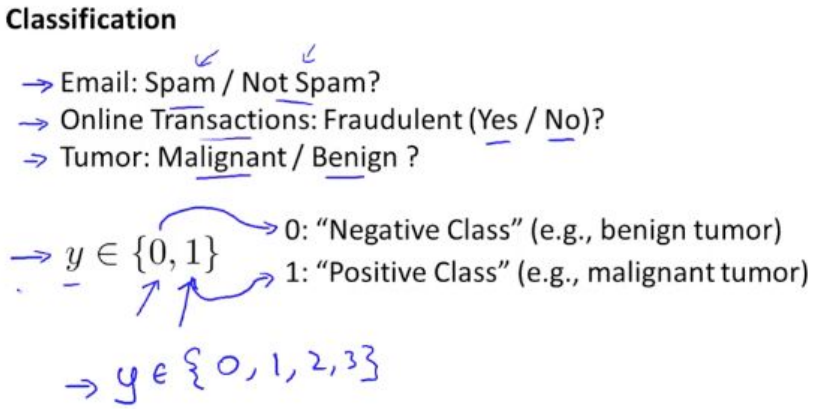
Andrew Ng's Machine Learning Course

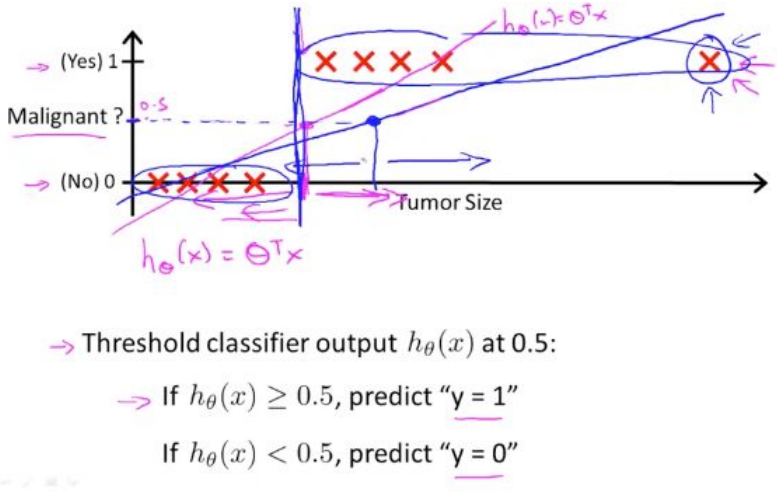
Week 3

Classification and Representation

- To refresh our memory, classification problems are problems where we have to decide if a value belongs to a 'class': whether an email is spam or not, whether a tumor is malignant or benign and so on. Initially we will focus in problems with only having 2 classes (binary classification) but later we will have multiple class classification problems.



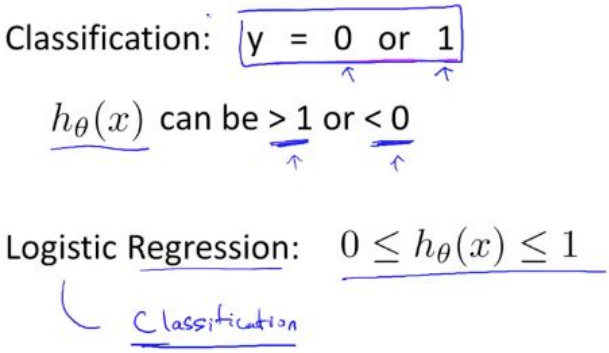
- Professor does not recommend using linear regression for classification problems and the example below shows why. Classification is not a linear function



In this example we have an initial training set that has 4 positive values and 4 negative values. We set the threshold classifier output at 0.5 (anything above 0.5 we predict 1,

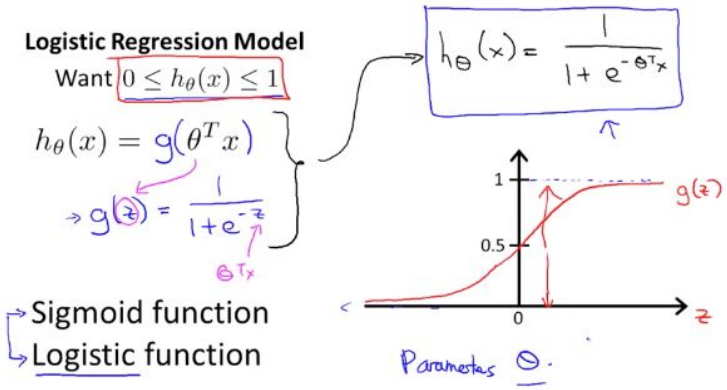
anything < 0.5 we predict 0) and the regular linear regression equation we get as a magenta line above does a reasonable job for this training set (all the positive values are to the right of the threshold and all the negative values are to the left of the threshold). However, if we add one more data point (a positive value way to the right) it causes our hypothesis to shift to the right (blue hypothesis) and that causes us to misclassify a couple of the training set examples as 0's when they should have been 1's

- Another problem in using linear regression is that even if we have a training set with just 1's and 0's linear regression can still have a hypothesis that produces values < 0 and/or > 1. This is strange and therefore another reason not to consider linear regression for classification problems
- We will use an algorithm called logistic regression to make sure our hypothesis output will have values between 0 and 1. By the way, the resulting classification, the y to each x, is know as the *label*



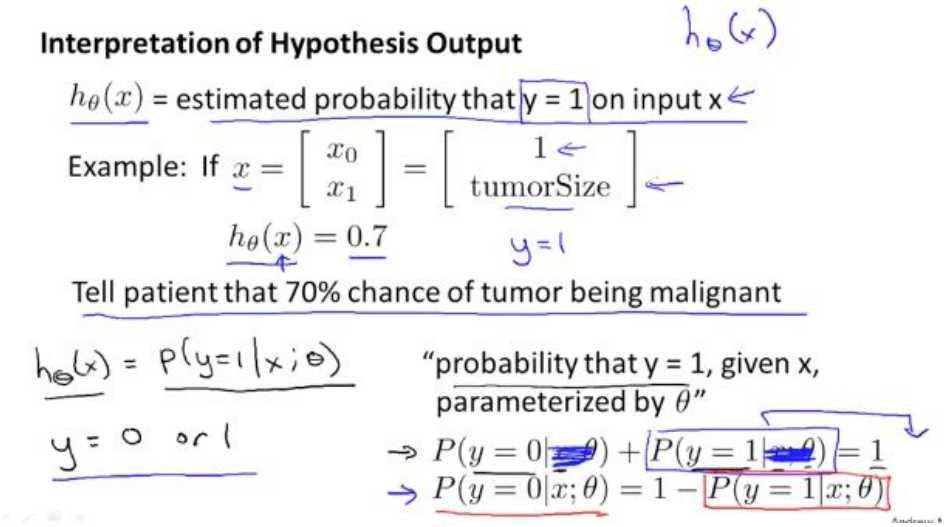
Hypothesis Representation

- Now we are going to see how to represent our hypothesis in the logistic regression model. In linear regression our hypothesis function was $h = \theta^T X$. For logistic regression we first introduce the logistic (or sigmoid function) which is $g(z) = \frac{1}{1+e^{-\theta^T X}}$. We then take $h_\theta(x) = g(\theta^T X)$ and plug in $\theta^T X$ in place of z and end up with the equation in the blue box



The logistic function plateaus at + infinity around 1 and plateaus at - infinity around 0 which is what we want because we want our function to produce values between 0 and 1

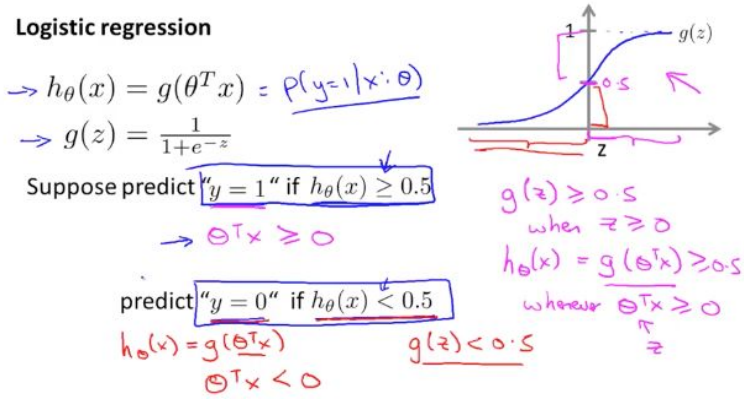
- How do we interpret the output of the $h_\theta(x)$ function? We interpret it as the estimated probability that y=1 given input x. In the example below we see the output of the function is 0.7 and we read that as saying this particular tumor has a 70% chance of being malignant



A more formal way of defining the function is $h_\theta(x)$ = P(y=1 | x ; θ) which we read as the probability of y =1 given x parametrized by θ. Finally, if we have the value for P(y=1 | x ; θ) we can also get the value for P(y=0 | x ; θ) because *P(y=0 | x ; θ) + P(y=1 | x ; θ) = 1;* solving for the probability of y =0 we get *P(y=0 | x ; θ) = 1 - P(y=1 | x ; θ)*

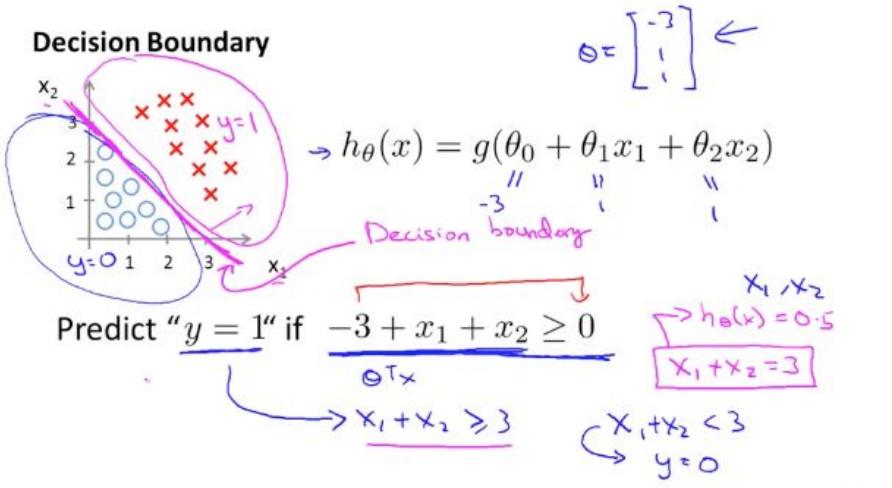
Decision Boundary

- Here we are trying to better understand when $h_\theta(x)$ is going to predict 1 and when it is going to predict 0. Looking at the graph of the sigmoid function we can see that g(z) is >= 0.5 whenever z >= 0 therefore we can say that $h_\theta(x)$ = g($\theta^T x$) >= 0.5 whenever $\theta^T x$ >= 0.

