07: Regularization

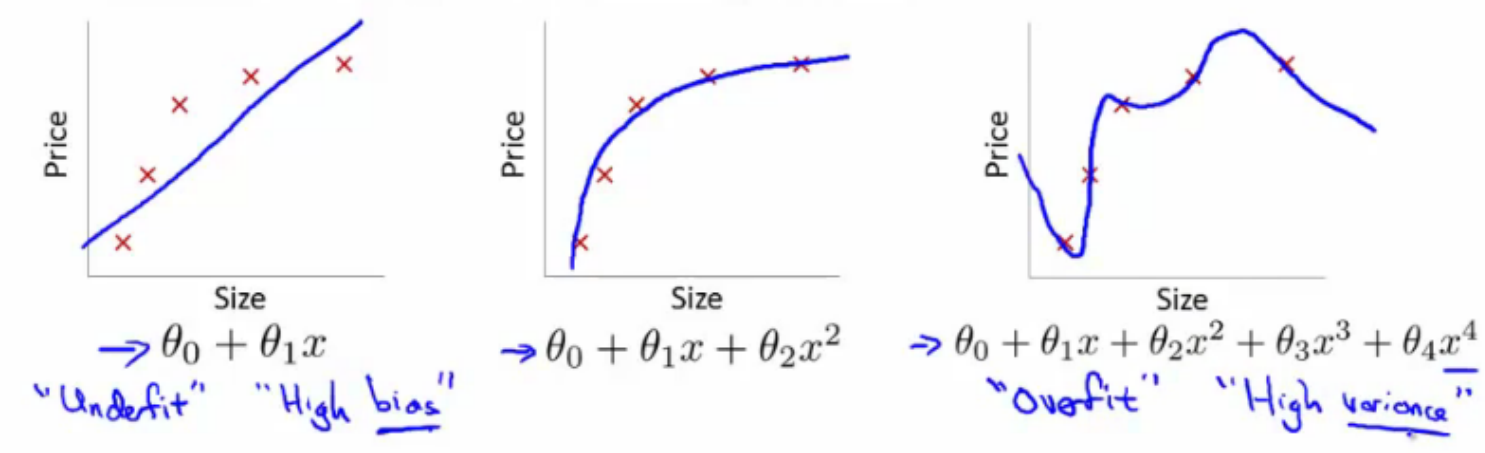
Previous Next Index

The problem of overfitting

- So far we've seen a few algorithms - work well for many applications, but can suffer from the problem of overfitting
- What is overfitting?
- What is regularization and how does it help

