}) + (1 - y_{test}^{(i)}) \log h_\theta(x_{test}^{(i)})$$

- Misclassification error ($\underline{0/1}$ misclassification error):

$$err(h_\theta(x), y) = \begin{cases} 1 & \text{if } h_\theta(x) \geq \underline{0.5}, \; y = \underline{0} \\ & \text{or if } h_\theta(x) < \underline{0.5}, \; y = \underline{1} \\ 0 & \text{otherwise} \end{cases} \Big\} \text{ error}$$

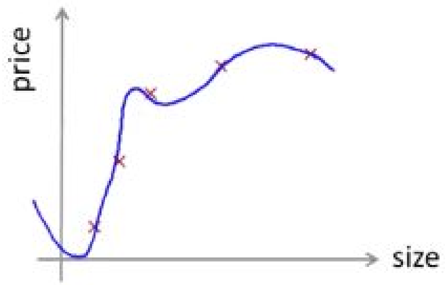
$$\text{Test error} = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err(h_\theta(x_{test}^{(i)}), y_{test}^{(i)}).$$

Finally, to get the average test error we do a summation of the individual errors scaled by 1/mtest as shown above

Model Selection and Train/Validation/Test Sets

- We start by generalizing an overfitting example. We have seen overfitting many times before but the more general principle is that once my parameters θ fit some set of data (the training set), the error of these parameters as measured to that data is likely to be lower than the actual generalization error.

Overfitting example



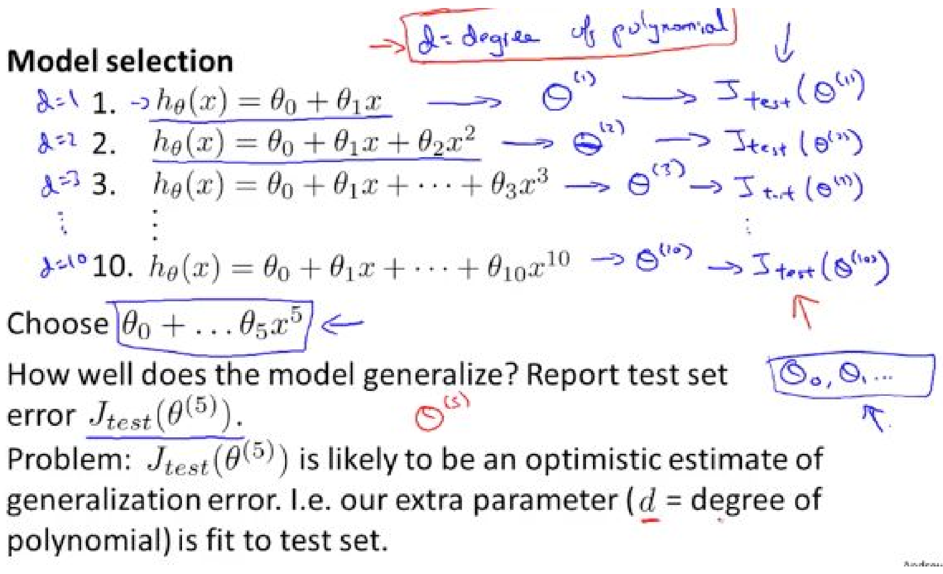
$$h_\theta(x) = \underline{\theta_0} + \underline{\theta_1} x + \underline{\theta_2} x^2 + \underline{\theta_3} x^3 + \theta_4 x^4$$

Once parameters $\theta_0, \theta_1, \ldots, \theta_4$ were fit to some set of data (training set), the error of the parameters as measured on that data (the training error $J(\theta)$) is likely to be lower than the actual generalization error.

In other words, just because the error is low for the training set it does not mean it is going to be the same when applied to other data. In fact, it will probably be higher for the general case

- A related question is how do we select the model to use. That is, what hypothesis should we choose? One with a quadratic equation? One with a 5th order polynomial? One with a 10th order polynomial? What we do here is we introduce a variable,

$d = \textit{degree of polynomial}$ and then we calculate the cost function $J_{test}(\theta)$ for each model choice and choose the one with the lowest cost. Let's assume that it is the 5th order polynomial. Here the question is still the same: how well does this generalize? The answer is, as with the previous slide, not very well and is likely to be an optimistic estimate of the generalization error. Why? For the same reason as before: we choose the d that best fits the test set.



Model selection

$d=1$ 1. $h_\theta(x) = \theta_0 + \theta_1 x \longrightarrow \Theta^{(1)} \longrightarrow J_{test}(\Theta^{(1)})$

$d=2$ 2. $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 \longrightarrow \Theta^{(2)} \longrightarrow J_{test}(\Theta^{(2)})$

$d=3$ 3. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_3 x^3 \longrightarrow \Theta^{(3)} \longrightarrow J_{test}(\Theta^{(3)})$

$\vdots$

$d=10$ 10. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_{10} x^{10} \longrightarrow \Theta^{(10)} \longrightarrow J_{test}(\Theta^{(10)})$

$\boxed{d = \text{degree of polynomial}}$

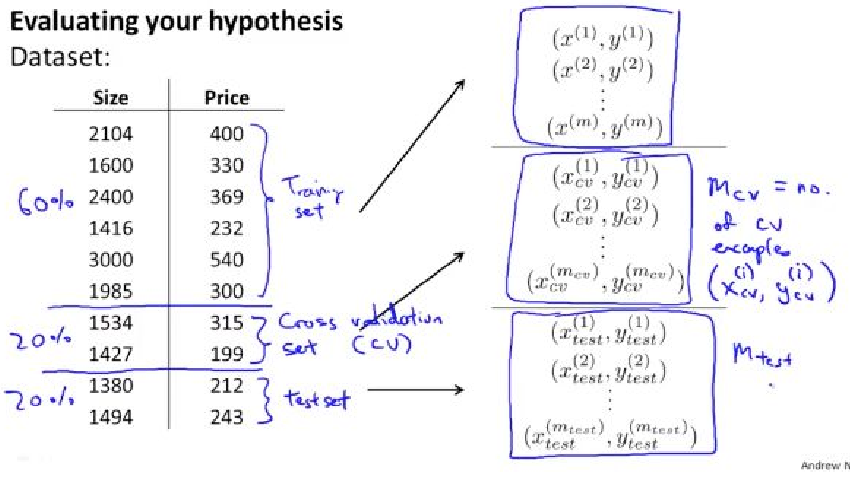
Choose $\boxed{\theta_0 + \ldots \theta_5 x^5}$

How well does the model generalize? Report test set error $J_{test}(\theta^{(5)})$. $\Theta^{(5)}$ $\boxed{\Theta_0, \Theta_1 \ldots}$

Problem: $J_{test}(\theta^{(5)})$ is likely to be an optimistic estimate of generalization error. I.e. our extra parameter (d = degree of polynomial) is fit to test set.

Andrew N

- What we do to get over this problem is to divide our data set in 3 ways: 60% is the training set, 20% is the cross validation (cv) test and the remaining 20% is the usual test set. We then have $m_{cv} = \textit{number of cv examples}$ and an example from this data set is $(x_{cv}^{(i)}, y_{xv}^{(i)})$



Evaluating your hypothesis

Dataset:

	Size	Price	
60%	2104	400	Training set
	1600	330	
	2400	369	
	1416	232	
	3000	540	
	1985	300	
20%	1534	315	Cross validation set (CV)
	1427	199	
20%	1380	212	test set
	1494	243	

$(x^{(1)}, y^{(1)})$
$(x^{(2)}, y^{(2)})$
$\vdots$
$(x^{(m)}, y^{(m)})$

$(x_{cv}^{(1)}, y_{cv}^{(1)})$
$(x_{cv}^{(2)}, y_{cv}^{(2)})$
$\vdots$
$(x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})$

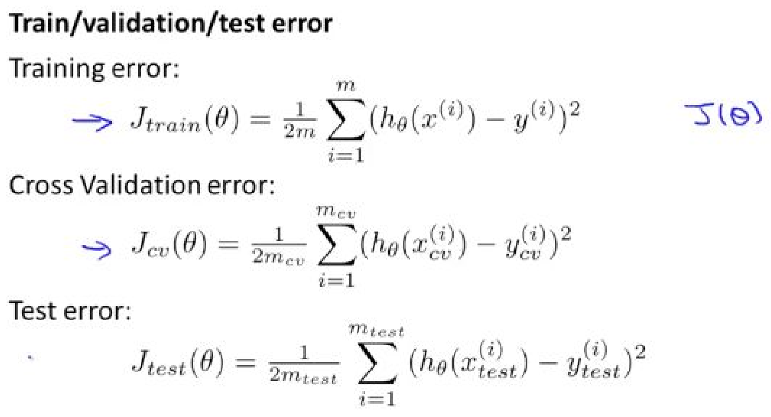
m_{cv} = no. of cv example $(x_{cv}^{(i)}, y_{cv}^{(i)})$