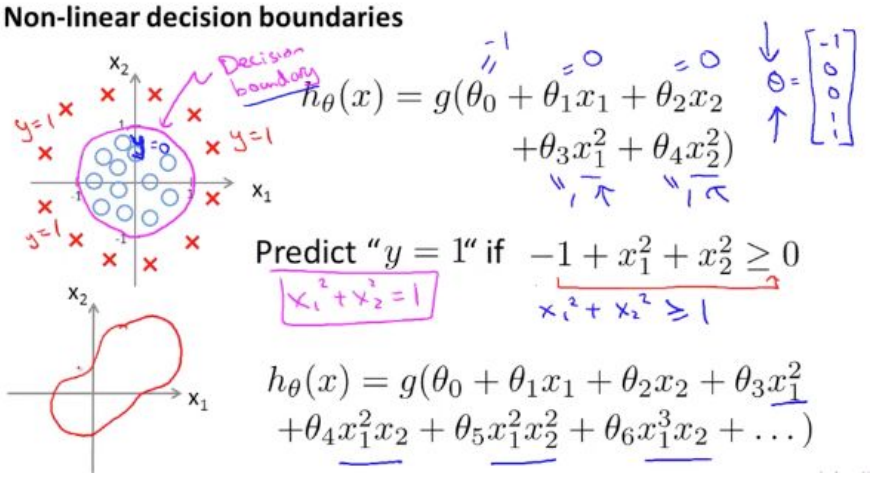


Similarly by looking at the sigmoid graph again we can say that g(z) is < 0.5 when z < 0. Applying the same logic as before we can say $h_\theta(x)$ = g $(\theta^T x)$ will predict 0 when $\theta^T x < 0$. *We can summarize all of this by saying we can predict y =1 when* $\theta^T x >= 0$ *and we can predict y = 0 when* $\theta^T x < 0$

- *The decision boundary is the line that separates the region where the hypothesis predicts y = 1 from the region where the hypothesis predicts y= 0*. This line is a property of the hypothesis and not a property of the training set. In the example below we have a hypothesis $h_\theta(x) = g(\theta_0 + \theta_1 x + \theta_2 x)$. Assuming $\theta_0 = -3$, $\theta_1 = 1$ and $\theta_2 = 1$, we can then say we can predict y=1 if $-3 + x_1 + x_2$ >= 0 which we can say is also $x_1 + x_2$ >= 3. This equation is the magenta line (the decision boundary)

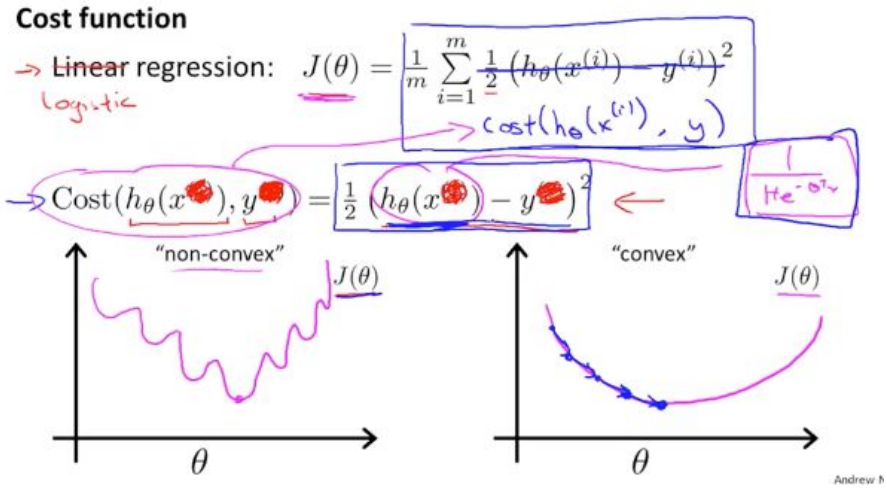


- We can get non linear decision boundaries by adding a few more terms to our hypothesis (just like we did with polynomial regression in linear regression). In the first example below we get a hypothesis that predicts y=1 if $-1 + x_1^2 + x_2^2$ >= 0 or $x_1^2 + x_2^2 \geq 1$. Plotting this equation gives us the magenta circle and that circle is the decision boundary. Adding even more terms will get us 'funny-shaped' decision boundaries like the one shown in the second example below

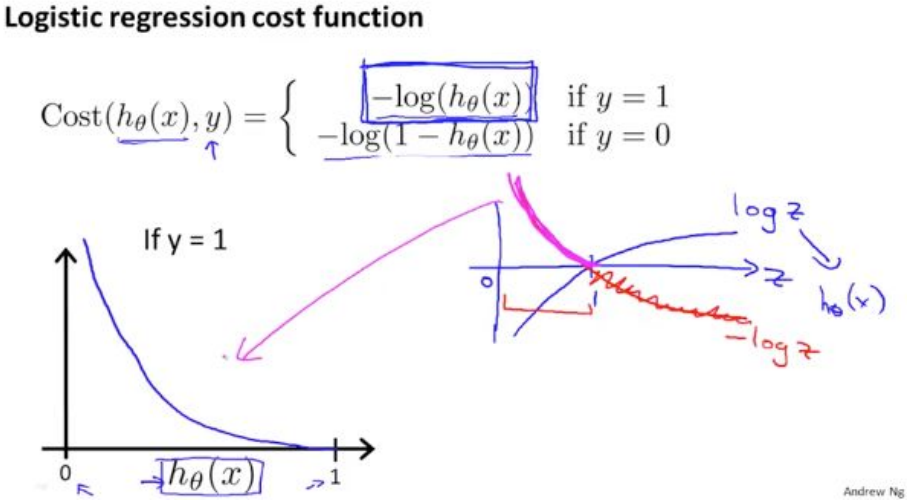


Cost Function

- A way to get an initial cost function for Logistic Regression is to slightly modify the linear regression function so that we end up with a cost function cost($h_\theta(x)$,y) = $\frac{1}{2}(h_\theta(x)-y)^2$. The problem is that when we plug in $\frac{1}{1+e^{-\theta^T x}}$ the resulting function is a non-convex function and with such a function it is not guaranteed that gradient descent will find the global minimum. What we want to do is find a convex cost function where gradient descent can converge



- Given that, the cost function that we are going to use for Logistic Regression is shown below. If y=1 the function is $-log(h_\theta(x))$ is graphed below. We can see that, as $h_\theta(x)$ goes to 1 the cost goes to 0 (the y value tends to zero which is what we want); in other words, if y=1 and $h_\theta(x)$= 1 then the cost = 0.

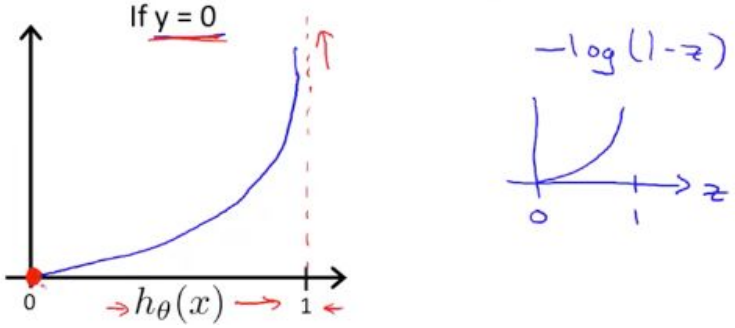


This function also captures the intuition that if y=1 but $h_\theta(x)$ = 0 we penalize the learning algorithm by a very large cost. This is represented by the cost (y axis) tending to infinity.

- Conversely, now we check the plot for the case where we are predicting y=0. In this case, we have the chart below. We can see how if y=0 and $h_\theta(x)$ = 0 then the cost is 0 (represented by the left side of the chart). We can also see how the cost -> infinity if we predicted a 1 but y = 0.

Logistic regression cost function

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$



Summarizing, Cost(hθ(x),y)=0 if hθ(x)=y
Cost(hθ(x),y)→∞ if y=0andhθ(x)→1
Cost(hθ(x),y)→∞ if y=1andhθ(x)→0

Simplified Cost Function & Gradient Descent

- We can simplify the way we write the logistic regression cost function in the following way: $Cost(h_\theta(x),\ y) = -y\ log(\ h_\theta(x)) - (1-y)\ log(1-h_\theta(x))$. We can prove that by plugging the 2 y values. If y=1 the 2nd value in our rewritten cost function is 0 and we are then left with $-y\ log(\ h_\theta(x))$. If y=0 the first value is multiplied by 0 and therefore it goes away and we are left with $-log(1-h_\theta(x))$

