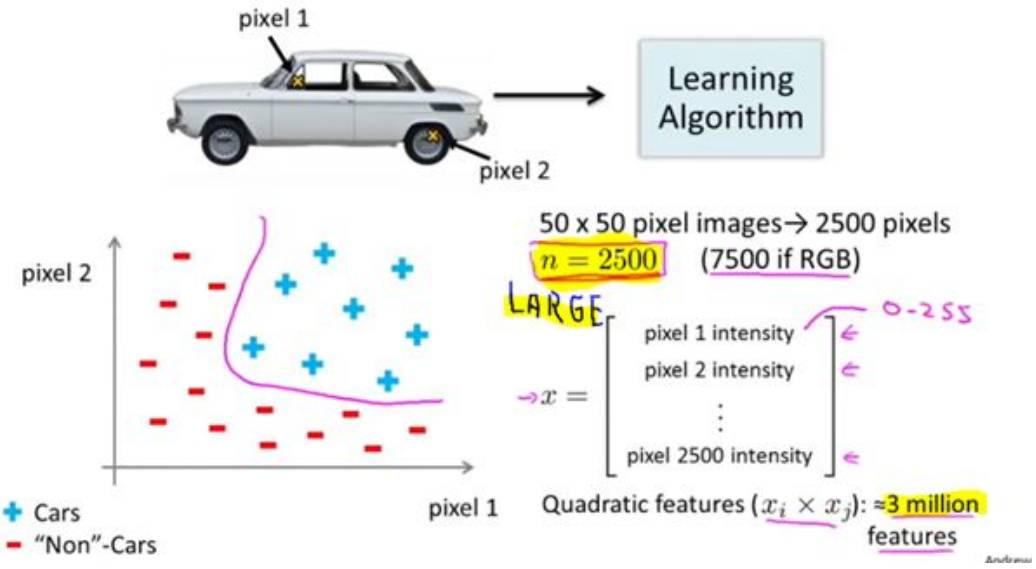
~Machine Learning Notes (part 2)~
Prof. Andrew Ng.

Week 4

Part VIII (1) - Non-Linear Hypothesis:

A **non-linear** hypothesis is be needed to fit a good model to a complex problem.

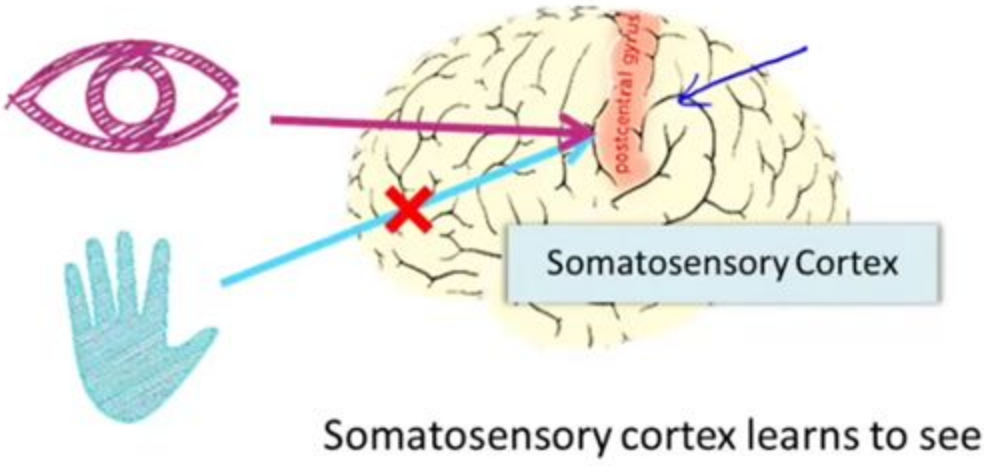
When the number of features is large (like the following example, where number of features is 2500), then the number of terms in the hypothesis could grow dramatically (to the order of O(n^2) or even O(n^3)). This is when another learning algorithm **Neural Network** is needed.



Part VIII (2) – Neural Network:

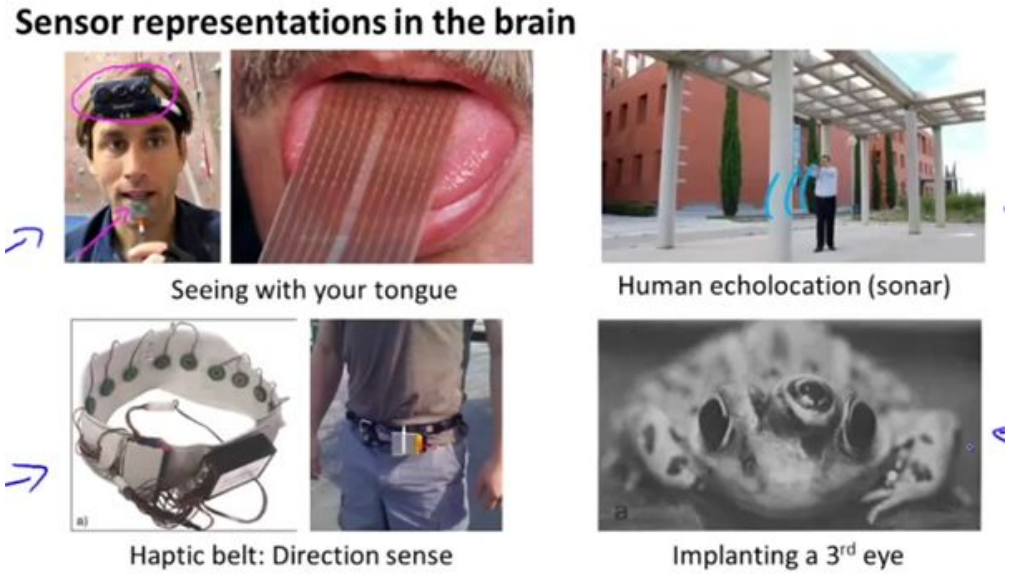
Neural Network is a learning algorithm that originates from trying to mimic the brain.
The brain is capable of taking in any types of input and processing it in a way that is usable for the subject. For example, the Somatosensory Cortex that responsible for sensing touch can interpret visual input when it is "rewired".