$(x_{test}^{(1)}, y_{test}^{(1)})$
$(x_{test}^{(2)}, y_{test}^{(2)})$
$\vdots$
$(x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$

m_{test}

Andrew N

- The cost function for each of the 3 sections is shown below. It's really the same equation but just uses the training set from each section



Train/validation/test error

Training error:

$$\rightarrow J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \qquad J(\theta)$$

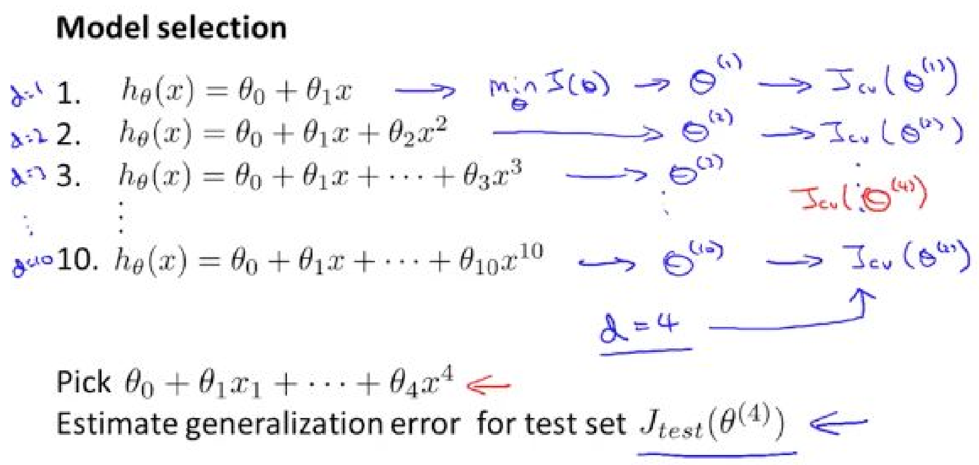
Cross Validation error:

$$\rightarrow J_{cv}(\theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\theta(x^{(i)}_{cv}) - y^{(i)}_{cv})^2$$

Test error:

$$J_{test}(\theta) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}(h_\theta(x^{(i)}_{test}) - y^{(i)}_{test})^2$$

- Our model selection process changes: we still minimize $J(\theta)$ for each of our model selections and that gives us a θ vector for each choice. We then take that parameter vector and apply it to the cross validation set to get the error for each possible choice. Once we are done calculating the error we pick the $h_\theta(x)$ that has the lowest error. In the example below we pick the 4th-order polynomial and use it to estimate the generalization error for the test set (we are now using the 3rd section of our data set). With this method our d parameter has not been trained using the test set, it was trained using the cross validation set



Model selection

$d=1$ 1. $h_\theta(x) = \theta_0 + \theta_1 x \longrightarrow \min_\theta J(\theta) \rightarrow \theta^{(1)} \longrightarrow J_{cv}(\theta^{(1)})$

$d=2$ 2. $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 \longrightarrow \theta^{(2)} \longrightarrow J_{cv}(\theta^{(2)})$

$d=3$ 3. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_3 x^3 \longrightarrow \theta^{(3)}$

$\vdots$ $\qquad J_{cv}(\theta^{(4)})$

$d=10$ 10. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_{10} x^{10} \longrightarrow \theta^{(10)} \longrightarrow J_{cv}(\theta^{(10)})$

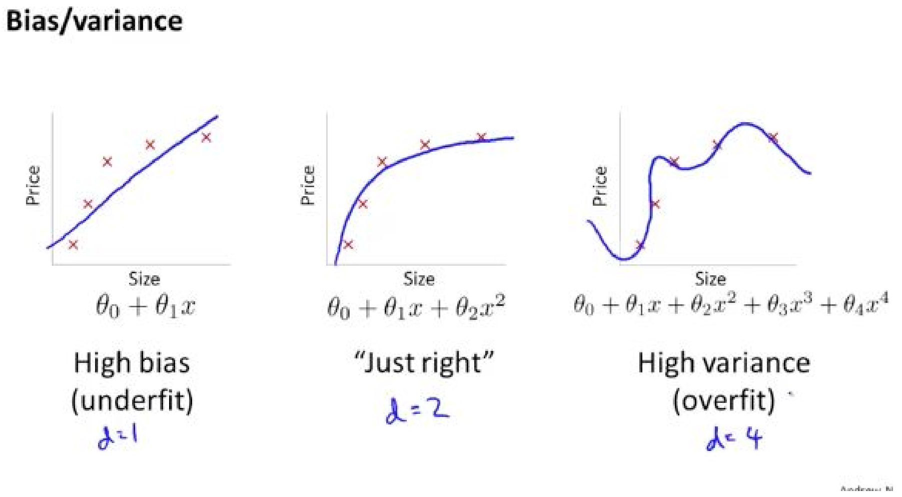
$\underline{d=4}$

Pick $\theta_0 + \theta_1 x_1 + \cdots + \theta_4 x^4 \leftarrow$

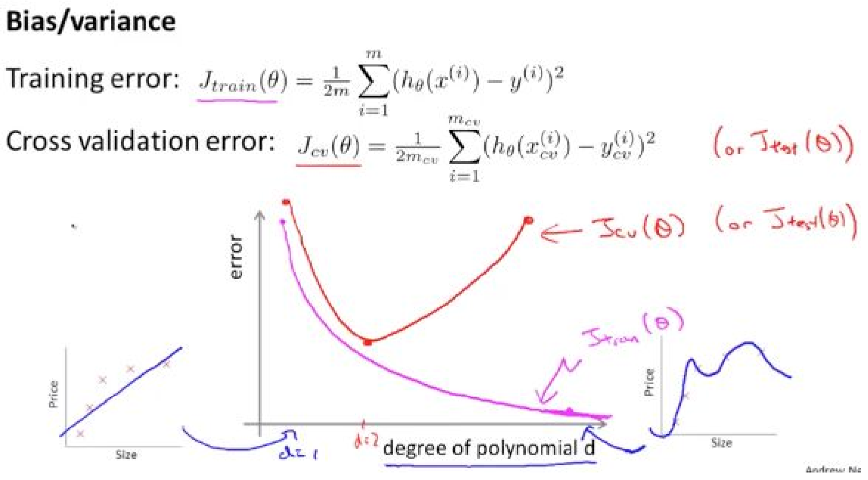
Estimate generalization error for test set $\underline{J_{test}(\theta^{(4)})} \leftarrow$

Diagnosing Bias vs Variance

- To refresh our memory, high bias is underfitting when we have a hypothesis that is too simple (with a low order polynomial) and high variance is the opposite case: overfitting, where we have hypothesis that fits the data set too well (with a somewhat high order polynomial) but does not generalize. Both cases are shown below



- We can get a graphical understanding of bias and variance by taking our $J_{train}(\theta)$ and our $J_{cv}(\theta)$ cost functions and graphing them against the degree of polynomial d. We end up with a chart as below. The $J_{train}(\theta)$ is like that because our error is high when we have a simple hypothesis (underfitting) but it is close to 0 (or perhaps 0) because a higher order polynomial allows to fit (and overfit) our hypothesis better with the training set.