Overfitting with linear regression

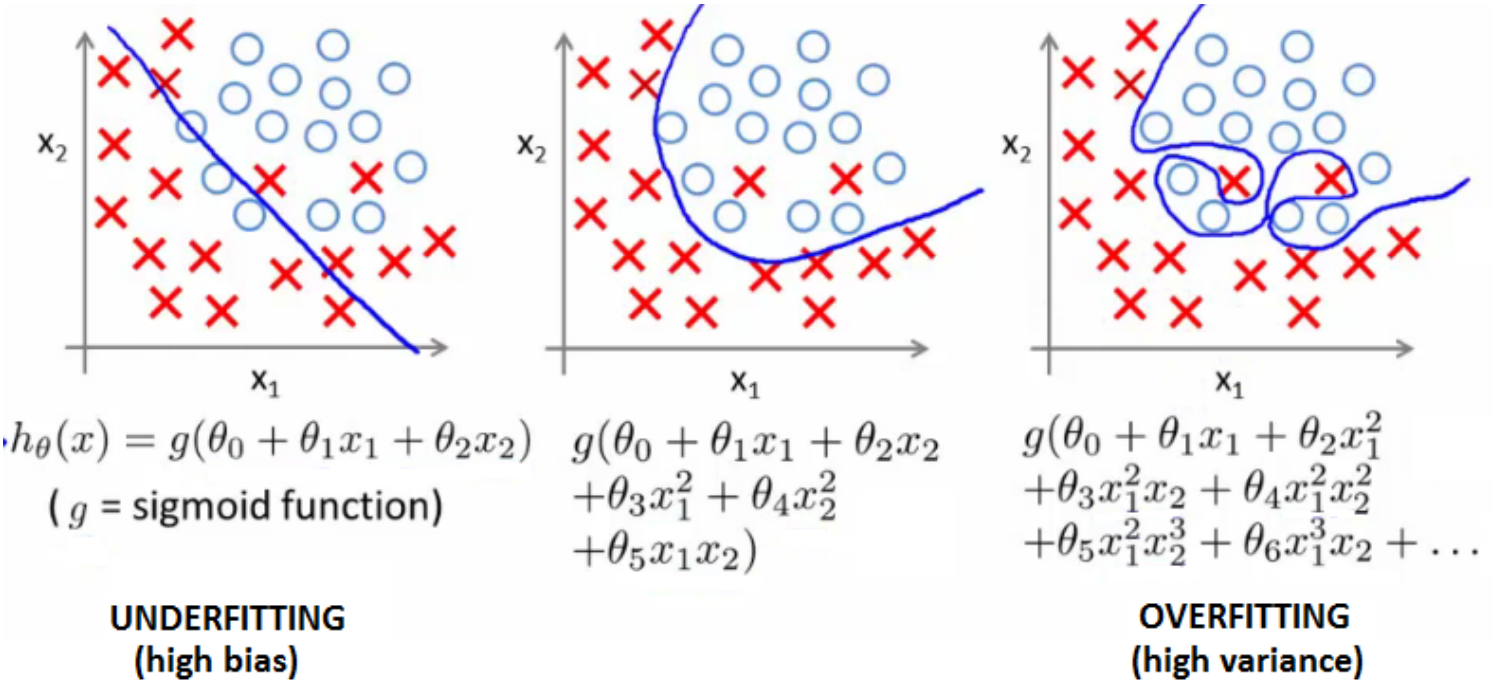
- Using our house pricing example again
  - Fit a linear function to the data - not a great model
    - This is **underfitting** - also known as **high bias**
    - Bias is a historic/technical one - if we're fitting a straight line to the data we have a strong preconception that there should be a linear fit
      - In this case, this is not correct, but a straight line can't help being straight!
  - Fit a quadratic function
    - Works well
  - Fit a 4th order polynomial
    - Now curve fit's through all five examples
      - Seems to do a good job fitting the training set
      - But, despite fitting the data we've provided very well, this is actually not such a good model
    - This is **overfitting** - also known as **high variance**
  - Algorithm has high variance
    - High variance - if fitting high order polynomial then the hypothesis can basically fit any data
    - Space of hypothesis is too large



- To recap, if we have too many features then the learned hypothesis may give a cost function of exactly zero
  - But this tries too hard to fit the training set
  - Fails to provide a *general* solution - **unable to generalize** (apply to new examples)

Overfitting with logistic regression

- Same thing can happen to logistic regression
  - Sigmoidal function is an underfit
  - But a high order polynomial gives and overfitting (high variance hypothesis)



Addressing overfitting

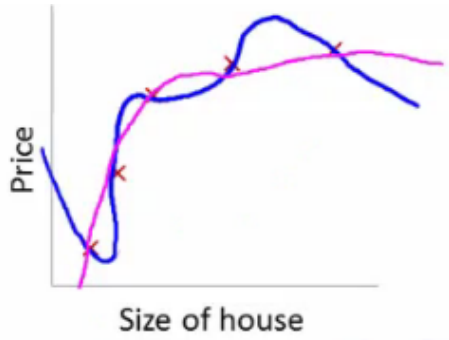
- Later we'll look at identifying when overfitting and underfitting is occurring
- Earlier we just plotted a higher order function - saw that it looks "too curvy"
  - Plotting hypothesis is one way to decide, but doesn't always work
  - Often have lots of a features - here it's not just a case of selecting a degree polynomial, but also harder to plot the data and visualize to decide what features to keep and which to drop
  - If you have lots of features and little data - overfitting can be a problem
- How do we deal with this?
  - 1) **Reduce number of features**
    - Manually select which features to keep
    - Model selection algorithms are discussed later (good for reducing number of features)
    - But, in reducing the number of features we lose some information
      - Ideally select those features which minimize data loss, but even so, some info is lost
  - 2) **Regularization**
    - Keep all features, but reduce magnitude of parameters θ
    - Works well when we have a lot of features, each of which contributes a bit to predicting y

Cost function optimization for regularization

- Penalize and make some of the θ parameters really small
  - e.g. here θ_3 and θ_4