The "one learning algorithm" hypothesis



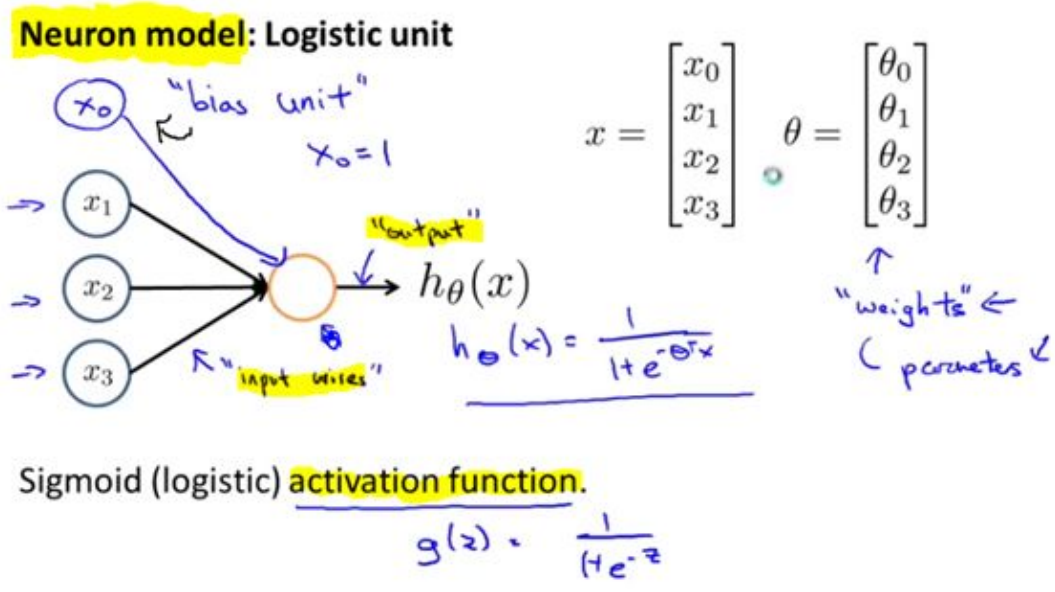
Somatosensory cortex learns to see

Some other examples (below) that the brain can be trained to process different type of inputs in non-ordinary situations suggest that: The brain only uses ONE single learning algorithm to perform the executions of all these tasks, which is what **Neural Network** tries to achieve.

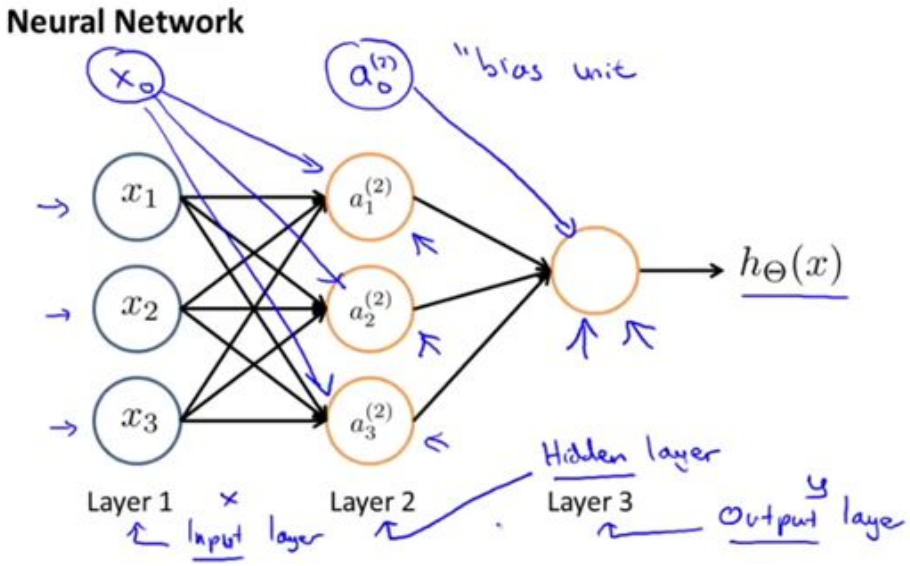


Part VIII (3~4) – Neuron Model representation:

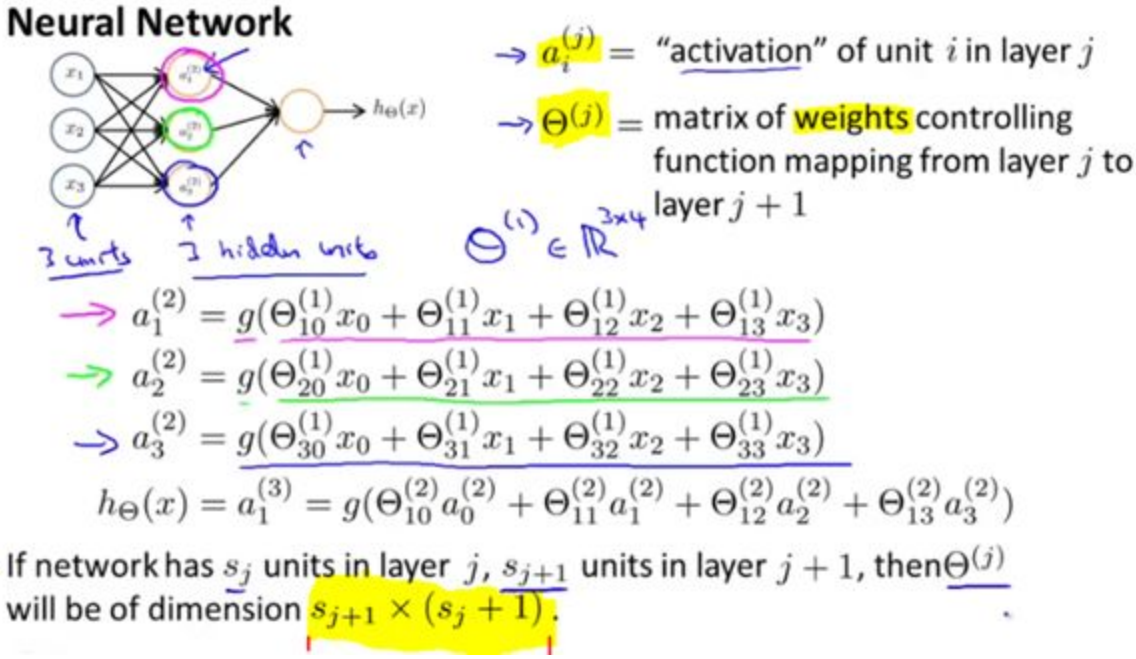
A simple Neuron Model representation consist of the "**Input wires**" (x) which act as Dendrites in actual neuron, the **Body** (hidden), and the "**output wires**" like the axons, all these results in giving the hypothesis h(x).