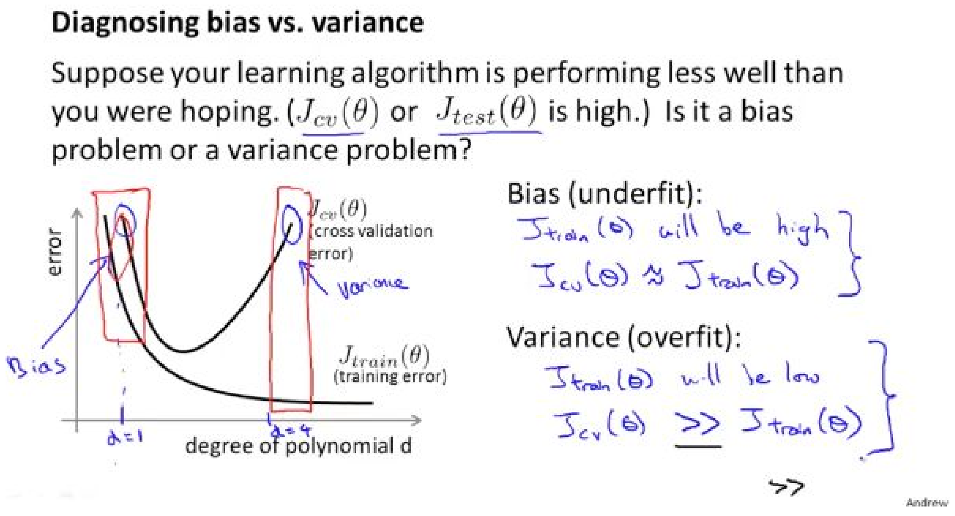
Bias/variance

Training error: $J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$

Cross validation error: $J_{cv}(\theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$ (or $J_{test}(\theta)$)

The cross validation error is a bit different because it starts similar (high error going down as the polynomial order increases) and then it reaches a point where the error starts to increase again (the point where overfitting starts)

- This graph then is a tool to help us identify if we have a bias or variance problem. If we have a bias problem then $J_{train}(\theta)$ is going to be high and $J_{train}(\theta) \simeq J_{cv}(\theta)$. If we have a variance problem $J_{train}(\theta)$ is going to be very low and and $J_{cv}(\theta) \gg J_{train}(\theta)$



Diagnosing bias vs. variance

Suppose your learning algorithm is performing less well than you were hoping. ($J_{cv}(\theta)$ or $J_{test}(\theta)$ is high.) Is it a bias problem or a variance problem?

Bias (underfit):

$J_{train}(\theta)$ will be high

$J_{cv}(\theta) \approx J_{train}(\theta)$

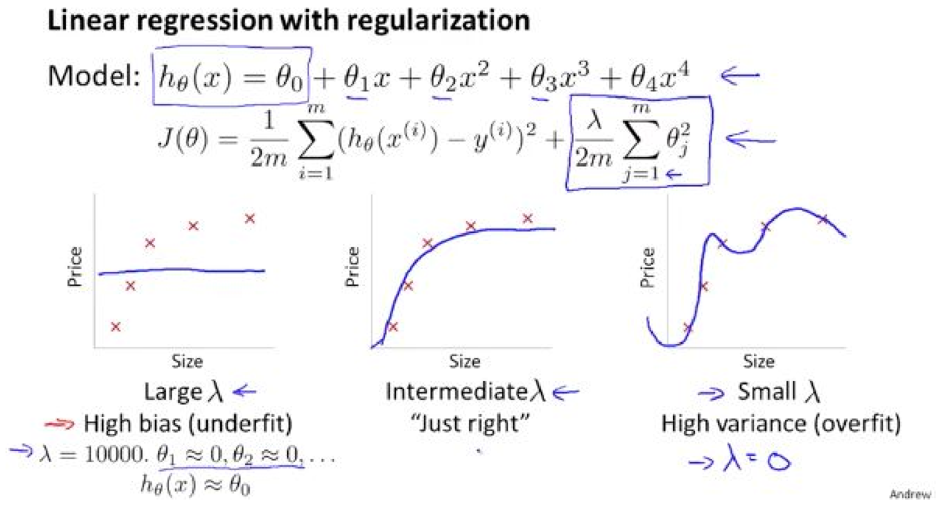
Variance (overfit):

$J_{train}(\theta)$ will be low

$J_{cv}(\theta) \underline{\gg} J_{train}(\theta)$

Regularization & Bias/Variance

- Now we take a look to how regularization impacts bias and variance. Considering the equation below: if we have a big λ value then all our $\theta's$ are 0 and we end up with $h_\theta(x) = \theta_0$. If we have a small λ value (say $\lambda = 0$) then we end up with original $h_\theta(x)$ which is overfitting. An intermediate λ value is the one that gets it 'just right'



Linear regression with regularization

Model: $\boxed{h_\theta(x) = \theta_0} + \underline{\theta_1} x + \underline{\theta_2} x^2 + \underline{\theta_3} x^3 + \theta_4 x^4 \leftarrow$

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \boxed{\frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2} \leftarrow$$

Large $\lambda \leftarrow$

$\rightarrow$ High bias (underfit)

$\rightarrow \lambda = 10000.\ \underline{\theta_1 \approx 0}, \underline{\theta_2 \approx 0}, \ldots$

$h_\theta(x) \approx \theta_0$

Intermediate $\lambda \leftarrow$

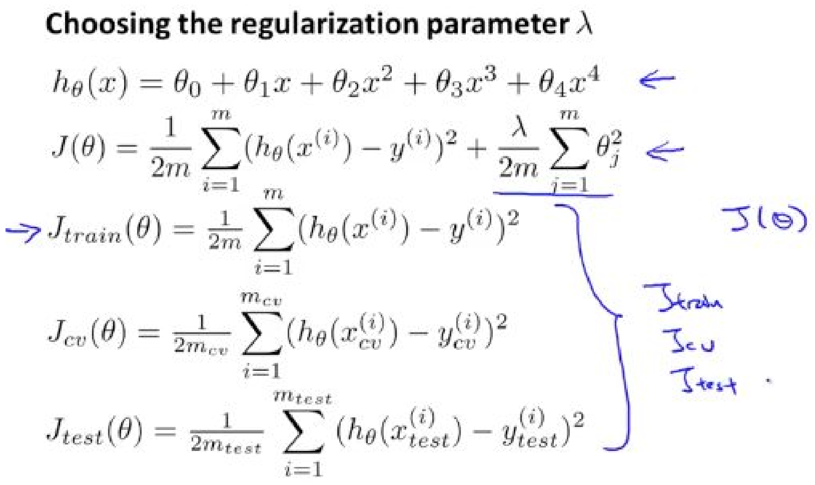
"Just right"

$\rightarrow$ Small λ

High variance (overfit)

$\rightarrow \lambda = 0$

- How can we then automatically select a good λ value? We start by changing our cost function definition as shown below. Our original cost function includes the regularization parameter. In this case, $J_{train}(\theta)$ and $J_{cv}(\theta)$ and $J_{test}(\theta)$ will NOT have a regularization param, they will just be the sum of the square of the errors * 1/(2m)



Choosing the regularization parameter λ

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4 \leftarrow$$

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \underline{\frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2} \leftarrow$$

$$\rightarrow J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

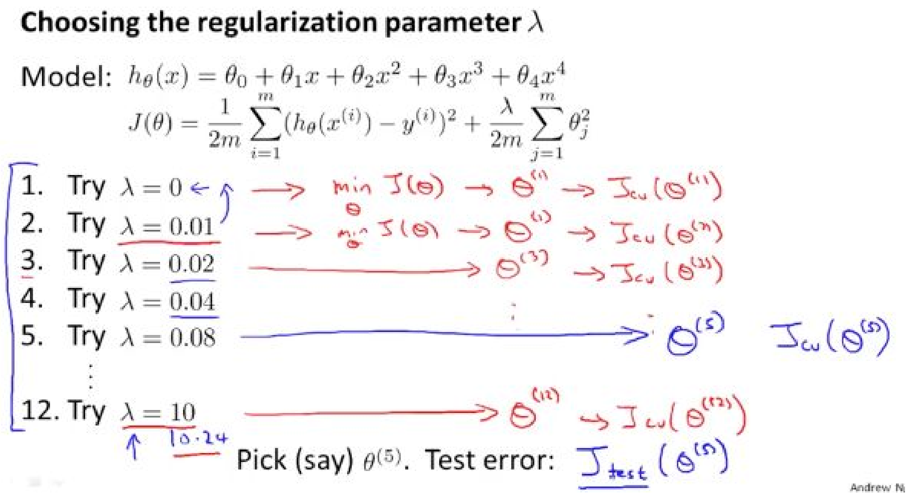
$$J_{cv}(\theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

$$J_{test}(\theta) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}(h_\theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$$

$J(\theta)$

J_{train}
J_{cv}
J_{test} .