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

- The addition in blue is a modification of our cost function to help penalize θ_3 and θ_4
  - So here we end up with θ_3 and θ_4 being close to zero (because the constants are massive)
  - So we're basically left with a quadratic function



$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

- In this example, we penalized two of the parameter values
  - More generally, regularization is as follows

- Regularization
  - Small values for parameters corresponds to a simpler hypothesis (you effectively get rid of some of the terms)
  - A simpler hypothesis is less prone to overfitting
- Another example
  - Have 100 features x_1, x_2, ..., x_{100}
  - Unlike the polynomial example, we don't know what are the high order terms
    - How do we pick the ones to pick to shrink?
  - With regularization, take cost function and modify it to shrink all the parameters
    - Add a term at the end
      - This regularization term shrinks every parameter
      - By convention you don't penalize θ_0 - minimization is from θ_1 onwards

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

$\theta_1, \theta_2, \theta_3, \ldots, \theta_{100}$

- In practice, if you include θ_0 has little impact
- λ is the **regularization parameter**
  - Controls a trade off between our two goals
    - 1) Want to fit the training set well
    - 2) Want to keep parameters small
- With our example, using the **regularized objective** (i.e. the cost function with the regularization term) you get a much smoother curve which fits the data and gives a much

better hypothesis
    - If **λ** is very large we end up penalizing ALL the parameters (θ_1, θ_2 etc.) so all the parameters end up being close to zero
        - If this happens, it's like we got rid of all the terms in the hypothesis
            - This results here is then underfitting
        - So this hypothesis is too biased because of the absence of any parameters (effectively)
- So, **λ** should be chosen carefully - not too big...
    - We look at some automatic ways to select **λ** later in the course

Regularized linear regression

- Previously, we looked at two algorithms for linear regression
    - Gradient descent
    - Normal equation
- Our linear regression with regularization is shown below

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

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

- Previously, gradient descent would repeatedly update the parameters θ_j, where j = 0,1,2...n simultaneously
    - Shown below

Repeat $\{$

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

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

$$(j = 0, 1, 2, 3, \ldots, n)\ \}$$

- We've got the θ_0 update here shown explicitly
    - This is because for regularization we don't penalize θ_0 so treat it slightly differently
- How do we regularize these two rules?
    - Take the term and add $\lambda/m * \theta_j$
        - Sum for every θ (i.e. j = 0 to n)
    - This gives regularization for gradient descent
- We can show using calculus that the equation given below is the partial derivative of the regularized J(θ)

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

$$(j = \cancel{0}, 1, 2, 3, \ldots, n)$$

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

- The update for θ_j
  - θ_j gets updated to
    - θ_j - α * [a big term which also depends on θ_j]
- So if you group the θ_j terms together

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

- The term

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

  - Is going to be a number less than 1 usually
  - Usually learning rate is small and m is large
    - So this typically evaluates to (1 - a small number)
    - So the term is often around 0.99 to 0.95
- This in effect means θ_j gets multiplied by 0.99
  - Means the squared norm of θ_j a little smaller
  - The second term is exactly the same as the original gradient descent

Regularization with the normal equation

- Normal equation is the other linear regression model
  - Minimize the J(θ) using the normal equation
  - To use regularization we add a term (+ λ [n+1 x n+1]) to the equation
    - [n+1 x n+1] is the n+1 identity matrix

$$\theta = \left(X^T X + \lambda \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & 1 & \\ & & & \ddots \\ & & & & 1 \end{bmatrix}_{(n+1)\times(n+1)} \right)^{-1} X^T y$$

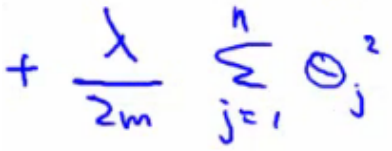
e.g. if n = 2 $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Regularization for logistic regression

- We saw earlier that logistic regression can be prone to overfitting with lots of features
- Logistic regression cost function is as follows;

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

- To modify it we have to add an extra term



- This has the effect of penalizing the parameters θ_1, θ_2 up to θ_n
  - Means, like with linear regression, we can get what appears to be a better fitting lower order hypothesis
- How do we implement this?
  - Original logistic regression with gradient descent function was as follows

$$\theta_j := \theta_j - \alpha \quad \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$
$$(j = 0, 1, 2, 3, \ldots, n)$$