Logistic regression cost function

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

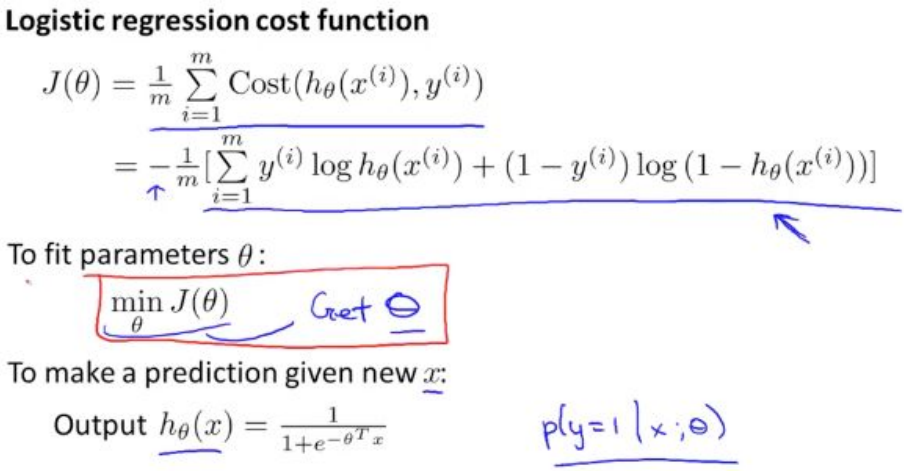
Note: $y = 0$ or 1 always

$\text{Cost}(h_\theta(x), y) = -y \log(h_\theta(x)) - (1-y) \log(1-h_\theta(x))$

If y=1: $\text{Cost}(h_\theta(x), y) = -\log h_\theta(x)$

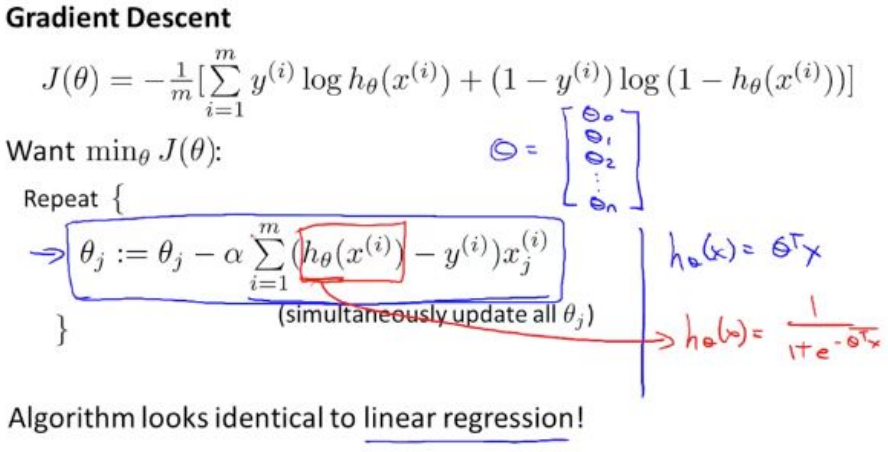
If y=0: $\text{Cost}(h_\theta(x), y) = -\log(1-h_\theta(x))$

- The fully rewritten logistic regression cost function is then
$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x), y) = -\frac{1}{m}[\sum_{i=1}^{m} y log(h_\theta(x)) + (1-y)\, log(1-h_\theta(x))]$



To make a prediction the output of $h_\theta(x)$ is read as the probability that y=1 given x instrumented by θ or, more formally p(y=1 | x; θ)

- A vectorized implementation of this cost function is
$J(\theta) = \frac{1}{m}(-y^T log(h) - (1-y)^T log(1-h))$
- We are going to minimize/optimize our cost function using gradient descent. If we want to minimize $J(\theta)$ our equation looks like this: $\theta_j := \theta_j - \alpha \sum_{i-1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^i$. This equation looks familiar as it is almost identical to the equation for linear regression. The difference is that $h_\theta(x)$ is different. $h_\theta(x)$ for linear regression was $\theta^T X$ but $h_\theta(x)$ for logistic regression is $\frac{1}{1+e^{-\theta^T X}}$



- A vectorized implementation of gradient descent for logistic regression is
$\theta = \theta - \frac{\alpha}{m} X^T (g(X\theta) - y))$
- Feature Scaling also applies to Logistic Regression

Advanced Optimization

- We already saw how we can use gradient descent to minimize/optimize our Logistic Regression cost function. However, gradient descent is not the only choice, we can also use other algorithms shown below. We do not implement these algorithms ourselves, we just use the libraries provided by octave, matlab or any other programming language

Optimization algorithm

Given θ, we have code that can compute

- $J(\theta)$
- $\frac{\partial}{\partial \theta_j} J(\theta)$ (for $j = 0, 1, \ldots, n$)

Optimization algorithms:

- Gradient descent
- Conjugate gradient
- BFGS
- L-BFGS

Advantages:

- No need to manually pick α
- Often faster than gradient descent.

Disadvantages:

- More complex

- An example of how to write an octave function to optimize $J(\theta)$ is shown below. It basically codes $J(\theta)$ and the partial derivatives for each θ parameter.

Example: $\min_{\theta} J(\theta)$

$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$ $\theta_1 = 5, \theta_2 = 5.$

$J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$

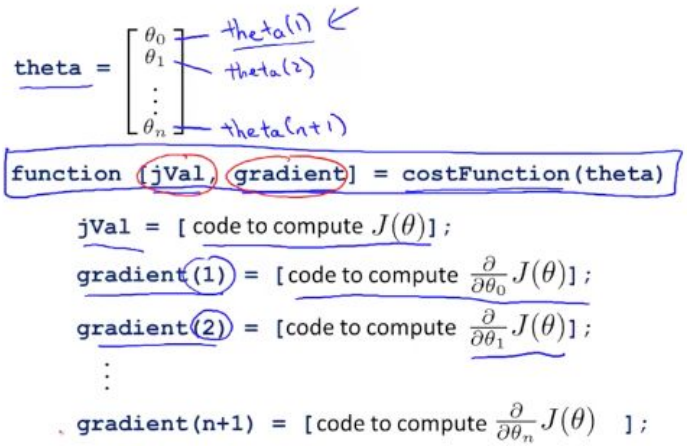
$\frac{\partial}{\partial \theta_1} J(\theta) = 2(\theta_1 - 5)$

$\frac{\partial}{\partial \theta_2} J(\theta) = 2(\theta_2 - 5)$

```
function [jVal, gradient]
            = costFunction(theta)
  jVal = (theta(1)-5)^2 + ...
         (theta(2)-5)^2;
  gradient = zeros(2,1);
  gradient(1) = 2*(theta(1)-5);
  gradient(2) = 2*(theta(2)-5);
```

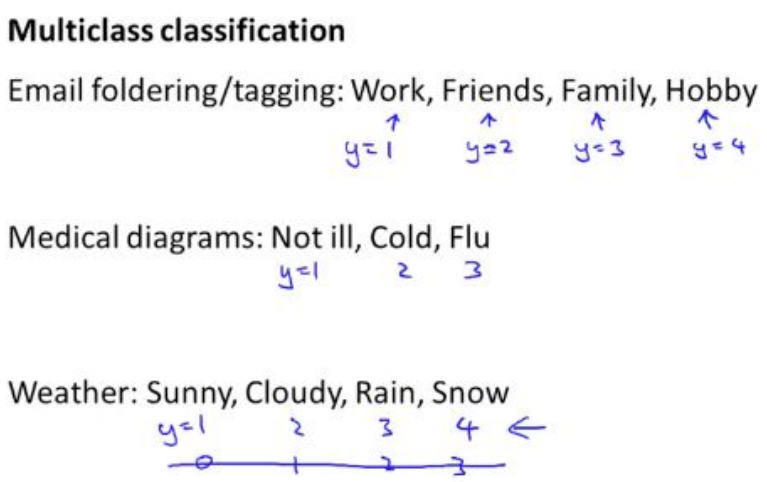
```
options = optimset('GradObj', 'on', 'MaxIter', '100');
initialTheta = zeros(2,1);
[optTheta, functionVal, exitFlag] ...
      = fminunc(@costFunction, initialTheta, options);
```

- Formalizing, what we need to do is write a cost function that returns the value of J and the value for each θ parameter

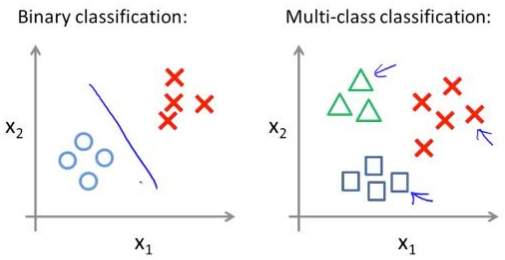


Multiclass Classification (one vs all)

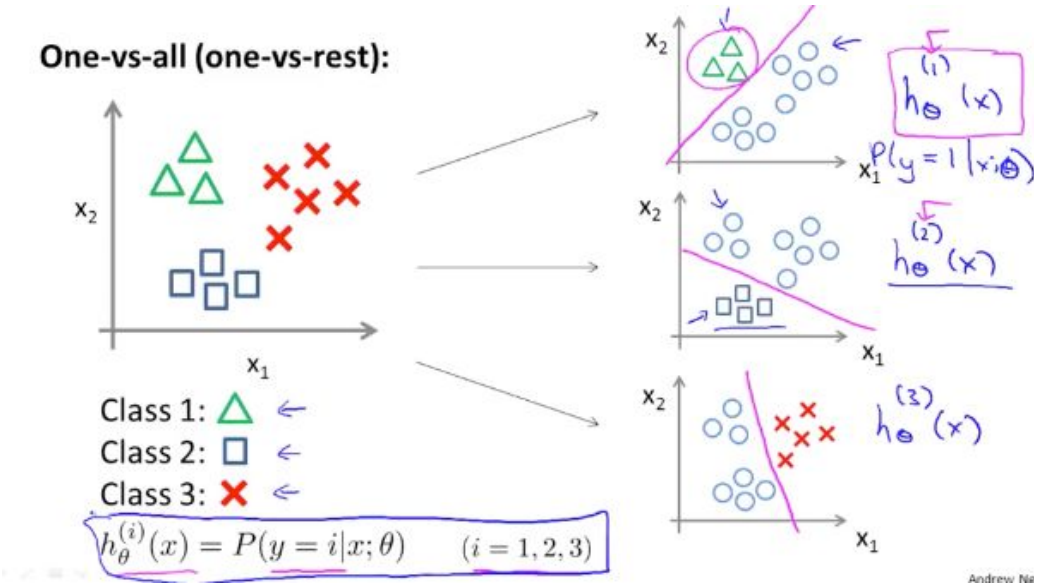
- Some examples of multi-classification problems are email foldering, medical diagnosis or weather



- An example of a training set with multiple classes is shown below. The example on the right uses 3 symbols to denote the 3 classes. We can still use decision boundary for these cases using the one-vs-all method



- In one-vs-all we take one class and make it the 'positive' class and all the other classes are negative classes. We do this process for every single class we have. The example shown below has 3 classes. The first example has class 1 (triangles) as the positive class and we are trying to predict what is the probability of y=1 when x is parametrized by θ and the other 2 classes are the negative class and we probably will get a decision boundary like the magenta line. We then repeat the process for all the other classes until we have all the decision boundaries and probabilities for each class.