- Once we have our cost functions for all 3 sections of the data set what we do is to create a list of possible λ values. Professor recommends that you start with a small value (say 0.01) and double it until you get to around 10. After having the list what we do is we try to minimize the original cost function (the one that has the regularization parameter) using all the λ values from the list and we end up with a list of possible parameters θ. We take each member of the list and apply it to our $J_{cv}(\theta)$ equation and we then select the cost function that minimizes the cross validation error($\theta^{(5)}$ in the example below). We take $\theta^{(5)}$ and apply it to $J_{test}(\theta)$ so we end up with $J_{test}(\theta^{(5)})$



Choosing the regularization parameter λ

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2$$

1. Try $\lambda = 0 \leftarrow \quad \rightarrow \min_\theta J(\theta) \rightarrow \theta^{(1)} \rightarrow J_{cv}(\theta^{(1)})$
2. Try $\underline{\lambda = 0.01} \quad \rightarrow \min_\theta J(\theta) \rightarrow \theta^{(2)} \rightarrow J_{cv}(\theta^{(2)})$
3. Try $\lambda = \underline{0.02} \quad \rightarrow \theta^{(3)} \rightarrow J_{cv}(\theta^{(3)})$
4. Try $\lambda = \underline{0.04}$
5. Try $\lambda = 0.08 \quad \rightarrow \theta^{(5)} \quad J_{cv}(\theta^{(5)})$

   ⋮

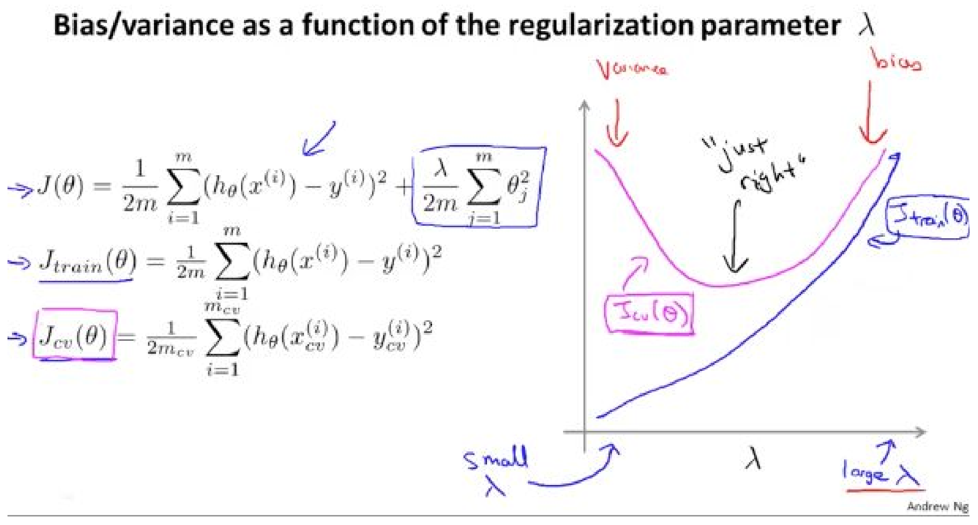
12. Try $\lambda = 10 \quad \rightarrow \theta^{(12)} \rightarrow J_{cv}(\theta^{(12)})$

$\uparrow$ 10.24

Pick (say) $\theta^{(5)}$. Test error: $J_{test}(\theta^{(5)})$

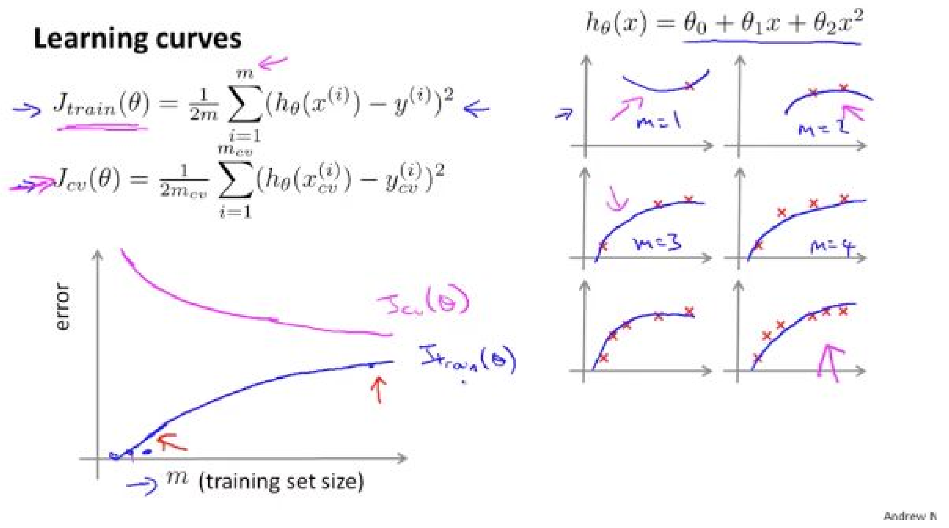
Andrew Ng

- We can then try to graph $J_{train}(\theta)$ and $J_{cv}(\theta)$ against λ and we end up with a chart as shown below. If λ is big it will penalize all my $\theta's$ and I end up with $h_\theta(x) = \theta_0$ which is then again a simple hypothesis with high bias so my chart below has the bias regime on the right. Conversely, if λ is low then I end up with the original high order polynomial and am therefore in the high variance (overfit) area. Just as in the previous chart there is a portion where the λ value is 'just right'



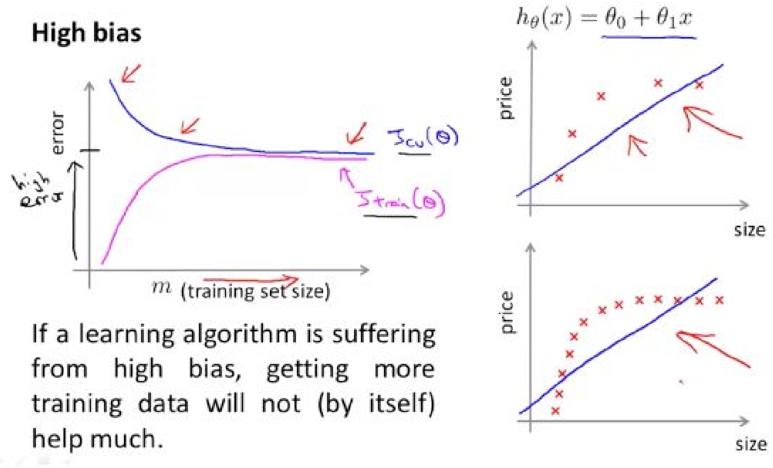
Learning Curves

- Learning curves are useful to check if the algorithm is working correctly or if you want to improve its performance. The learning curves for $J_{train}(\theta)$ and $J_{cv}(\theta)$ are shown below. The intuition is that if we have a small set of training examples (say 1-3) we will be able to fit $h_\theta(x)$ to the data so our error is 0 or very close to 0. However, as m starts to increase it gets harder to fit the dataset and our error increases.



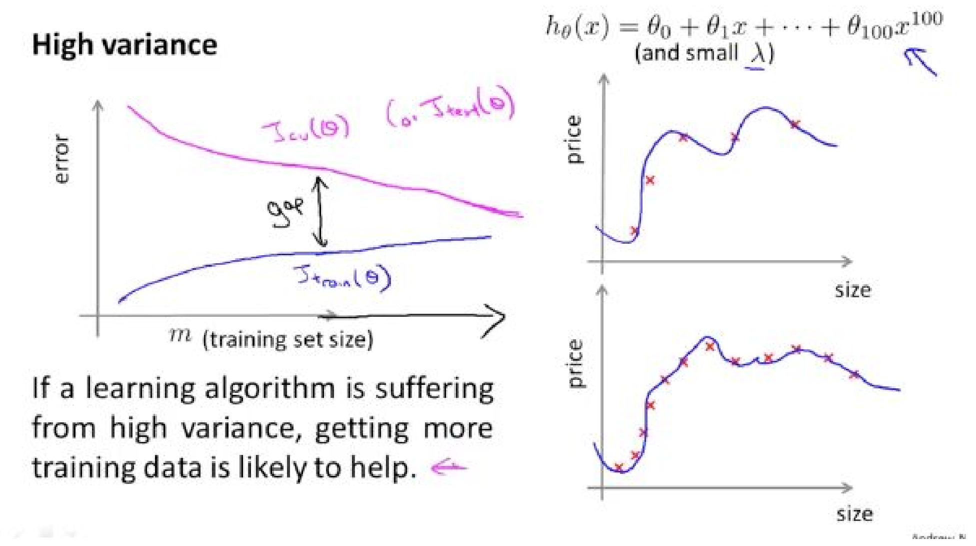
  Cross validation is different. We trained on a small number of examples so we don't expect our generalization to be very good therefore our error will be high. However, as we get more training examples we expect our cross validation error to decrease.
- Now let's look at how the learning curves change when we have high bias or high variance. Let's start with high bias. Suppose we have a hypothesis like the one below which is a line. What we see in this case is that we start with a small amount of data but

as we increase the amount of data we end up with kind of the same line. This means that initially I will have a small error for $J_{train}(\theta)$ and the error increases and kind of plateaus as we get a bigger m. For $J_{cv}(\theta)$ is kind of the opposite: it starts with a high error because we have started training with a small *m* and as the *m* gets bigger we get kind of the best fit for the data and our error decrease also plateaus. As m gets bigger $J_{train}(\theta)$ and $J_{cv}(\theta)$ kind of converge. What this also tells us is if you are underfitting (high bias) your data you will not get any significant improvement from getting from training data (potentially saving us days/months of work of data collection that will not work)



If a learning algorithm is suffering from high bias, getting more training data will not (by itself) help much.