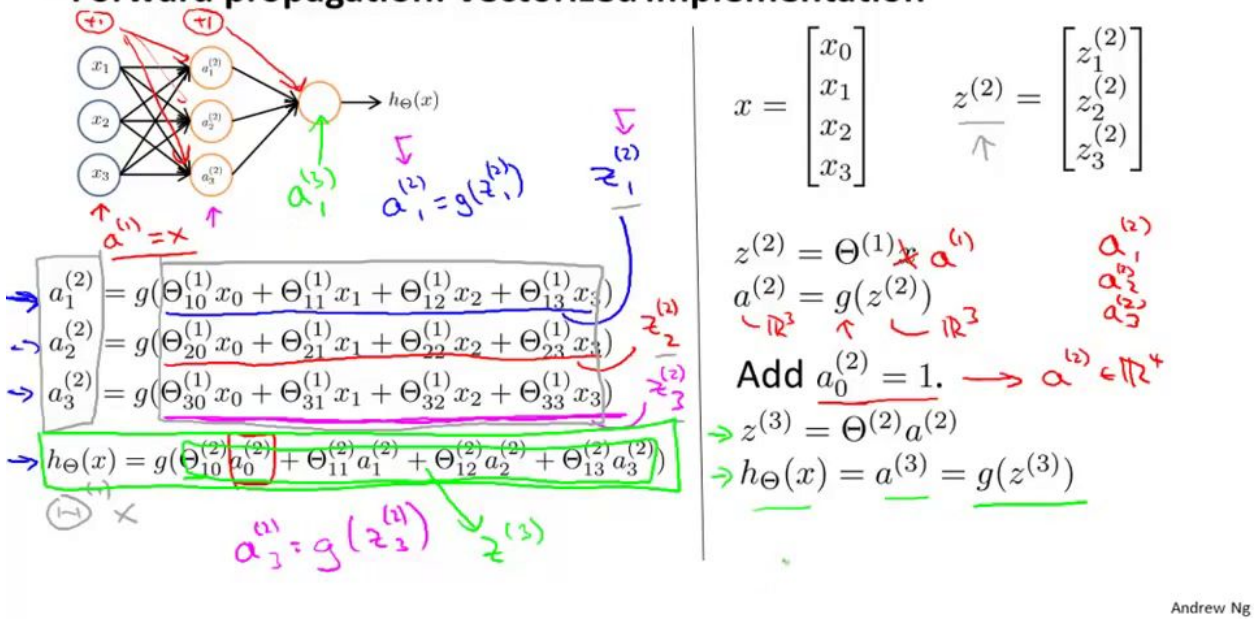
Putting together a group of simple Neuron makes a **Neural Network**:



Mathematical representation of the model:



Forward propagation: Vectorized implementation



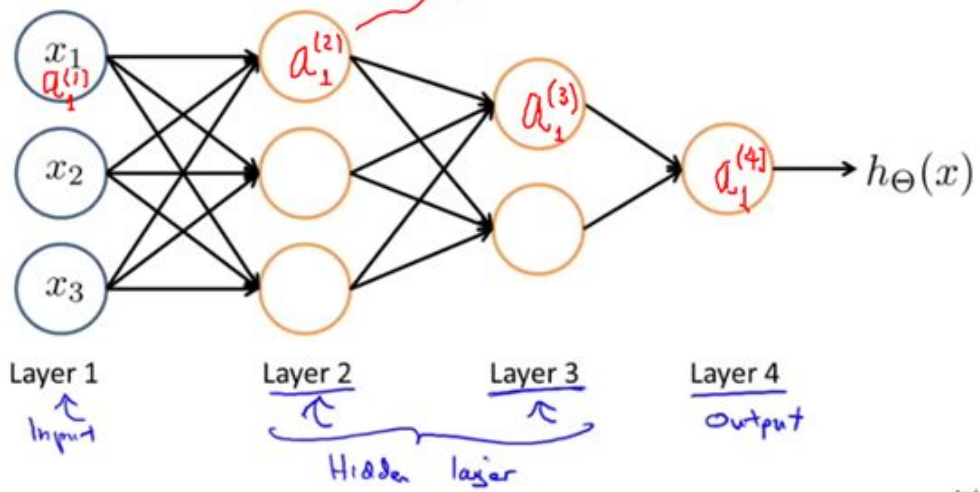
Andrew Ng

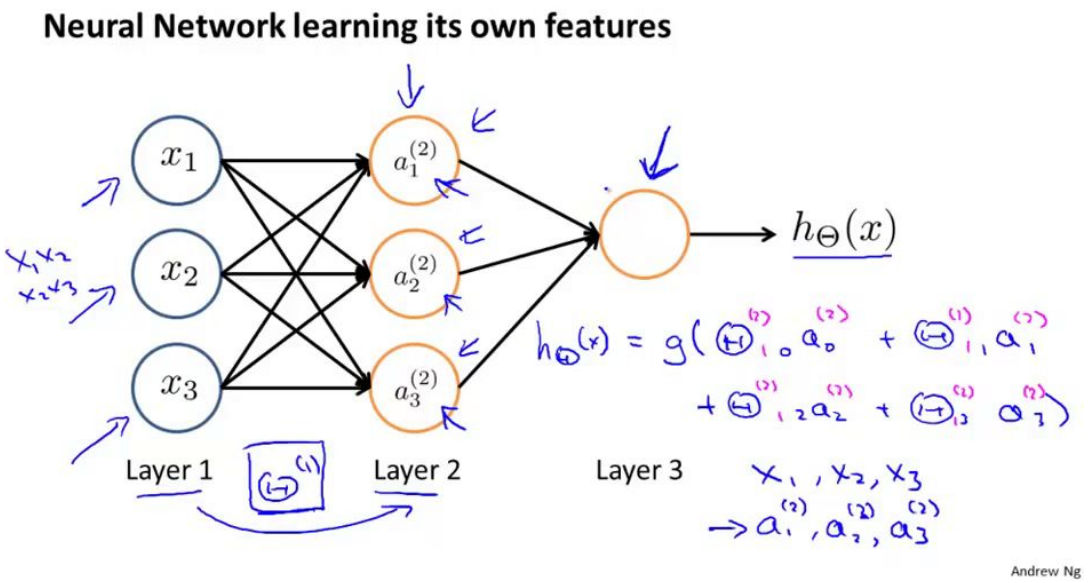
Note: Row of Theta of layer j = # of units in layer j+1(1st subscript index). Column of Theta of layer j = # of units in layer j + 1 (for bias unit) (2nd subscript index).

$$a_i^{(j)} = g(z_i^{(j)}) = g(\Theta^{(j-1)} a^{(j-1)})$$

Other network architectures

$g(z_1^{(2)})$





Each node in layer 2 is learned from layer 1 with parameters of mapping from layer 1 to 2. $\Theta^{(1)}$

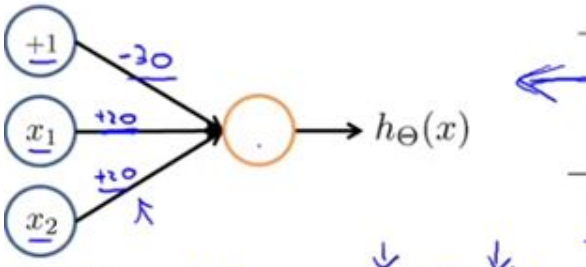
Part VIII (5) Examples of Neural Network (logical function):

The AND function: y = 1 only when **both** x1 and x2 are of value 1.

Simple example: AND

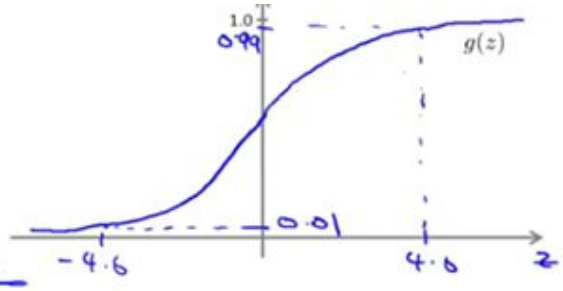
$x_1, x_2 \in \{0, 1\}$

$y = x_1$ AND x_2



$h_\Theta(x) = g(-30 + 20x_1 + 20x_2)$

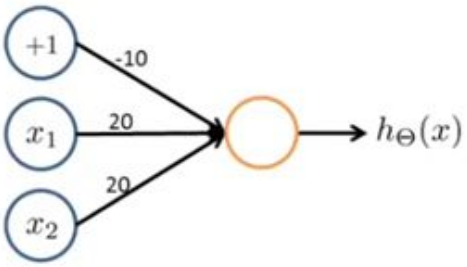
$\Theta^{(1)}_{10}$ $\Theta^{(1)}_{11}$ $\Theta^{(1)}_{12}$



x_1	x_2	$h_\Theta(x)$
0	0	$g(-30) \approx 0$
→ 0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

The OR function: y = 1 when **either** x1 or x2 are of value 1.