- Summarizing, we are training a logistic regression classifier for each class i to predict the probability that y = 1. To make a prediction on an new input x we take all our classifiers and pick the one that maximizes the probability that y = i (the one that maximizes $h_\theta(x)$

One-vs-all

Train a logistic regression classifier $h_\theta^{(i)}(x)$ for each class i to predict the probability that $y = i$.

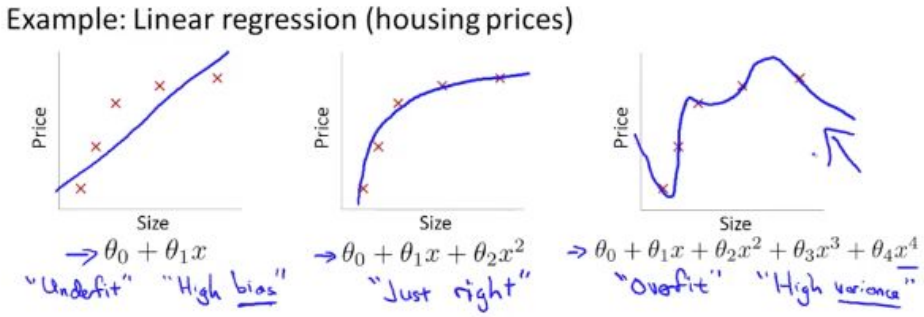
On a new input x, to make a prediction, pick the class i that maximizes

$$\max_i h_\theta^{(i)}(x)$$

The problem of Overfitting

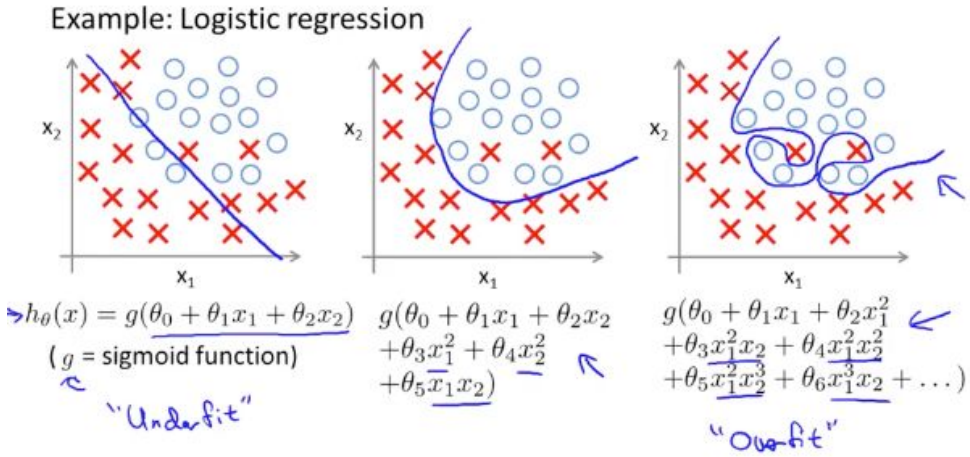
- Below we have 3 hypotheses for linear regression. The first hypothesis does kind a poor job because it's a linear equation and it assumes the price of the house will continue to increase as the size of the house increases. However, the training data shows the price

plateaus at a certain point making this hypothesis a poor fit or *underfit* or one that has *high bias*. The 2nd hypothesis has a quadratic equation that is just right for this data set. The final hypothesis with a 4th-order polynomial 'tries' too hard to fit the dataset and we end up with a clunky curve as shown below. This 3rd case is the *overfitting/high variance* case: initially the hypothesis may look like it fits the training set very well but it will fail to generalize for new examples



Overfitting: If we have too many features, the learned hypothesis may fit the training set very well ($J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \approx 0$), but fail to generalize to new examples (predict prices on new examples).

- We can have the same case of overfitting for Logistic Regression. Below is an example of that. In this case, the first hypothesis does a poor job and is therefore underfit. The second hypothesis does a better job and is probably the best fit. The 3rd hypothesis tries too hard and is therefore an overfit



- We can address overfitting with a couple of options mentioned below. The first one is to reduce the number of features by selecting some features to keep; the problem is that we lose some information. The second one is regularization where we keep all the features but reduce the magnitude/values of θ (sounds like feature scaling?). Regularization works well when we have a lot of useful features

Addressing overfitting:

Options:

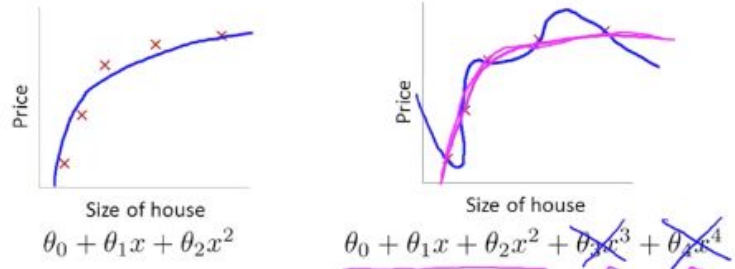
1. Reduce number of features.
   - — Manually select which features to keep.
   - — Model selection algorithm (later in course).
2. Regularization.
   - — Keep all the features, but reduce magnitude/values of parameters θ_j.
   - — Works well when we have a lot of features, each of which contributes a bit to predicting y.

Andre

Cost Function

- In the example below we have on the left a quadratic equation that is a good fit to the dataset and on the right a higher order polynomial that tries too hard to fit the dataset. To reverse this what we can do this is modify the cost function by penalizing the higher order parameters (θ_3 *and* θ_4) with a big amount (1,000 below). In this case, we will try to penalize θ_3 *and* θ_4 to optimize the cost function and therefore it's like θ_3 *and* θ_4 are 0 and we are back to a quadratic function which is a much better hypothesis

Intuition



Suppose we penalize and make θ_3, θ_4 really small.

$$\min_\theta \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + 1000\,\theta_3^2 + 1000\,\theta_4^2$$

$\theta_3 \approx 0$ $\theta_4 \approx 0$

- Regularization is the idea that having small values for the θ parameters will lead to a 'simpler' hypothesis and this hypothesis would be less prone to overfitting. In the previous example we knew θ_3 *and* θ_4 were the parameters to shrink. However, in many cases we don't know what parameters we need to minimize.

Regularization.

Small values for parameters $\theta_0, \theta_1, \ldots, \theta_n$
- "Simpler" hypothesis
- Less prone to overfitting

Housing:
- Features: $x_1, x_2, \ldots, x_{100}$
- Parameters: $\theta_0, \theta_1, \theta_2, \ldots, \theta_{100}$

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

What we do to account for this fact is to modify the cost function by adding a new regularization term that will shrink parameters $\theta_1 \ldots \theta_n$. In practice we do not penalize/minimize θ_0

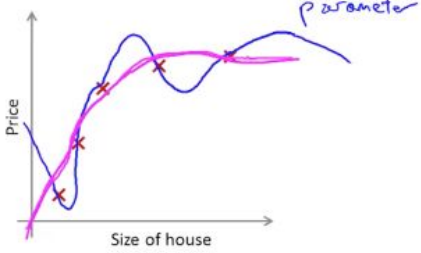
- Our regularized linear regression cost function is shown below and the new term (the regularization parameter λ, which tells us how inflated our parameters are) is helping us do a trade off between the two goals of the cost function: the first goal (captured by the first term) is that we want to do a good job of fitting the data set; the second goal (captured by the second term) is that we want to make sure the parameters are small because that leads to a simpler hypothesis and diminishes overfitting

Regularization.

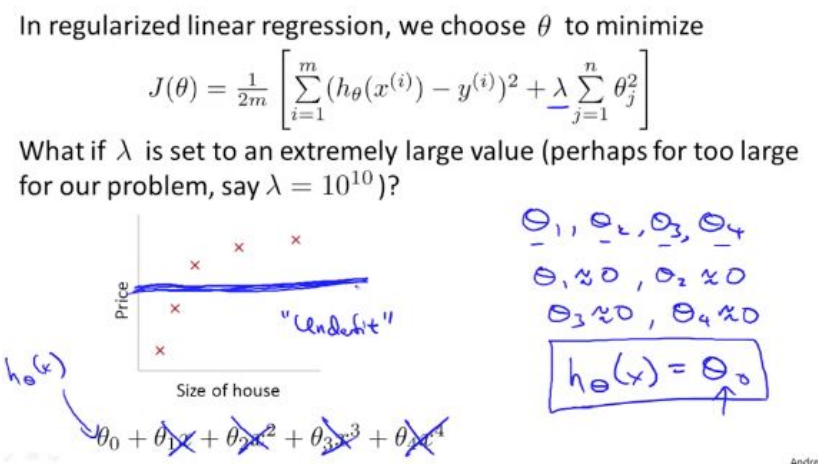
$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2\right]$$

regularization parameter

$$\min_\theta J(\theta)$$

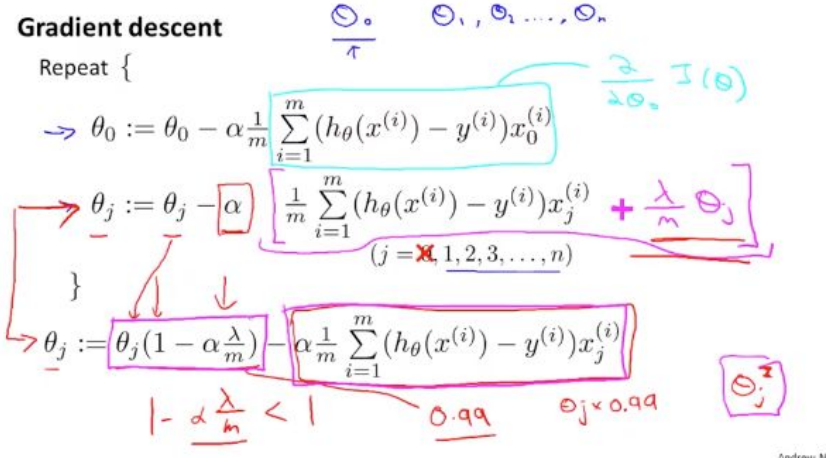


- We can have the opposite problem: if we penalize the parameters too much, as shown below, we will make all of them close to 0 and therefore our hypothesis would become $h_\theta(x) = \theta_0$ which would give us a bad fit to the data. This problem is called underfitting