- Now let's take a look at high variance (overfitting). If we are overfitting we have a high order polynomial. In this case, graphing our error against m will show us that
  - $J_{train}(\theta)$ will be initially low and will remain low as the training size increases. Makes sense since we are overfitting for this training set and the error will then be low.
  - $J_{cv}(\theta)$ will be initially high which also makes sense because overfitting for one training set does not generalize to other data sets. As we get bigger training sizes the cross validation error goes down.

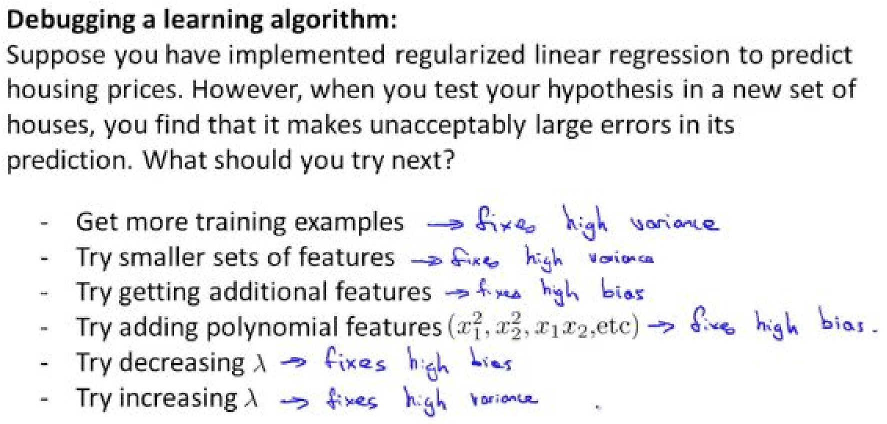


If a learning algorithm is suffering from high variance, getting more training data is likely to help.

In a case like this getting more training data will help because it will help us to see if $J_{cv}(\theta)$ continues to decrease. Finally, we can identify if the algorithm suffers from high variance if we see a relatively big gap between the $J_{cv}(\theta)$ and $J_{train}(\theta)$

Deciding what to do next Revisited

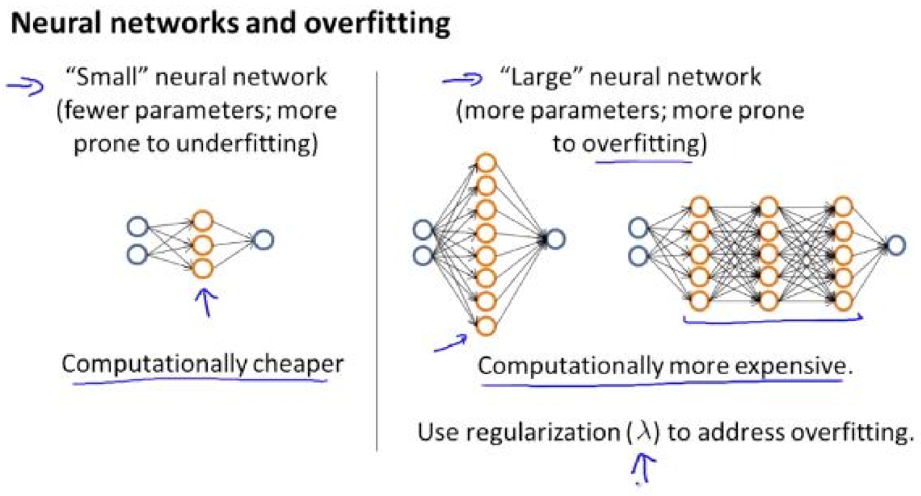
- Now we can go back to original problem and provide cases where the below options are going to be helpful
    - Getting more training examples is going to fix *high variance*
    - Trying a smaller set of features is going to fix *high variance*
    - Getting additional features is going to fix *high bias*
    - Adding more polynomial features is going to fix *high bias*
    - Decreasing λ is going to fix *high bias*
    - Increasing λ is going to fix *high variance*



Debugging a learning algorithm:
Suppose you have implemented regularized linear regression to predict housing prices. However, when you test your hypothesis in a new set of houses, you find that it makes unacceptably large errors in its prediction. What should you try next?

- Get more training examples $\rightarrow$ fixes high variance
- Try smaller sets of features $\rightarrow$ fixes high variance
- Try getting additional features $\rightarrow$ fixes high bias
- Try adding polynomial features $(x_1^2, x_2^2, x_1x_2, \text{etc})$ $\rightarrow$ fixes high bias.
- Try decreasing λ $\rightarrow$ fixes high bias
- Try increasing λ $\rightarrow$ fixes high variance

- Now let's apply what we learned to neural networks. A small neural network is prone to underfitting but is computationally cheap. A large neural network is prone to overfitting and computational expensive. We can use regularization to address overfitting



Neural networks and overfitting

"Small" neural network (fewer parameters; more prone to underfitting)

Computationally cheaper

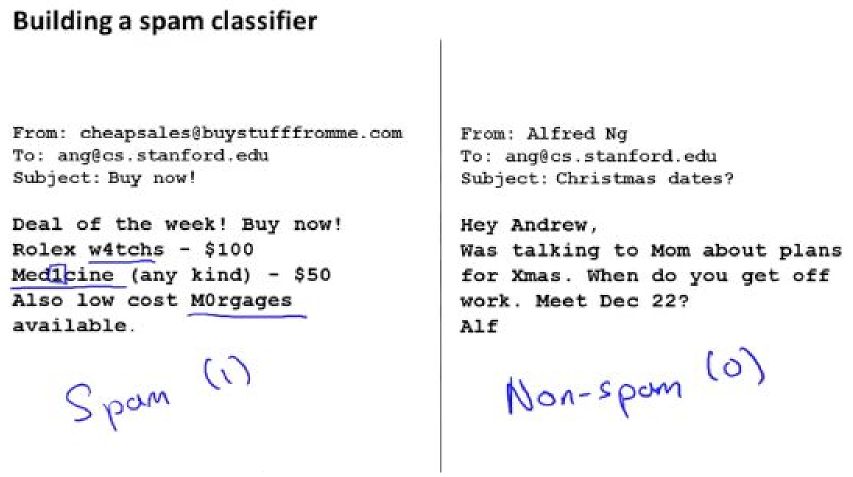
"Large" neural network (more parameters; more prone to overfitting)

Computationally more expensive.

Use regularization (λ) to address overfitting.

Building a Spam Classifier: Prioritizing what to work on

- We are going to try to build a spam classifier. We note some characteristics of spam emails (words that have numbers instead of letters). If an email is classified as spam we have a positive id (1), if non-spam we have a negative id (0).