Example: OR function

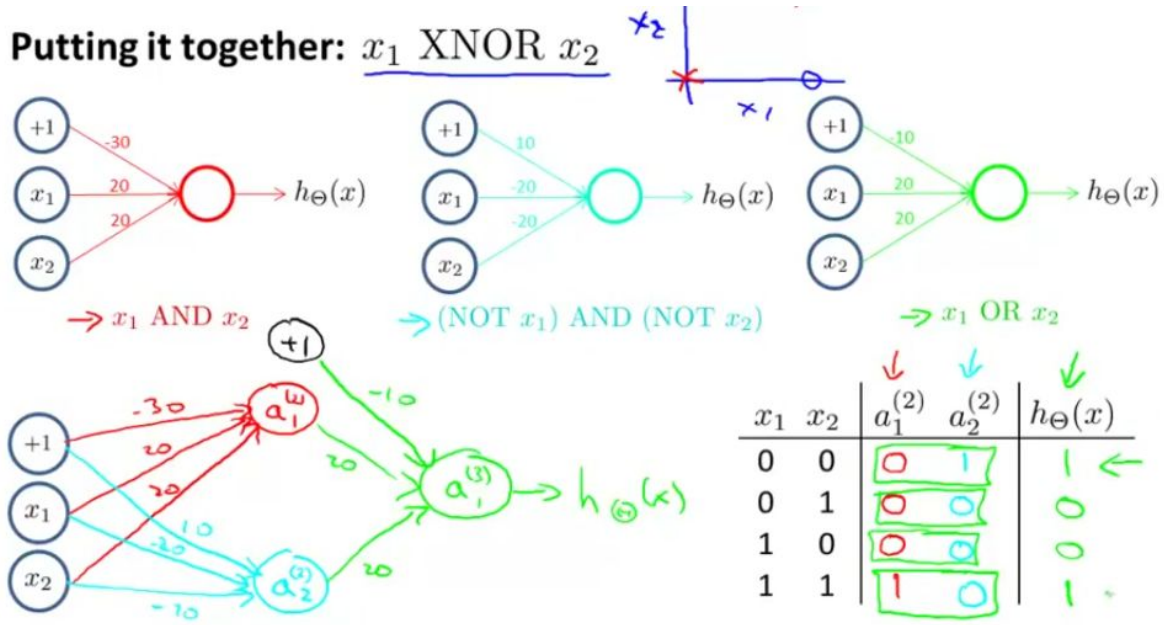


$g(-10 + 20x_1 + 20x_2)$

x_1	x_2	$h_\Theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	≈ 1
1	1	≈ 1

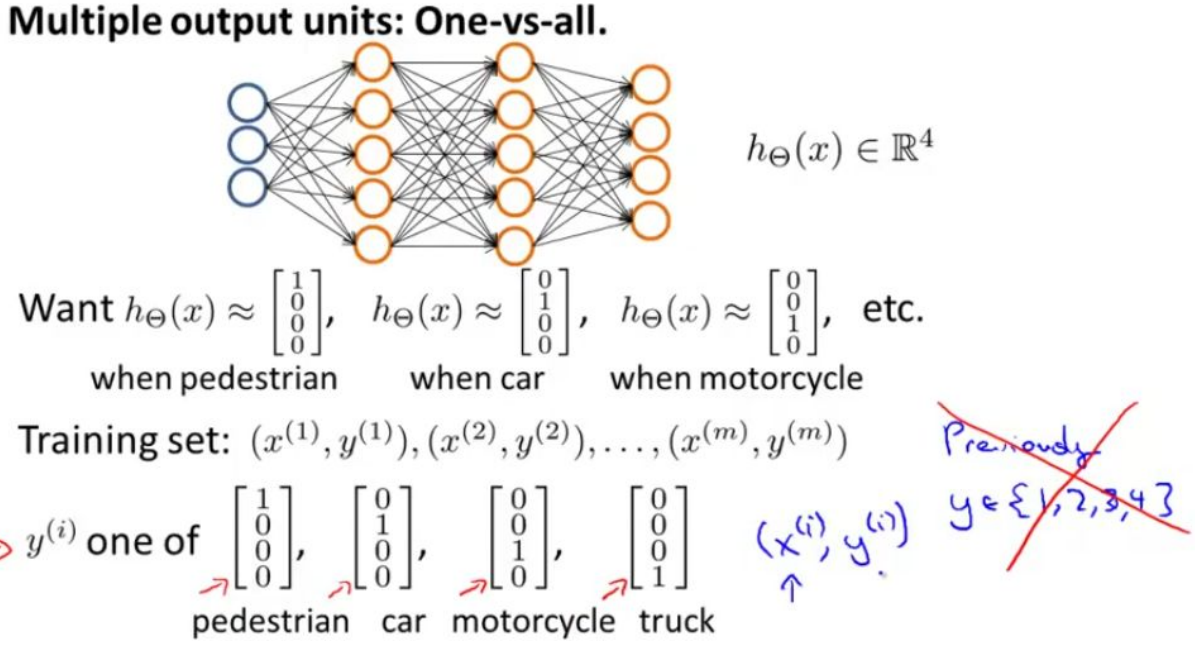
Notice how the implementation differs in value of the parameters.

Part VIII (6) Examples of Neural Network II :



This network putting together 3 hypothesis in 2 layers to output more complicated logical function.

Part VIII (7) Multiclass Classification (one vs all):



==========

Week 5

Part IX (1) Cost Function:

Notations:

- L: number of layers in the neural network.
- S_l: number of units in layer l.
- K: number of classes in the problem. (i.e. y & the output units are K dimensional vectors).