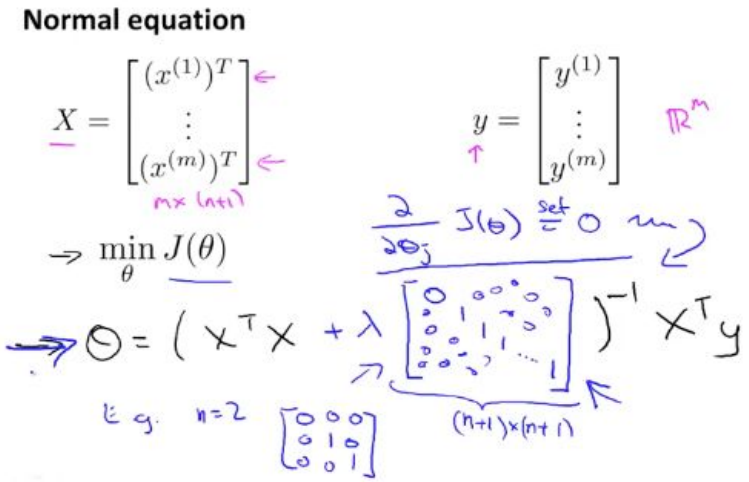


Regularized Linear Regression

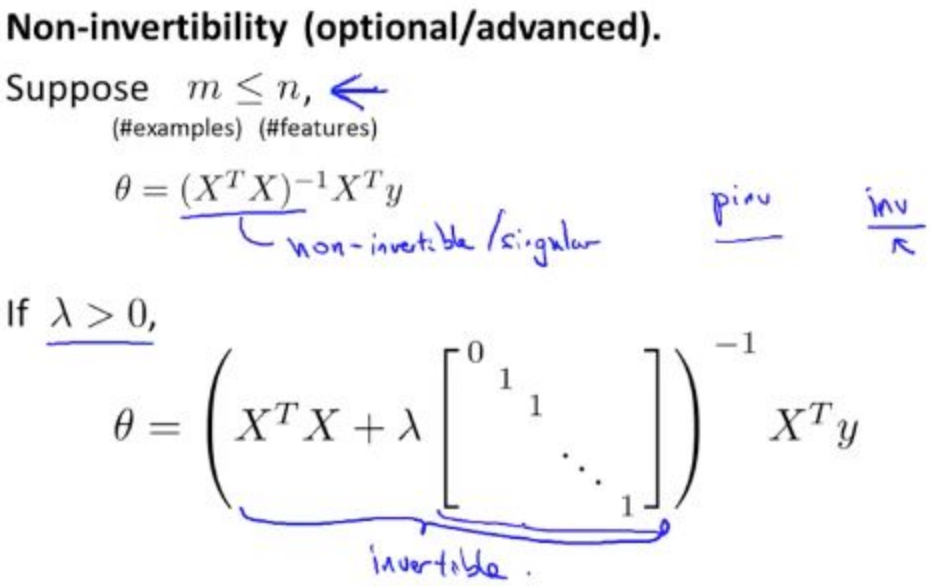
- In regularized linear regression we can still apply gradient descent. We take the regularized linear regression cost function and after separating the case for θ_0 and doing some math our θ_j equation is the term $\theta_j(1-\alpha\frac{\lambda}{m})$ - the original linear regression cost function. The term $1-\alpha\frac{\lambda}{m}$ is close to 0.99 (and will always be < 1) so its effect is to 'shrink' θ_j a little bit which is OK because what we want is to penalize the parameter θ a little bit.



- For normal equation we can also rewrite our cost function as shown below. Basically we add a term that has λ and a n+1 x n+1 matrix that is almost like the identity matrix with the exception the first term is always a 0.



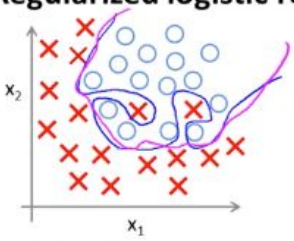
- A quick note above non-invertibility: regularization also takes care of the non invertibility problem we had in the past. Using math that we are not going to show here we can prove that, as long as $\lambda > 0$, the new normal equation term is invertible.



Regularized Logistic Regression

- To regularize logistic regression we take our original logistic regression cost function and add at the end the term $\frac{\lambda}{2m} \sum_{i=1}^{n} \theta_j^2$. This addition will have the effect of penalizing the θ parameters from being too large and we then are more likely to get a better decision boundary. Summarizing, regularization can also take care of overfitting for this case

Regularized logistic regression.



$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2 + \theta_5 x_1^2 x_2^3 + \dots)$$

Cost function:

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

$\theta_1, \theta_2, \dots, \theta_n$

- Gradient descent for regularized logistic regression can be modified similarly to the way we modified gradient descent for regularized linear regression. First we separate the case of θ_0 and then add the term $\frac{\lambda}{m}\theta_j$ to the original equation

Gradient descent

Repeat {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

$$\theta_j := \theta_j - \alpha \left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j\right] \quad (j = 1, 2, 3, \dots, n)$$

$\theta_1 \dots \theta_n$

}

$\frac{\partial}{\partial \theta_j} J(\theta)$

$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$

- For advanced optimization we still use the same function fminunc we used previously but the difference is in the code used to compute jVal (and most of the gradients (with the exception of gradient 1) now includes a regularization term as shown below

Advanced optimization

fminunc [@ costFunction] $\theta = \begin{bmatrix}\theta_0 \\ \theta_1 \\ \vdots \\ \theta_n\end{bmatrix}$ theta(1) theta(2) theta(n+1)

```
function [jVal, gradient] = costFunction(theta)
    jVal = [code to compute J(θ)];
        J(θ) = [-(1/m) Σ_{i=1}^{m} y^(i) log(h_θ(x^(i))) + (1 - y^(i)) log 1 - h_θ(x^(i))] + (λ/2m) Σ_{j=1}^{n} θ_j^2
    gradient(1) = [code to compute ∂/∂θ_0 J(θ)];
        (1/m) Σ_{i=1}^{m} (h_θ(x^(i)) - y^(i)) x_0^(i)
    gradient(2) = [code to compute ∂/∂θ_1 J(θ)];
        ((1/m) Σ_{i=1}^{m} (h_θ(x^(i)) - y^(i)) x_1^(i)) + (λ/m) θ_1
    gradient(3) = [code to compute ∂/∂θ_2 J(θ)];
        ((1/m) Σ_{i=1}^{m} (h_θ(x^(i)) - y^(i)) x_2^(i)) + (λ/m) θ_2
    :
    gradient(n+1) = [code to compute ∂/∂θ_n J(θ)];
```

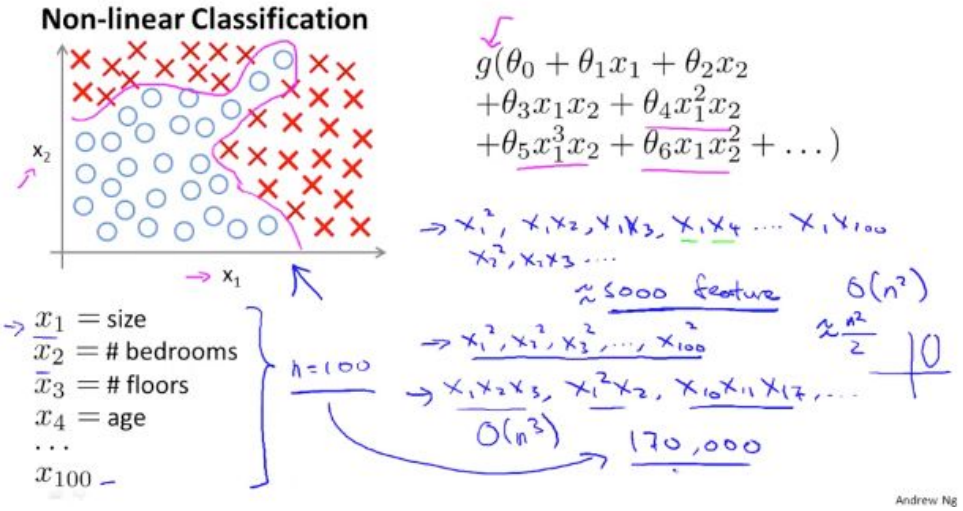
J(θ)

Week 4

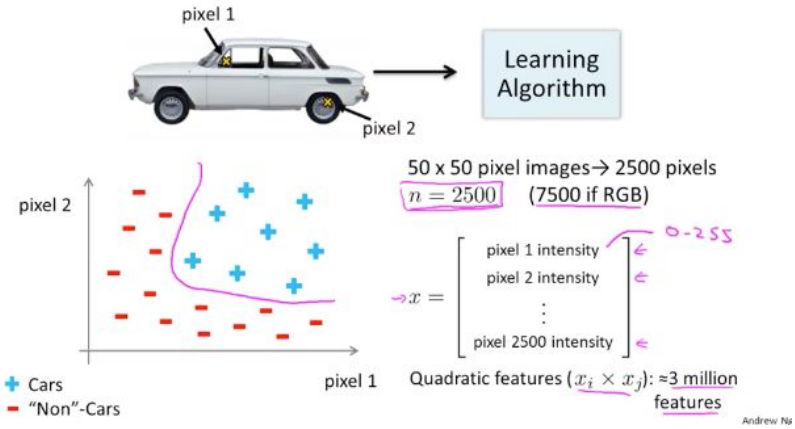
Neural Networks Motivations

Non Linear Hypothesis

- Neural Networks is considered the state of the art of machine learning and is yet another learning algorithm
- Why do we need another learning algorithm? Because we will have problems like the one shown below. In the dataset shown below we can try logistic regression and add a number of high order polynomials to try to find a hypothesis that fits the training set. However, we find out that if we have a large set of original features (say a 100) we end up creating features in the 1000's. Below we have a case of 5,000 and 170,000 features. This does not seem like a good way to build additional features for a non-linear classification problem when we have a large number of original features