Building a spam classifier

```
From: cheapsales@buystufffromme.com
To: ang@cs.stanford.edu
Subject: Buy now!

Deal of the week! Buy now!
Rolex w4tchs - $100
Med1cine (any kind) - $50
Also low cost M0rgages
available.
```

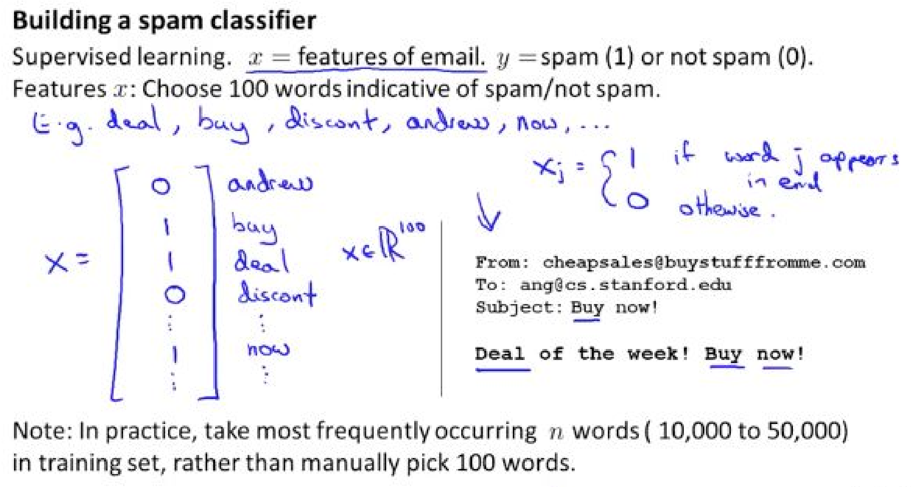
Spam (1)

```
From: Alfred Ng
To: ang@cs.stanford.edu
Subject: Christmas dates?

Hey Andrew,
Was talking to Mom about plans
for Xmas. When do you get off
work. Meet Dec 22?
Alf
```

Non-spam (0)

- Now we need to create the X vector of features. In this case X will be 100 words indicative of spam or not spam. We order them alphabetically and add a 1 or 0 in our X vector if the word appears on an email message. We are then saying that $x_j = 1 \; if \; word \; j \; appears \; in \; email, \; x_j = 0 \; otherwise$. In practice, we don't chose 100 words, we take the most frequently occurring n words (10,000 to 50,000)



Building a spam classifier

Supervised learning. $x =$ features of email. $y =$ spam (1) or not spam (0).

Features x: Choose 100 words indicative of spam/not spam.

E.g. deal, buy, discont, andrew, now, ...

$$x = \begin{array}{c|l} \left[\begin{matrix} 0 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 1 \\ \vdots \end{matrix}\right] & \begin{matrix} \text{andrew} \\ \text{buy} \\ \text{deal} \\ \text{discont} \\ \vdots \\ \text{now} \\ \vdots \end{matrix} \end{array} \quad x \in \mathbb{R}^{100}$$

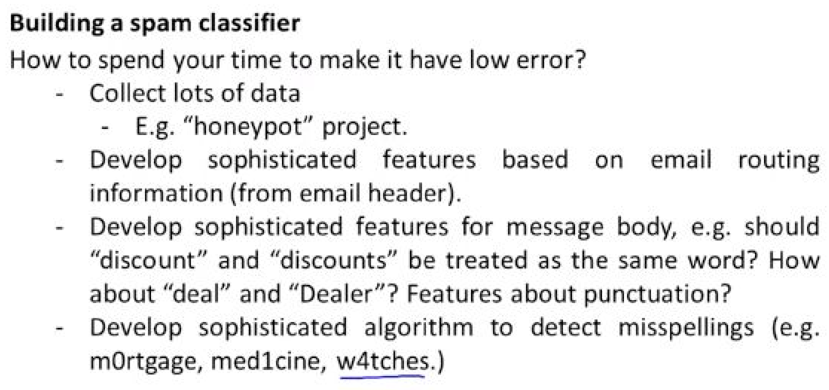
$$x_j = \begin{cases} 1 & \text{if word } j \text{ appears in email} \\ 0 & \text{otherwise.} \end{cases}$$

```
From: cheapsales@buystufffromme.com
To: ang@cs.stanford.edu
Subject: Buy now!

Deal of the week! Buy now!
```

Note: In practice, take most frequently occurring n words (10,000 to 50,000) in training set, rather than manually pick 100 words.

- Then we try to answer the question of what is the best use of my time if I want to have a classifier with low error? Some options are
  - Collect lots of data, i.e. honeypot projects
  - Develop sophisticated features based on email header routing info
  - Develop sophisticated features for for message body
  - Develop sophisticated algorithms to detect misspellings



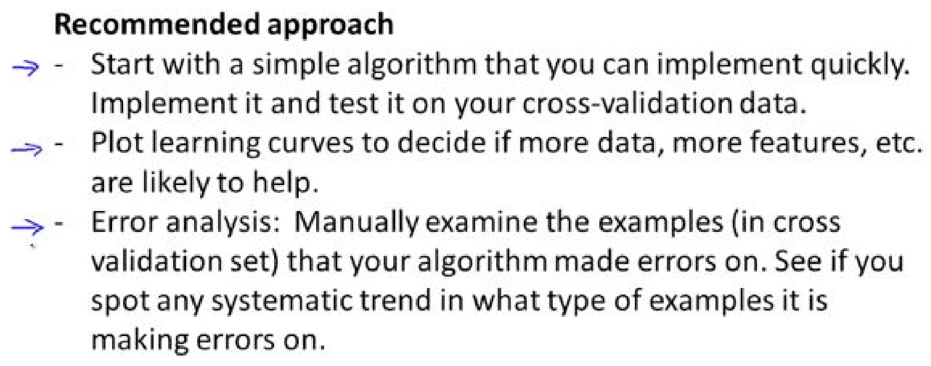
Building a spam classifier

How to spend your time to make it have low error?

- Collect lots of data
  - E.g. "honeypot" project.
- Develop sophisticated features based on email routing information (from email header).
- Develop sophisticated features for message body, e.g. should "discount" and "discounts" be treated as the same word? How about "deal" and "Dealer"? Features about punctuation?
- Develop sophisticated algorithm to detect misspellings (e.g. m0rtgage, med1cine, w4tches.)

Building a Spam Classifier: Error Analysis

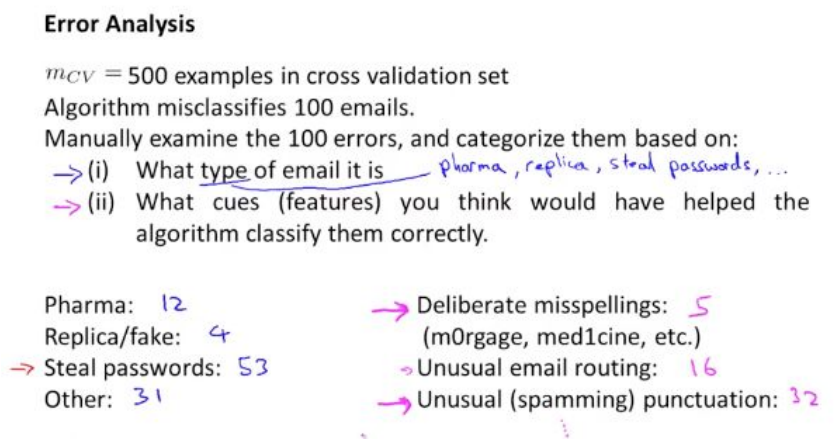
- When faced with a new machine learning problem we can start with the approach detailed below: start with a simple algo, implement it and test it on the cv data. Plot learning curves to decide what to do next and, finally, examine the examples where the algo made errors to see if we can spot a trend in what examples we are making errors on. In this class we are going to concentrate in Error Analysis, the last step.



Recommended approach

- Start with a simple algorithm that you can implement quickly. Implement it and test it on your cross-validation data.
- Plot learning curves to decide if more data, more features, etc. are likely to help.
- Error analysis: Manually examine the examples (in cross validation set) that your algorithm made errors on. See if you spot any systematic trend in what type of examples it is making errors on.

- Here's an example of how we use Error Analysis: in a cross validation set of 500 examples, 100 were misclassified. We manually examine the errors and categorize them by type of email and see if we have any clues on what features could have helped in classifying those samples correctly. The manual classification allows us to understand the type of email that is giving us trouble along with the feature(s) that is also fooling the algorithm. In the example below is 'steal passwords' and 'unusual punctuation'. This is why it pays to use a simple algorithm: with it we can quickly see what type of emails and/or features are giving us trouble.

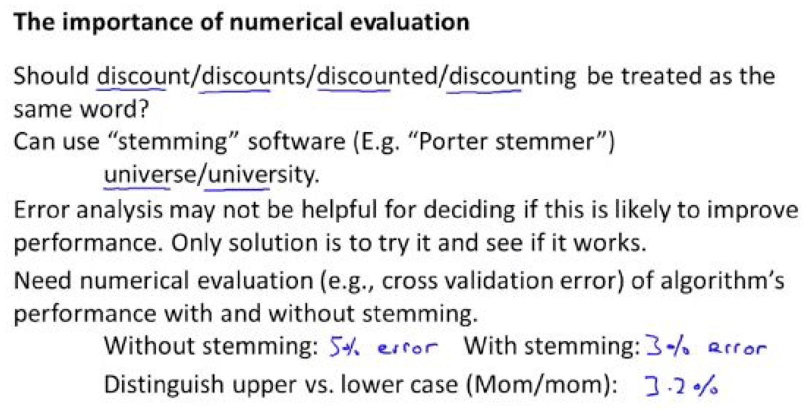


Error Analysis

m_{CV} = 500 examples in cross validation set
Algorithm misclassifies 100 emails.
Manually examine the 100 errors, and categorize them based on:
$\rightarrow$ (i) What type of email it is pharma, replica, steal passwords, ...
$\rightarrow$ (ii) What cues (features) you think would have helped the algorithm classify them correctly.