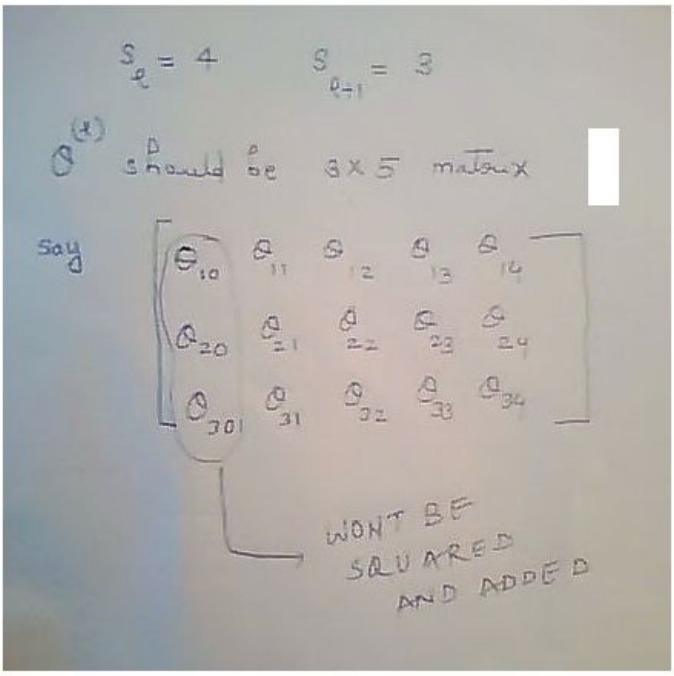
Neural network:

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

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

Note: $\Theta^{(l)}_{ji}$ means the parameter in layer l for jth feature, of the ith row.

The j = 0 terms of theta will not be squared and summed up.



Part IX (2) Minimizing Cost Function (Backpropagation):

To minimize cost function, the derivative terms of the cost function with respect to its parameter have to be calculated first.

"Error" of the final output layer $\delta^{(L)}$ is being compute by:

$$\delta^{(L)} = a^{(L)} - y \text{ (vectorized)}$$

Then working backward, the error for preceding layers are computed by:

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

$a^{(3)} .* (1 - a^{(3)})$

$a^{(2)} .* (1 - a^{(2)})$

Note: there are no $\delta^{(1)}$, as it is input layer.

Summing all the steps:

Backpropagation algorithm

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

Set $\Delta_{ij}^{(l)} = 0$ (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 $\delta^{(L)} = a^{(L)} - y^{(i)}$

→ Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$

→ $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$.

→ $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$

→ $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$ if $j = 0$

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

***

Part IX (4) Unrolling matrices for implementation:

Advance optimization algo assume "intialTheta" that gets feed in and "gradientVec" that gets return are **vector**, therefore conversion back & forth between matrices & vectors is needed.

To unroll:

Example

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

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

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

$\Theta^{(1)}$ $\Theta^{(2)}$ $\Theta^{(3)}$

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

To retrieve original matrices (reshape):

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

Learning Algorithm

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

Unroll to get `initialTheta` to pass to

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

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

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

***

Part IX (5) Gradient Checking:

Backprop often need to be checked in order to make sure that it's working correctly. This can be done by approximate the gradient by using the secant method and compare it to the output of backprop.

Implementation:

```
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;
```

After checking, be sure to **disable gradient checking** before training the classifier by backprop, because it's very slow.

***

Part IX (6) Random initialization of Theta:

If we initialize Initial theta to be all zeros when training neural network, means that theta mapping to each unit of next layer will be identical(symmetric weight), resulting in identical activations value.

So we have to randomly assign initial theta for each element of Big Theta:

Random initialization: Symmetry breaking

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

E.g.

random 10×11 matrix (betw. 0 and 1)

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

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

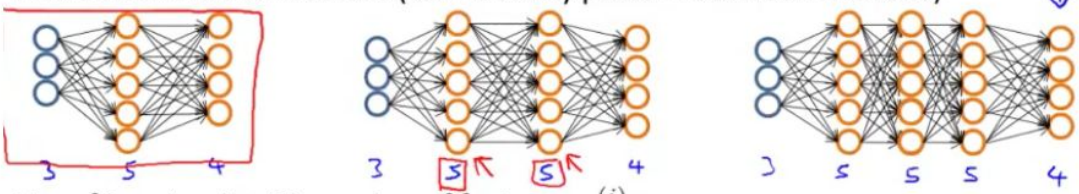
[-ε, ε]

***

~Summary~

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)

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

Training a neural network

5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ 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

***

Week 6

Part X (1-2) - Hypothesis evaluation:

When a Learnt Algorithm fails to work, the following can be applied to improve:

- Get more training examples.
- Try smaller/larger set of features (overfitting/underfitting).
- More Polynomial features.
- Increase/decrease λ.

But to decide which to apply, a Machine Learning Diagnostic procedure can help to gain insight of what is/isn't working and how to improve.

First Evaluate a Hypothesis - split dataset into 2 portion (e.g. 70%-30% split after random shuffling), the 1st portion is the Training set, 2nd portion is the Test set. If Hypothesis is overfitting, Training error will be low, while Test error will be high.

Calculate errors as usual according to types of problems (Linear or Logistic).
For Logistic, there's a special types of error computation:

Misclassification error (0/1 misclassification error):

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

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

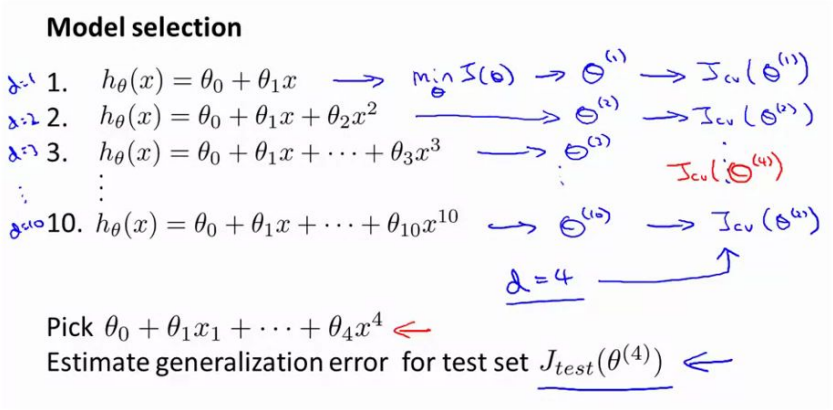
Part X (3) - Model Selection:

When choosing between models, such as different degree of polynomial (d):

- Split the data into 3 sets (Training/Cross Validation/Test): 60-20-20% after random shuffle.
- Train various models with the training set.
- Pick the best model (least errors with the Cross Validation (CV) set.
- Then test it with the Test set to check the error for determining the extent of generalization of the model.

Doing this will minimize bias decision from testing the model with the same data used for choosing model.