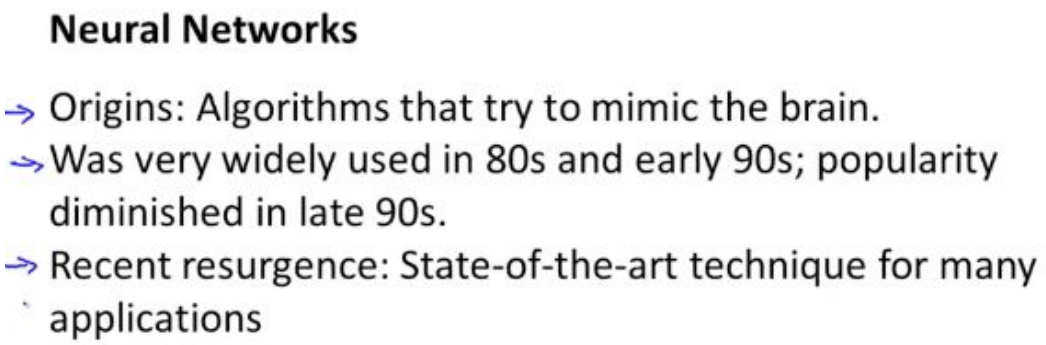
- To illustrate this point let's take a look at computer vision. When a computer is looking at a car it is really looking at a number of pixels that represent the car. We can start with taking 2 pixels (and their intensity) from the car as shown below. Plotting the cars as + and the non-cars as - we end up with a training set as shown in the graph. It is not a linear classification problem so we need a non-linear solution. If we take 50x50 pixel images we end up with 2,500 pixels of gray; if we do RGB we end up with 7,500 pixels. If we try to add all the quadratic features with for the 2,500 pixels we end up with 3 million (3 million!!) features. This is very expensive and therefore not feasible.



- Summarizing, simple logistic regression + the addition of the quadratic terms is not a good way to learn complex non linear hypothesis because you end with too many features. Neural networks is a better way even when we have many features (n is large)

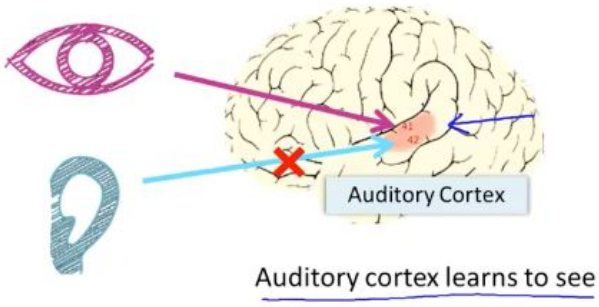
Neurons and the brain

- The origin of neural networks is with algorithms that try to mimic what the brain does. Widely used in the 80's and early 90's but diminished in popularity in the late 90's. The recent resurgence is because neural networks are expensive computationally speaking and is only recently that computers became fast enough to make neural networks feasible.



- We have a 'one learning algorithm' hypothesis on how the brain works. Scientists have been able to rewire an organism and send different kind of signals to the same brain tissue. In the example below we see how the auditory cortex, originally designed to 'hear' and process words, could learn how to see if signals from the eye are sent to it. A similar example is with the somatosensory cortex (designed to feel what we touch): if we send eye signals the somatosensory cortex will learn to see.

The "one learning algorithm" hypothesis

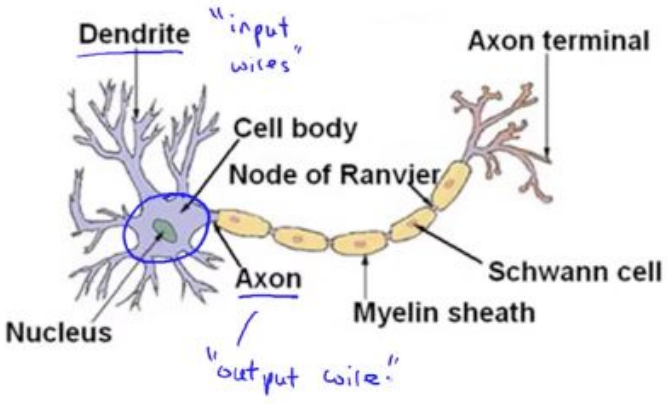


- We have other similar examples and the thinking says that we don't need to implement thousands of algorithms but rather just a approximation of how the brain learns to deal with whatever sensory information is sent to it and we are in business

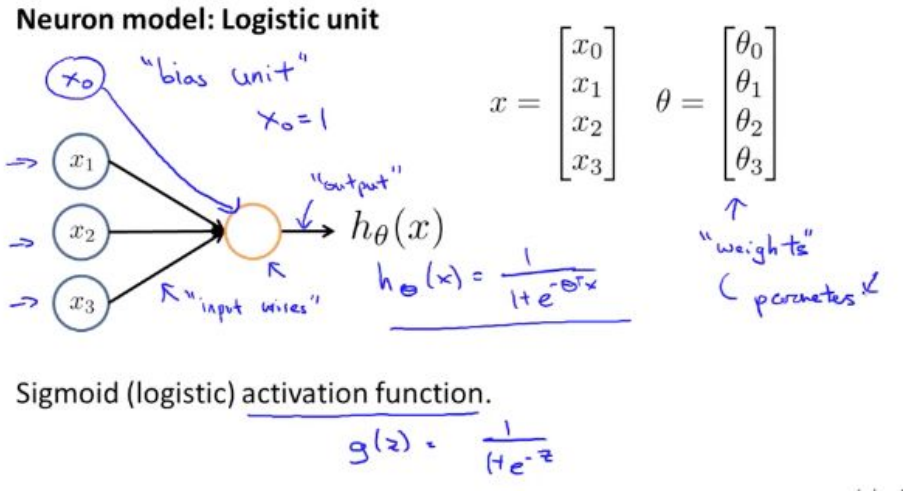
Model Representation I

- In simple terms a neuron in the brain is a computational unit that has input wires (Dendrite), does some computation on the inputs received and sends outputs using the output wires (Axons)

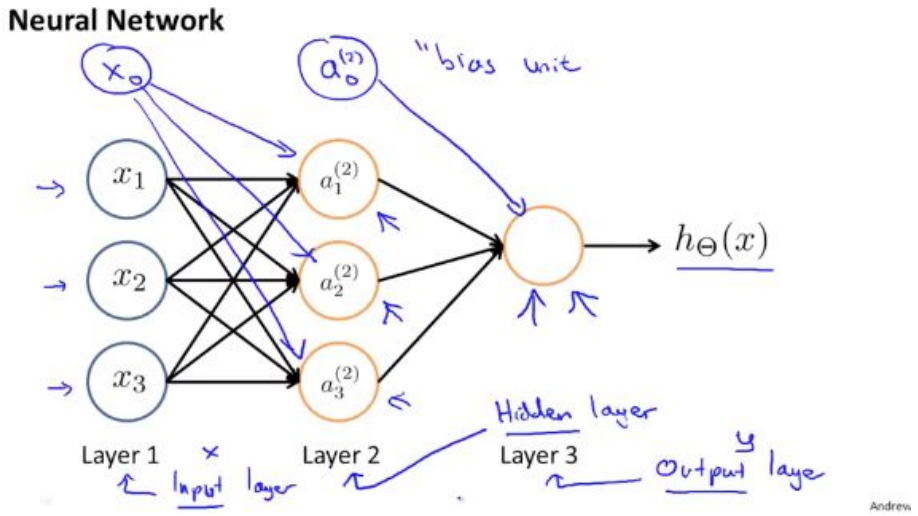
Neuron in the brain



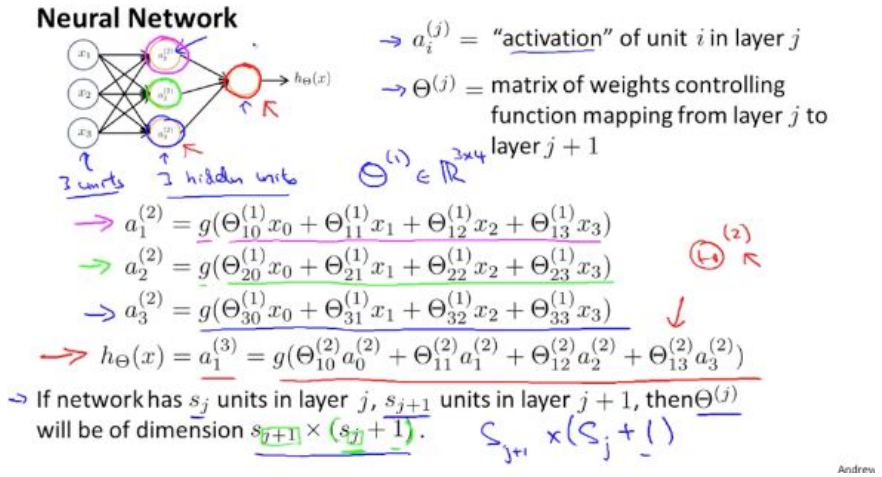
- We represent an 'artificial' neuron as a logistic unit that receives a set of inputs ($X_0, X_1, X_2 \ldots$) with X_0 *still equal to* 1 and the output is the hypothesis $h_\theta(x) = 1/(1+e^{-\theta^T X})$. In neural networks terminology we might hear the parameters referred to as weights and we also say we have a neuron with a sigmoid activation function



- We can represent a neural network as a group of neurons strung together as shown below. The first layer is called the input layer, the 3rd layer is called the output layer and the middle layer is called the hidden layer.