Pharma: 12	$\rightarrow$	Deliberate misspellings: 5 (m0rgage, med1cine, etc.)
Replica/fake: 4	$\rightarrow$	Unusual email routing: 16
$\rightarrow$ Steal passwords: 53	$\rightarrow$	Unusual (spamming) punctuation: 32
Other: 31		

- Next, we will be faced with some decisions where error analysis will not help. In such cases, it is important to get our error result as a single, numerical value. Otherwise, it is hard to assess the algo's performance. We should try new things, get their error rate and based on that result decide if we want to use the new feature or not.

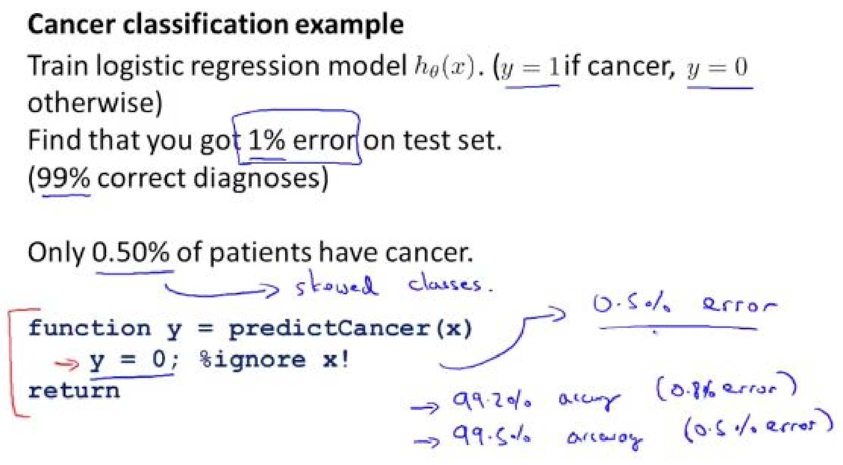


The importance of numerical evaluation

Should discount/discounts/discounted/discounting be treated as the same word?
Can use "stemming" software (E.g. "Porter stemmer")
universe/university.
Error analysis may not be helpful for deciding if this is likely to improve performance. Only solution is to try it and see if it works.
Need numerical evaluation (e.g., cross validation error) of algorithm's performance with and without stemming.
Without stemming: 5% error With stemming: 3% error
Distinguish upper vs. lower case (Mom/mom): 3.2%

Error Metrics for Skewed Classes

- Skewed classes are problems where we have way more examples from one class than from the other class as shown below with the cancer classification example. In this case we had an algorithm that achieved 99% correctness/accuracy but a non-learning algorithm can do better: 0.5% error. It then becomes harder to evaluate if we did a useful change to the algorithm or we just replaced it with a non-learning one? It is then harder to just use classification accuracy because we can get high classification accuracy with an algorithm that does not look like such a good choice. We then need a different error/evaluation metric. One such metric is called Precision/Recall.



Cancer classification example

Train logistic regression model $h_\theta(x)$. ($y = 1$ if cancer, $y = 0$ otherwise)

Find that you got 1% error on test set.

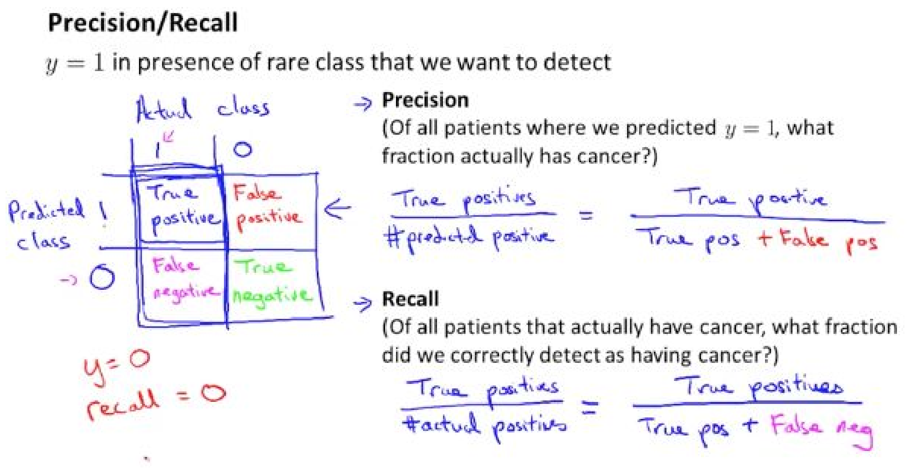
(99% correct diagnoses)

Only 0.50% of patients have cancer.

```
function y = predictCancer(x)
  y = 0; %ignore x!
return
```

- To define precision recall we start with an example where $y = 1$ (i.e. a tumor is malignant). In this case we can compute a table like the one below using the following cases:
  - When we predicted y = 1 or y =0
  - When the actual y = 1 or 0

If we predicted y=1 and y was actually =1 we have a true positive.
If we predicted y=0 and y was actually =0 we have a true negative.
If we predicted y=1 but y was actually = 0 we have a false positive.
If we predicted y=0 but y was actually = 1 we have a false negative.



Precision/Recall

$y = 1$ in presence of rare class that we want to detect

Precision
(Of all patients where we predicted $y = 1$, what fraction actually has cancer?)

Recall
(Of all patients that actually have cancer, what fraction did we correctly detect as having cancer?)

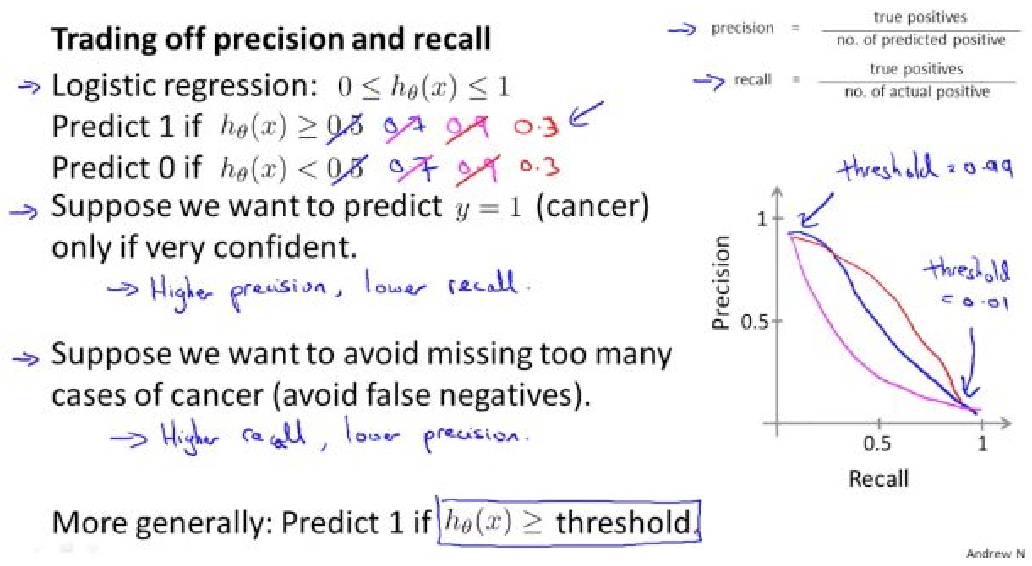
Actual class
1
0
Predicted class
1
0
True positive
False positive
False negative
True negative
True positives / #predicted positive = True positive / True pos + False pos
True positives / #actual positives = True positives / True pos + False neg
y=0
recall = 0

We now can compute
- $Precision = True\,Positive/(True\,Positive + False\,Positive)$ and
- $Recall = True\,Positive\,/\,(True\,Positive + False\,Negative)$

Finally, if we have an algorithm where y =0 all the time then $Recall = 0$ because it will not have any True Positives. Now we can see Precision/Recall is helping us to see if an algorithm is 'cheating' and gives us high confidence that we have a good algorithm.

Trading off Precision and Recall

- There is a trade off to be made between precision and recall. To illustrate it let's continue with our cancer example. We start with a prediction threshold of 0.5. First we consider that we want to predict y=1 only if we are very confident, we then change our threshold to something higher, say 0.7. In this case we have a classifier with higher precision but lower recall. Now let's consider the opposite case: we want to avoid missing too many cases of cancer so we lower the threshold to say 0.3. In this case we decrease our precision but increase our recall.