- Again, to modify the algorithm we simply need to modify the update rule for θ_1, onwards
  - Looks cosmetically the same as linear regression, except obviously the hypothesis is very different

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

Advanced optimization of regularized linear regression

- As before, define a costFunction which takes a θ parameter and gives jVal and gradient back

```
function [jVal, gradient] = costFunction(theta)
    jVal = [ code to compute J(θ) ];

    gradient(1) = [ code to compute ∂/∂θ0 J(θ) ];

    gradient(2) = [ code to compute ∂/∂θ1 J(θ) ];

    gradient(3) = [ code to compute ∂/∂θ2 J(θ) ];
      ⋮
    gradient(n+1) = [ code to compute ∂/∂θn J(θ) ];
```

- use `fminunc`
  - Pass it an `@costfunction` argument
  - Minimizes in an optimized manner using the cost function
- `jVal`
  - Need code to compute J(θ)
    - Need to include regularization term
- Gradient
  - Needs to be the partial derivative of J(θ) with respect to θ_i
  - Adding the appropriate term here is also necessary

```
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)];
```

- Ensure summation doesn't extend to to the lambda term!
  - It doesn't, but, you know, don't be daft!

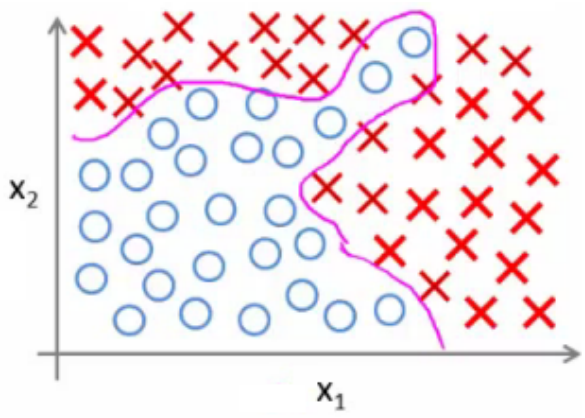
08: Neural Networks - Representation

Previous Next Index

Neural networks - Overview and summary

Why do we need neural networks?

- Say we have a complex supervised learning classification problem
  - Can use logistic regression with many polynomial terms
  - Works well when you have 1-2 features
  - If you have 100 features

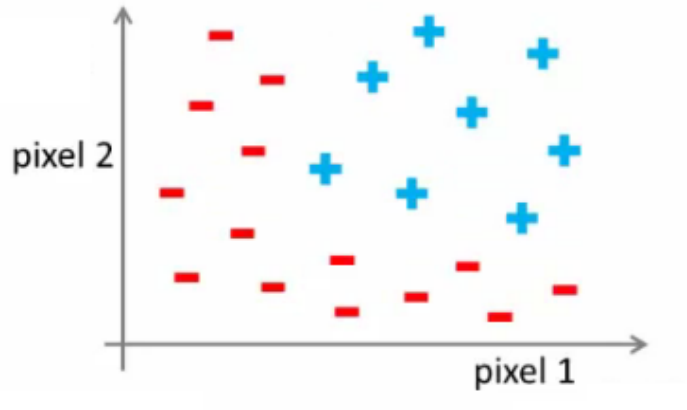


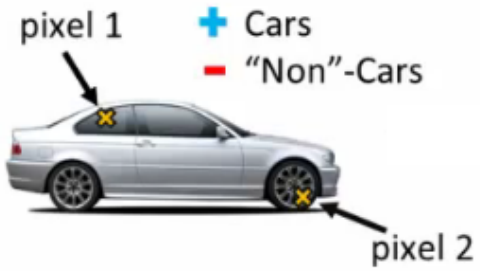
- e.g. our housing example
  - 100 house features, predict odds of a house being sold in the next 6 months
  - Here, if you included all the quadratic terms (second order)
    - There are lots of them (x_1^2 ,x_1x_2, x_1x_4 ..., x_1x_{100})
    - For the case of n = 100, you have about 5000 features
    - Number of features grows $O(n^2)$
    - This would be computationally expensive to work with as a feature set
- A way around this to only include a subset of features
  - However, if you don't have enough features, often a model won't let you fit a complex dataset
- If you include the cubic terms
  - e.g. ($x_1^2x_2$, $x_1x_2x_3$, $x_1x_4x_{23}$ etc)
  - There are even more features grows $O(n^3)$
  - About 170 000 features for n = 100
- Not a good way to build classifiers when n is large