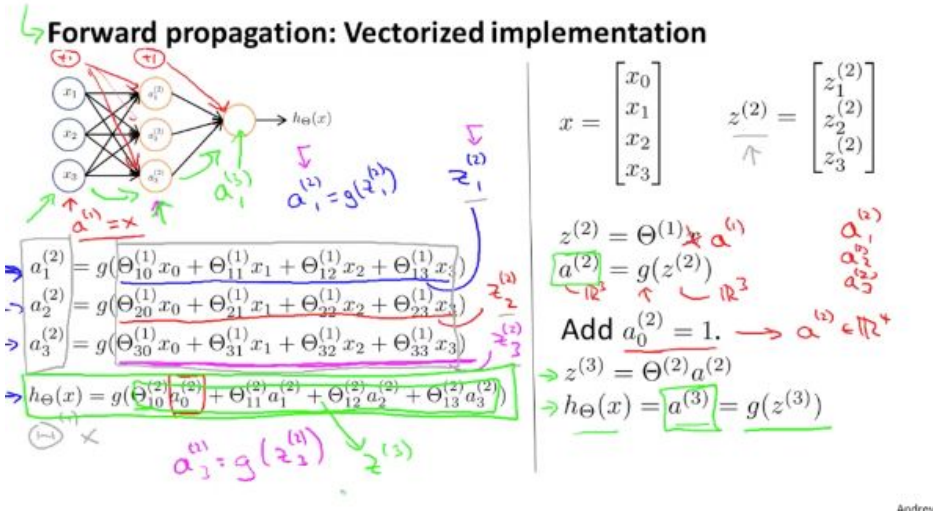
- The neural network has the following terminology
  - $a_i^{(j)}$ is the activation of unit i in layer j.
  - $\theta^{(j)}$ is the matrix of weights controlling function mapping from layer j to layer j+1
  - Additionally, in the example below we have 3 input units and 3 hidden layer units so $\theta^{(i)}$ is going to be a matrix of dimensions 3x4.
  - Generalizing, if a network has s_j units in layer j and s_{j+1} units in layer j+1 then the $\theta^{(j)}$ will be of dimensions $s_{j+1}\ x\ (s_j + 1)$.
  - An example is as follows: if layer 1 has 2 input nodes ($s_j = 2$) and layer 2 has 4 activation nodes ($s_{j+1} = 4$) then the dimensions of $\theta^{(j)}$ are going to be 4 x 3 because the dimensions are $s_{j+1}\ x\ (s_j + 1)$



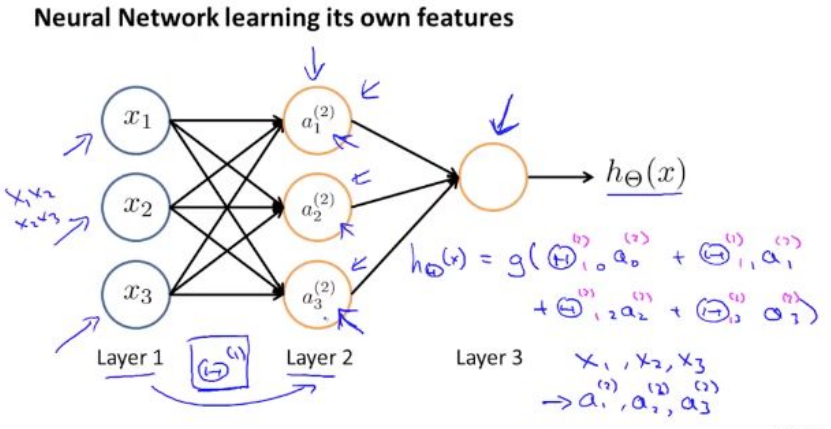
Model Representation II

- To get a vectorized implementation of the output hypothesis we use Forward Propagation. It's called Forward Propagation because we start from the first layer, the input layer, and we work our way to the output layer passing through the hidden layer(s).
- We start with layer 1 which is a vector of the x values. We then make the whole term inside g as z where $z = \theta^{(1)} * x$ but we can substitute x with an a to instead have $z^{(2)} = \theta^{(1)} * a^{(1)}$. It is $z^{(2)}$ because it correspond to the 2nd layer. Note that a is a vector and we add one more dimension by adding $a_0^{(2)} = 1$. Now we can define a as $a^{(2)} = g(z^{(2)})$. For the 3rd layer, the output layer, $z^{(3)} = \theta^{(2)}a^{(2)}$ and, therefore, our hypothesis $h_\theta(x) = a^{(3)} = g(z^{(3)})$ where g is the sigmoid function. Generalizing for z we can say that $z_{(j)} = \theta_{(j-1)} * a_{(j-1)}$. After adding a bias feature we can generalize z as $z^{(j+1)} = \theta^{(j)} * a^{(j)}$ Generalizing for a we can say $a^{(j)} = g(z^{(j)})$. Finally, the hypothesis is then

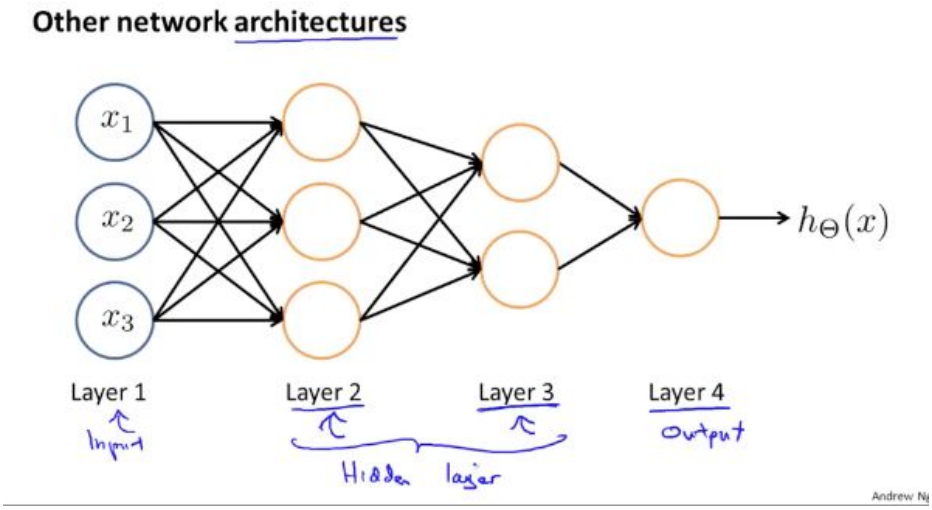
$h_\theta(x) = a^{(j+1)} = g(z^{(j+1)})$



- We can look at a neural network as learning its own features by doing the following: Suppose we cover the input layer and we are left with with just the hidden layer and the hypothesis layer. If we look at the resulting hypothesis equation we end up with something like $h_\theta(x) = g(\theta_0 a_0^{(2)} + \theta_1 a_1^{(2)} + \theta_2 a_2^{(2)} + \theta_3 a_3^{(2)})$ which is pretty much logistic regression (remember g is the sigmoid activation function) with the input being the hidden layer neurons, the a's, instead of the x's in the input layer. This is important because the algorithm is not constrained to the input x features. It has then the ability to learn its own features and, depending on the θ parameters selected, have at the end better hypothesis than if you were just constrained to the x parameters

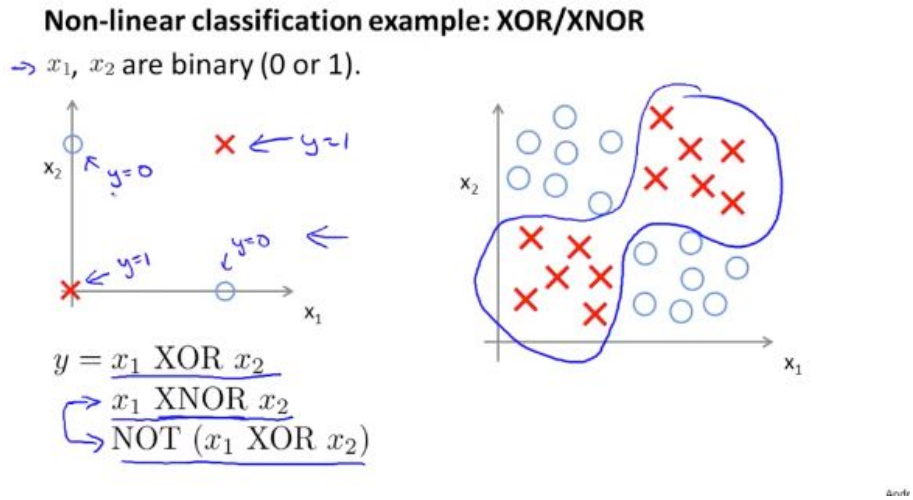


- There are other neural network architectures, an example is shown below. In this case we have 2 hidden layers and the usual input and output layer. What we can imagine here is how layer 2 computes some complex functions, layer 3 takes that input and computes even more complex functions and the hypothesis ends up being a better hypothesis at the end.



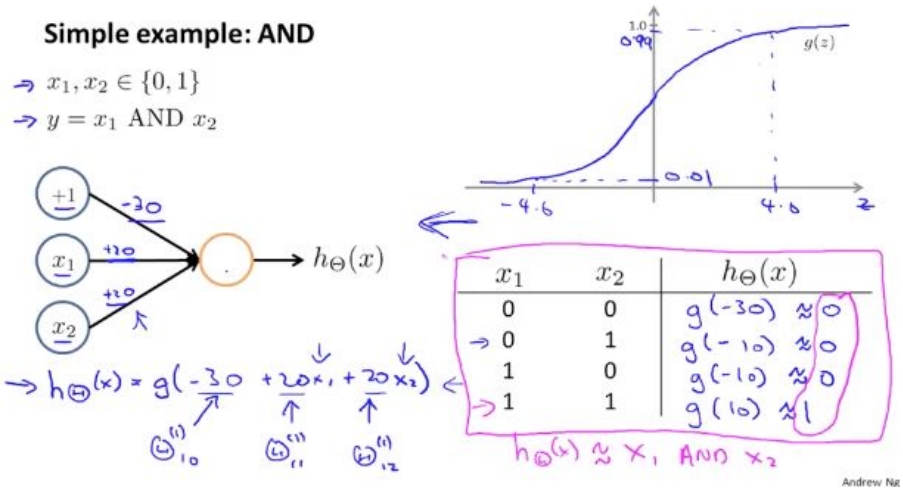
Examples and Intuitions I

- Here we are going to see how neural networks can solve a problem with non linear solutions. In this case we have a binary classification example where we want a decision boundary like the one on the right. However, we are going to use the example on the left and we are going to use the XNOR. In this case y=1 if x1 and x2 are both 0's or 1's and y=1 if any of the two are 1's.