Below is an example where d = 4 is chosen:



Train/validation/test error

Training error:

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

$J(\theta)$

Cross Validation error:

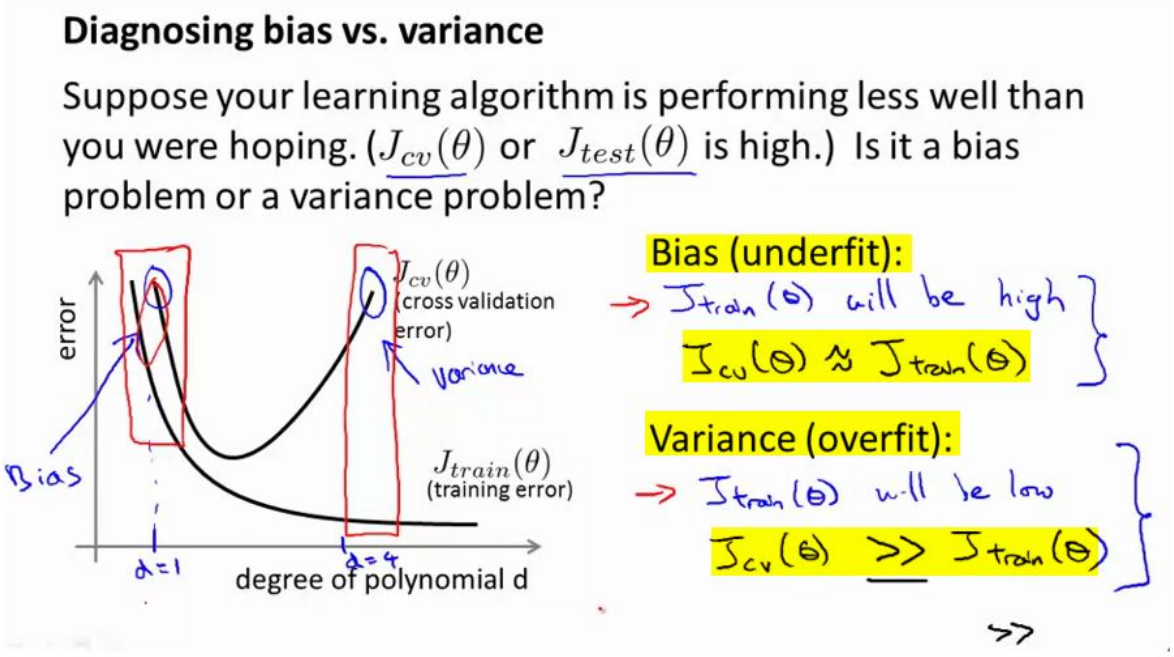
$$J_{cv}(\theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

Test error:

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

Part X (4) - Bias/Variance Diagnostic:

To determine whether a model is suffering High Bias (underfitting) or High Variance (overfitting), plotting Training error/CV error vs various models (maybe arrange in increasing order of numbers of features/orders) can be used.



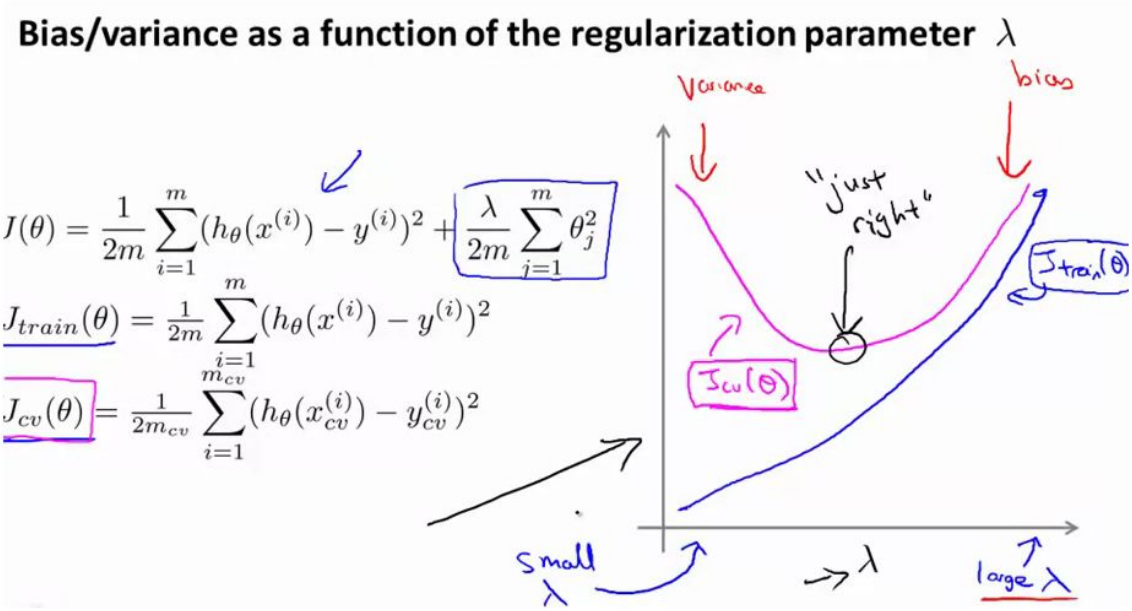
High Variance: when the CV error is much greater than Training error.
High Bias: when CV error is about the same as Training error.

Part X (5) - regularization (λ) effects on Bias/Variance:

Using different λ in the regularization term to minimize cost J(θ), will result in different Θ. This gives an opportunity to choose from different models corresponding to different Θ e.g. ($\Theta^{(1)}$, $\Theta^{(2)}$..$\Theta^{(n)}$).

Using model selection method to determines which Θ is the best by calculating the Training Error and CV error, but Omit the regularization term.

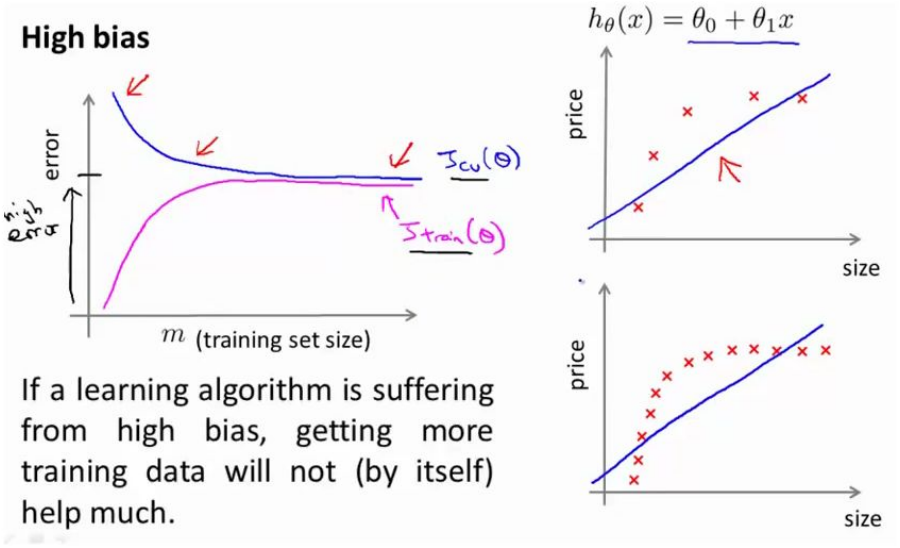
This is a graph of how varying λ affects the Training/CV error.



Part X (6) - Learning Curves:

When trying to fix a bad (overfit/underfit) model, plotting Training/CV errors vs m (training set size) can help give insight on whether gathering more training data will help.