Trading off precision and recall

→ Logistic regression: $0 \le h_\theta(x) \le 1$

Predict 1 if $h_\theta(x) \ge \cancel{0.5}$ ~~0.7~~ ~~0.9~~ 0.3

Predict 0 if $h_\theta(x) < \cancel{0.5}$ ~~0.7~~ ~~0.9~~ 0.3

→ Suppose we want to predict $y = 1$ (cancer) only if very confident.

→ Higher precision, lower recall.

→ Suppose we want to avoid missing too many cases of cancer (avoid false negatives).

→ Higher recall, lower precision.

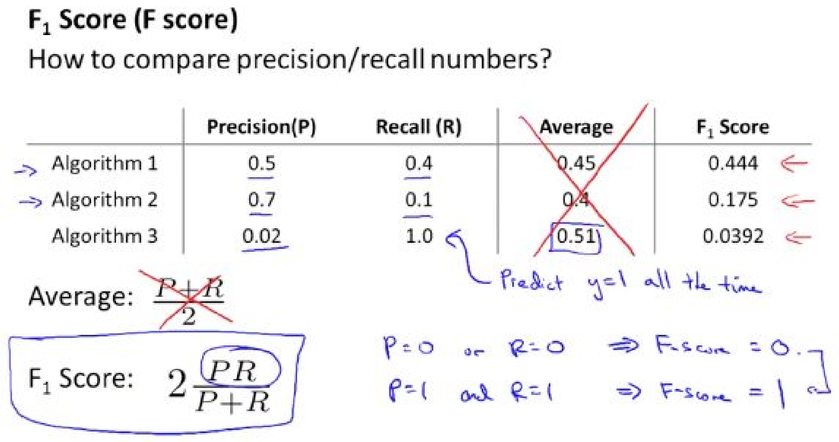
More generally: Predict 1 if $\boxed{h_\theta(x) \ge \text{threshold}}$

→ precision = $\frac{\text{true positives}}{\text{no. of predicted positive}}$

→ recall = $\frac{\text{true positives}}{\text{no. of actual positive}}$

A couple of different precision curves is shown on the chart above. We generalize this trade off as predict 1 if $h_\theta(x) \ge threshold$.

- Next question is how do we compare precision/recall numbers? The problem with precision/recall is that now we have 2 numbers to compare whereas before we insisted in the importance of having a single number to evaluate. The solution is to use the $F_1 score$ defined as $2\frac{PR}{P+R}$. This score gives us a good way to evaluate our algorithm and takes care of cases where we have a not very good classifier (i.e. one that always predict y=1 or y=0). It is tempting to use the average but the $F_1 score$ is better



F_1 Score (F score)

How to compare precision/recall numbers?

	Precision(P)	Recall (R)	~~Average~~	F_1 Score
→ Algorithm 1	0.5	0.4	~~0.45~~	0.444 ←
→ Algorithm 2	0.7	0.1	~~0.4~~	0.175 ←
Algorithm 3	0.02	1.0 ← Predict y=1 all the time	~~$\boxed{0.51}$~~	0.0392 ←

Average: $\cancel{\frac{P+R}{2}}$

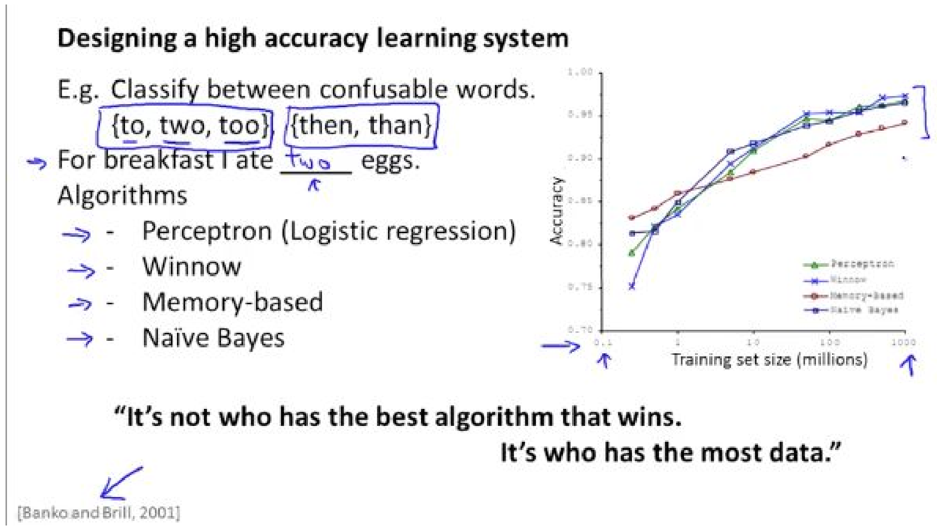
$\boxed{F_1 \text{ Score: } 2\frac{PR}{P+R}}$

P=0 or R=0 ⇒ F-score = 0.

P=1 and R=1 ⇒ F-score = 1

Data for Machine Learning

- How much data should we train on? Let’s start with an influential study by Banko and Brill that showed that the 4 algorithms shown below benefited greatly from having more data available to them. Other studies have shown the same result. This fact has led to the saying ‘It’s not who has the best algorithm that wins, it’s who has the most data’.



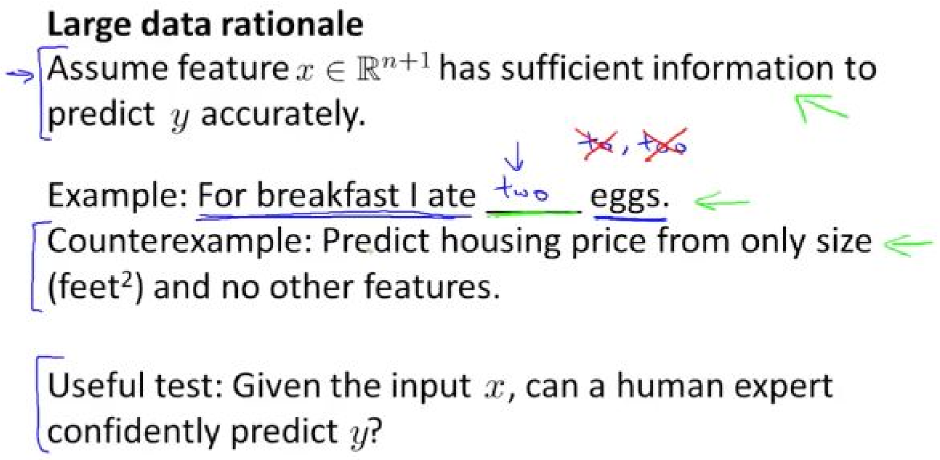
Designing a high accuracy learning system

E.g. Classify between confusable words.
$\boxed{\{\text{to, two, too}\}}$ $\boxed{\{\text{then, than}\}}$
→ For breakfast I ate two eggs.
Algorithms
→ - Perceptron (Logistic regression)
→ - Winnow
→ - Memory-based
→ - Naïve Bayes

“It’s not who has the best algorithm that wins.
It’s who has the most data.”

[Banko and Brill, 2001]

- Now let’s see the assumptions under which a massive set of data will help. The rationale says that feature x has sufficient information to predict y accurately. An example supporting this rationale is the ‘fill in the blank” example below. A counterexample is the housing price prediction with only size. Professor Ng uses the following way to test this rationale : given the feature x, would a human expert confidently predict y?



Large data rationale

→ Assume feature $x \in \mathbb{R}^{n+1}$ has sufficient information to predict y accurately.

~~to~~, ~~too~~

Example: For breakfast I ate two eggs.
Counterexample: Predict housing price from only size (feet^2) and no other features.

Useful test: Given the input x, can a human expert confidently predict y?

- Let’s see when having a lot of data helps. We have a learning algorithm with many parameters -> ensures low bias which means $J_{train}(\theta)$ will be small. Now assume we use a huge training set. With such a large training set (unlikely to overfit) we will have low variance and, therefore, $J_{train}(\theta) \approx J_{test}(\theta)$ and $J_{test}(\theta)$ will be small

Large data rationale

→ Use a learning algorithm with many parameters (e.g. logistic regression/linear regression with many features; neural network with many hidden units). low bias algorithms.

→ $J_{train}(\theta)$ will be small.

Use a very large training set (unlikely to overfit)

→ $J_{train}(\theta) \approx J_{test}(\theta)$

→ $J_{test}(\theta)$ will be small