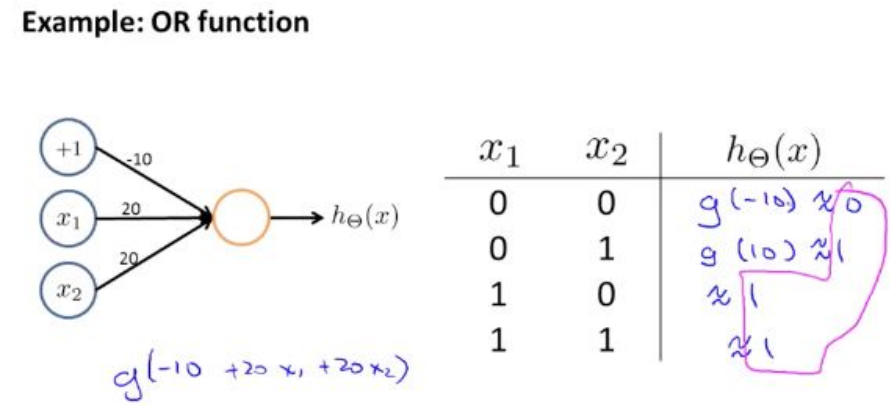


- In order to build up a neural network for XNOR we are going to start simple, with an example of how a neural network that is the AND function. We have two inputs: $x_1 and\, x_2$ and we add the bias unit. Then we give weights to each input: -30, 20 and 20 in this case. This will give us a hypothesis $h_\theta(x) = g(-30 + 20x_1 + 20x_2)$. We remember that g is the sigmoid function and know that at x = 4.0 g(z) is going to be 0.99 and at z = -4.6 g(z) is going to be 0.01. We then create our truth table like we always do for AND, OR, XOR and we end up with an $h_\theta(x)$ value for each of the 4 cases. For example, for the case when $x_1 = 0\ and\ x_2 = 0$ we end up with $h_\theta(x) = g(-30)$ which is way to the left of -4.6 so $h_\theta(x)$

is pretty much zero. We do the same for the other rows in our truth table and we end up with the truth table for AND. And this is how a single neuron neural network implements the logical AND

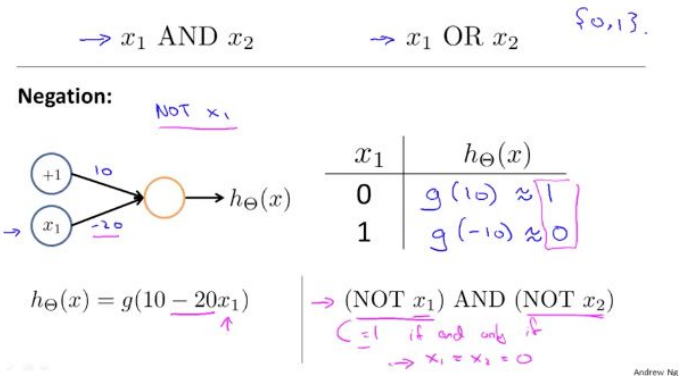


- An example of a neural network for OR is shown below

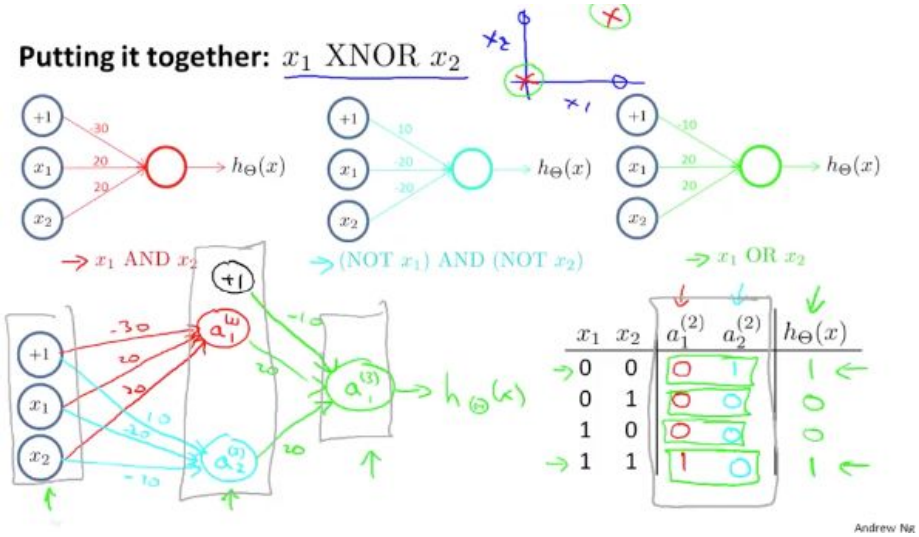


Examples and Intuitions I

- We continue with intuitions by demonstrating how a neural network can do a NOT. It is a neural network with just one input + the bias factor; we give 10 and -20 values to each and we end up with $h_\theta(x) = g(10 - 20x_1)$ and this hypothesis is our answer since when $x_1 = 0$ $h_\theta(x) = g(10) = 1$ and when $x_1 = 1$ $h_\theta(x) = g(-10) = 0$. Those two values give us our NOT truth table



- Now we combine our existing neural networks to create a combined neural network that computes the XNOR function as shown below. We start by taking the $x_1\ AND\ x_2$ neural network (including the weights) and ending it in a hidden unit. We then take the values of the $(NOT\ x_1)\ AND\ (NOT\ x_2)$ network and end the network in another hidden unit. The truth table for for those 2 hidden units is shown below. Finally, we add the bias factor and the $x_1\ OR\ x_2$ neural network and that output is our $h_\theta(x)$ which is the truth table for $x_1\ XNOR\ x_2$



- The general intuition is that we start with the simple calculations on the input layers but each subsequent layer can compute more complex calculations building on the work of the previous layers.
- Summarizing,
  - the θ^1 matrices for AND, NOR and OR are:
    - AND: $\theta^1 = [-30\ 20\ 20]$
    - NOR: $\theta^1 = [10\ -20\ -20]$
    - OR: $\theta^1 = [-10\ 20\ 20]$
  - The transition matrix between the first 1st and 2nd layer is: θ^1 =

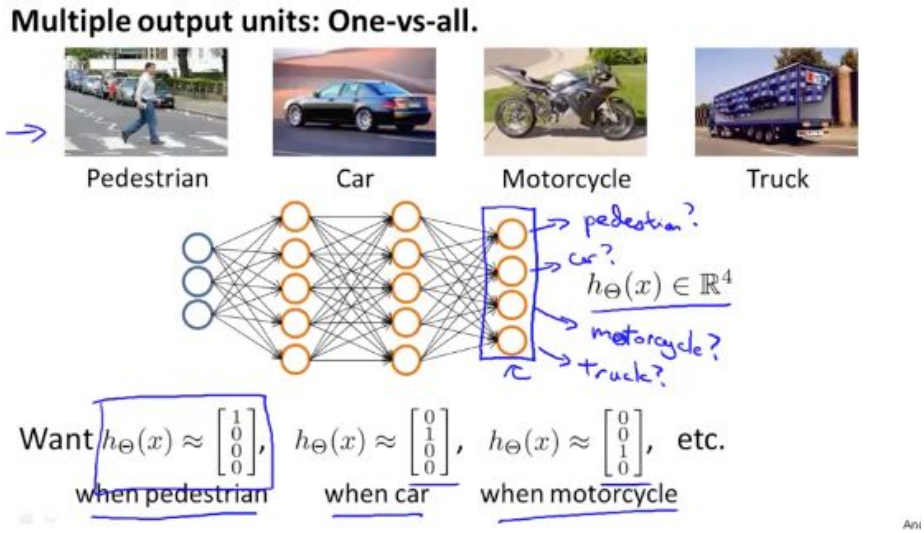
    -30 20 20

    10 -20 -20

- The transition matrix between the 2nd and 3rd layer is $\theta^1 = [-10\ 20\ 20]$
- The values for all the nodes are then:
  - $a^2 = g(\theta^1 * x)$
  - $a^3 = g(\theta^2 * a^2)$
  - $h_\theta(x) = a^3$

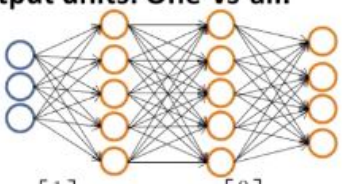
Multiclass Classification

- Multiclass classification with neural networks is an extension of the one-vs-all algorithm we saw in logistic regression. The example below shows a neural network that is trying to recognize if the image shown is a pedestrian, a car, a motorcycle or a truck. This neural network will have multiple outputs with the 1st output positive if the neural network thinks the image shown is a pedestrian, the 2nd output positive if the image is a car and so on. The output vector then is of size 4 as shown below. We essentially have 4 logistic regression classifiers, each one trying to classify an image in one of the 4 existing classes



- Rearranging thing a little bit we know our hypothesis is a size-4 vector with only one positive value and all the other values = 0. A training set is represented differently here because $y^{(i)} = [1\ 0\ 0\ 0]$ for a pedestrian, $y^{(i)} = [0\ 1\ 0\ 0]$ for a car, $y^{(i)} = [0\ 0\ 1\ 0]$ for a motorcycle and $y^{(i)} = [0\ 0\ 0\ 1]$ for a truck. Our training set is then $(x^{(i)}, y^{(i)})$ where x and y are vectors of size 4 and we are trying to get $h_\theta(x^{(i)}) = y^{(i)}$

Multiple output units: One-vs-all.



$h_\Theta(x) \in \mathbb{R}^4$

Want $h_\Theta(x) \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, etc.

when pedestrian when car when motorcycle

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

$y^{(i)}$ one of $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

$(x^{(i)}, y^{(i)})$

Previously $y \in \{1, 2, 3, 4\}$

$h_\Theta(x^{(i)}) \approx y^{(i)}$

$\in \mathbb{R}^4$

Andrew N