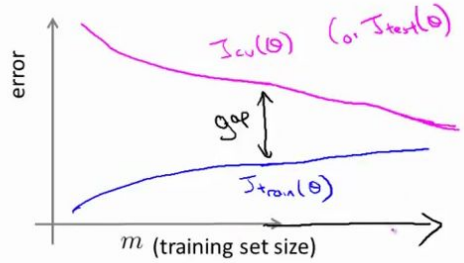
For a High bias problem (underfit), the CV error plateaus out quickly, indicates that getting larger m won't help, instead should try fitting more features(higher degree):



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

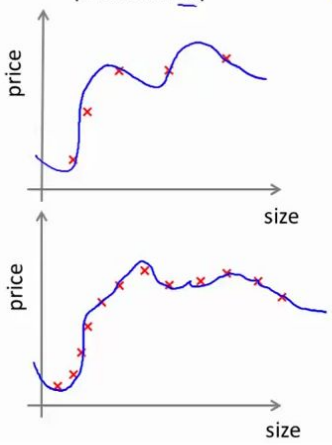
For a High Variance problem (overfit), the CV error continues to decrease and converge toward the "**Big Gap**" between the CV error & Training error, so gathering more data might help, if increasing λ (regularization) didn't help:

High variance



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

$h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_{100} x^{100}$
(and small λ)



Conclusion:

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

To build a ML algo effectively:

- Implement a very simple algo and get it to run.
- Test the algo over CV set.
- Apply Error Analysis (manually check what's the problem) to see how to improve the algo.

Part XI (1~3) - ML system design:

- Start with simple algo. To test it on Cross-validation data.
- Then plot learning curves to decide if more data or more features or something else might of useful.
- Then perform error analysis, where you manually examine the examples at which the algo made mistake on (e.g. type of spams with some specific features), to spot trends. etc.

Error analysis, Numerical evaluation:

- Single real number error metric: when a single number can tell the performance of current algorithm. This allows comparison between ideas.
- Error metric for skewed class: when distribution of different classes are skewed (one class have much less cases than the other). This type often gives high accuracy when the algorithm is actually bad. E.g. the one below gives higher accuracy when the algo always return y = 0 !!

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)

skewed

Only 0.50% of patients have cancer.

→ skewed classes.

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

0.5% error

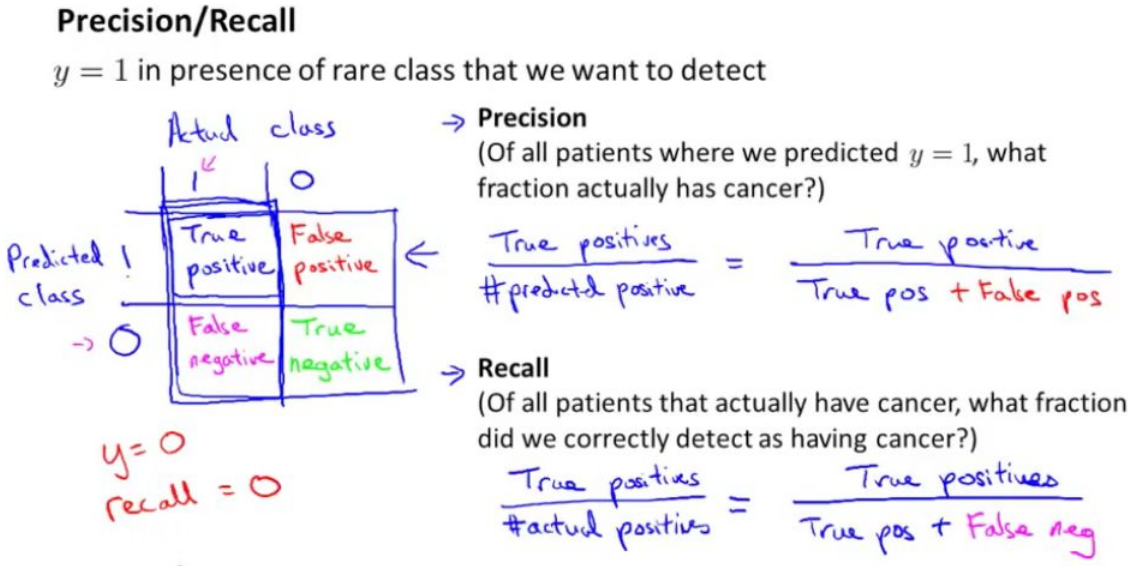
→ 99.2% accuracy (0.8% error)

→ 99.5% accuracy (0.5% error)

Even this work better

Therefore, a single number error metric no longer represent the quality of the algo.

So another Error metrics is required: Precision/Recall which can be found as follow:



A good algo will have both High precision and High recall, say when the algo always return 0, it will have zero Recall, thus a bad algo.

Part XI (4) - Trading Precision and Recall:

Logistic Regression: $0 \leq h_\theta(x) \leq 1$

Predict y = 1 $h_\theta(x) \geq 0.5$

Predict y = 0 $h_\theta(x) \leq 0.5$

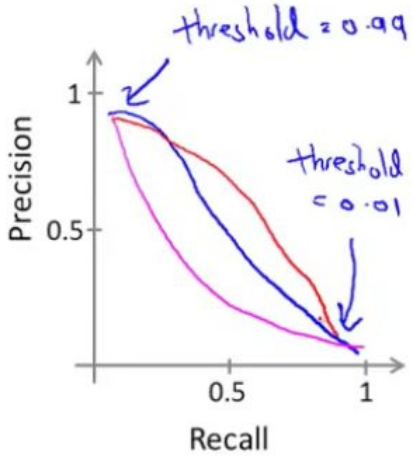
If we want to predict y = 1 only when very confident, we increase the precision and decrease the recall by:

Increasing the value 0.5, such as Predict y = 1 $h_\theta(x) \geq 0.8$ & Predict y = 0 $h_\theta(x) \leq 0.8$.

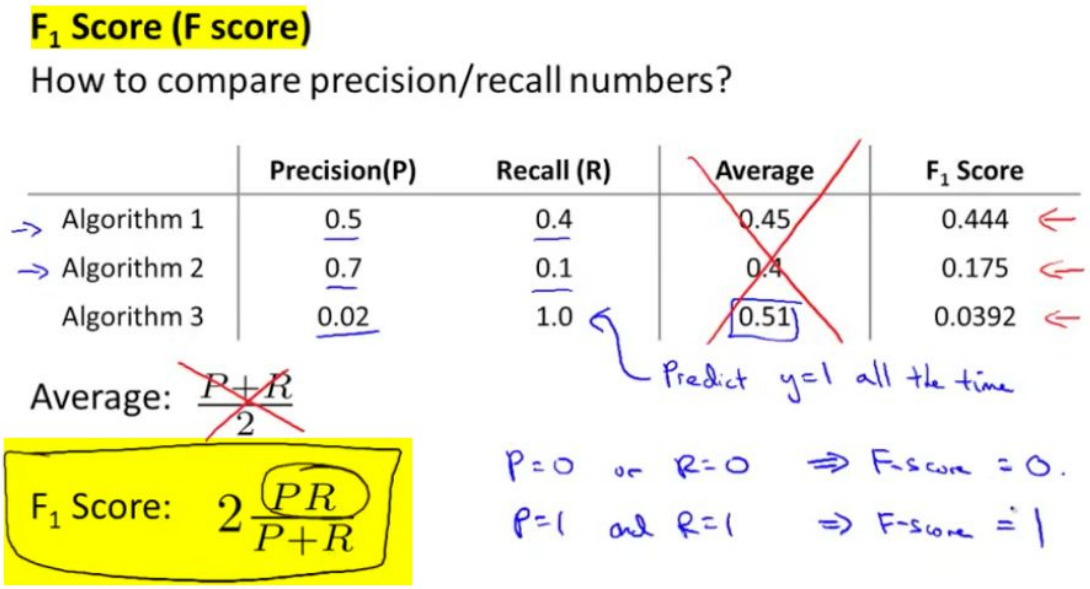
If we want to avoid false negatives instead, i.e. want to predict more y = 0, we increase recall and decrease precision by:

Decreasing the value 0.5, such as Predict y = 1 $h_\theta(x) \geq 0.3$ & Predict y = 0 $h_\theta(x) \leq 0.3$.

Or in general, predict y = 1 if $h_\theta(x) \geq threshold.$



Determining Good algo using F1 score:



Lastly, when deciding whether or not to gather more data for an algo, think about would that help if it was a human expert who is using the data to make prediction, if not, more data would NOT help the algo either!!!

==========

Week 7

XII (I) - Optimization Objective

Two definitions for simple expression :

$$cost_1(z) : \; - \log (1/1 + e^{-z})$$

$$cost_0(z) : \; - \log (1 - 1/1 + e^{-z})$$

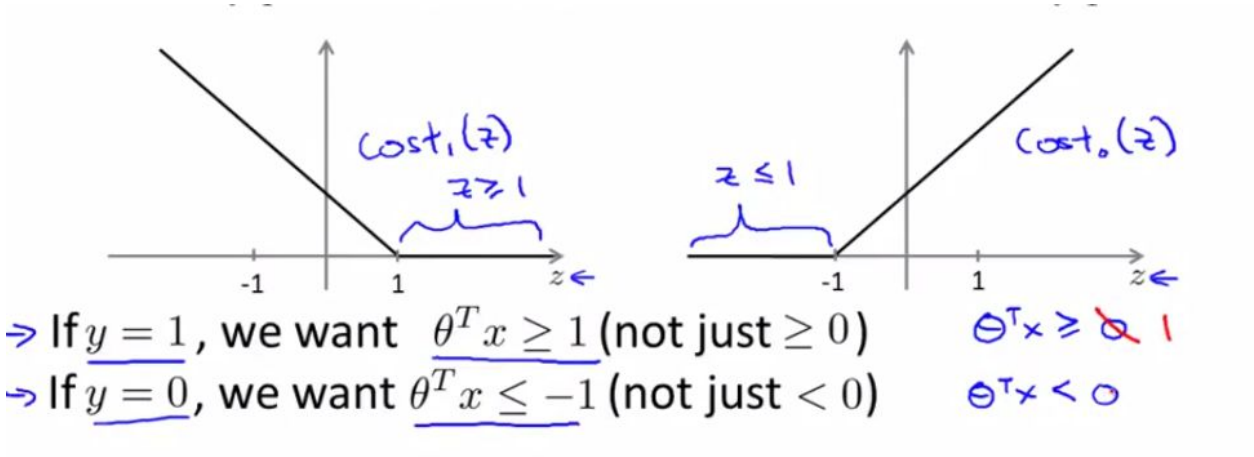
Optimization Function:

$$\min_{\theta} C \sum_{i=1}^{m} \left[y^{(i)} cost_1(\theta^T x^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{i=1}^{n} \theta_j^2$$

XII (II & III) - Large Margin Intuition

SVM optimization function ensures a larger margin in classification.

This is because of property of cost functions defined above.



Also the SVM optimization function ensures that the algorithm chooses parameter vector such that it maximizes difference between positive and negative examples.

Both these factors contribute to make SVM large margin classifier (much more compared to logistic regression)

XII (IV) - KERNELS I

Kernels: Make linear models work in nonlinear settings

- By mapping data to higher dimensions where it exhibits linear patterns
- Apply the linear model in the new input space
- Mapping ≡ changing the feature representation

We may use similarity with certain landmark points. This way we get as many features for each training examples as there are these landmarks. The Gaussian Kernel uses following expression for measuring similarity with the landmark points:

$$f_1 = \text{similarity}(x, l^{(1)}) = \exp\left(-\frac{\|x - l^{(1)}\|^2}{2\sigma^2}\right)$$

XII (V) - KERNELS II

We choose as many landmarks as there are number of training examples. So, for our implementation of SVM , n = m.

For each training examples , i, we choose :

f1 = sim($x^{(i)}$, l(1))
f2 = sim($x^{(i)}$, l(2))
f3 = sim($x^{(i)}$, l(3))
.....
.....
fm = sim($x^{(i)}$, l(m))

Hypothesis: Given x, compute features $f \in \mathbb{R}^{m+1}$

→ Predict "y=1" if $\theta^T f \geq 0$

Training:

$$\min_\theta C \sum_{i=1}^{m} y^{(i)} cost_1(\theta^T f^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^{m} \theta_j^2$$

Choosing parameters for SVM (Gaussian)

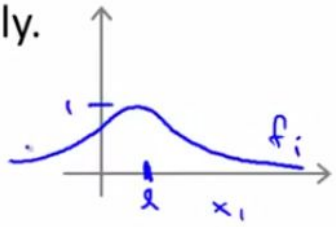
C parameter is a positive value that controls the penalty for misclassified training examples. A large C parameter tells the SVM to try to classify all the examples correctly. C plays a role similar to 1/λ , where λ is the regularization parameter that we were using previously for logistic regression

C ($= \frac{1}{\lambda}$). → Large C: Lower bias, high variance. (small λ)

→ Small C: Higher bias, low variance. (large λ)

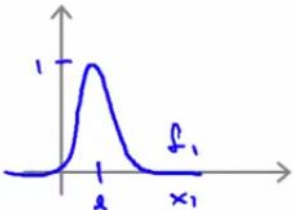
σ^2 Large σ^2: Features f_i vary more smoothly.

→ Higher bias, lower variance.



Small σ^2: Features f_i vary less smoothly.

Lower bias, higher variance.



XII (VI) - Using SVM

1. Use software libraries (liblinear, libsvm etc …)
2. Need to specify parameters like C or **σ**
3. Apply feature scaling if features are on very different scale, like it was done for logistic regression

There are many kernels

n = number of features ($x \in \mathbb{R}^{n+1}$), m = number of training examples

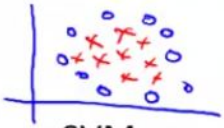
→ If n is large (relative to m): (e.g. $n \geq m$, $n = 10{,}000$, $m = 10 \cdots 1000$)

→ Use logistic regression, or SVM without a kernel ("linear kernel")

→ If n is small, m is intermediate: ($n = 1 - 1000$, $m = 10 - 10{,}000$) ←

→ Use SVM with Gaussian kernel

If n is small, m is large: ($n = 1 - 1000$, $m = 50{,}000+$)



→ Create/add more features, then use logistic regression or SVM without a kernel

→ Neural network likely to work well for most of these settings, but may be slower to train.

==========