Example: Problems where n is large - computer vision

- Computer vision sees a matrix of pixel intensity values
  - Look at matrix - explain what those numbers represent
- To build a car detector
  - Build a training set of
    - Not cars
    - Cars
  - Then test against a car

- How can we do this
  - Plot two pixels (two pixel locations)
  - Plot car or not car on the graph





- Need a non-linear hypothesis to separate the classes
- Feature space
  - If we used 50 x 50 pixels --> 2500 pixels, so n = 2500
  - If RGB then 7500
  - If 100 x 100 RB then --> 50 000 000 features
- Too big - wayyy too big
  - So - simple logistic regression here is not appropriate for large complex systems
  - Neural networks are much better for a complex nonlinear hypothesis even when feature space is huge

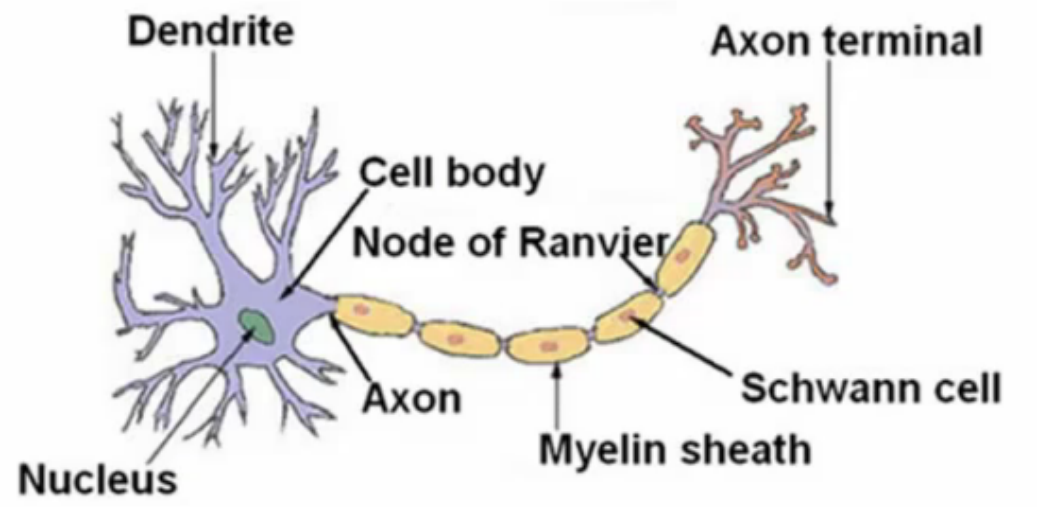
Neurons and the brain

- **Neural networks** (**NNs**) were originally motivated by looking at machines which replicate the brain's functionality
  - Looked at here as a machine learning technique
- Origins
  - To build learning systems, why not mimic the brain?
  - Used a lot in the 80s and 90s
  - Popularity diminished in late 90s
  - Recent major resurgence
    - NNs are computationally expensive, so only recently large scale neural networks became computationally feasible
- Brain
  - Does loads of crazy things
    - Hypothesis is that the brain has a single learning algorithm
  - Evidence for hypothesis
    - Auditory cortex --> takes sound signals
      - If you cut the wiring from the ear to the auditory cortex
      - Re-route optic nerve to the auditory cortex
      - Auditory cortex learns to see
    - Somatosensory context (touch processing)
      - If you rewrite optic nerve to somatosensory cortex then it learns to see
  - With different tissue learning to see, maybe they all learn in the same way
    - Brain learns by itself how to learn
- Other examples

    - Seeing with your tongue
        - Brainport
            - Grayscale camera on head
            - Run wire to array of electrodes on tongue
            - Pulses onto tongue represent image signal
            - Lets people see with their tongue
    - Human echolocation
        - Blind people being trained in schools to interpret sound and echo
        - Lets them move around
    - Haptic belt direction sense
        - Belt which buzzes towards north
        - Gives you a sense of direction
- Brain can process and learn from data from any source

Model representation 1

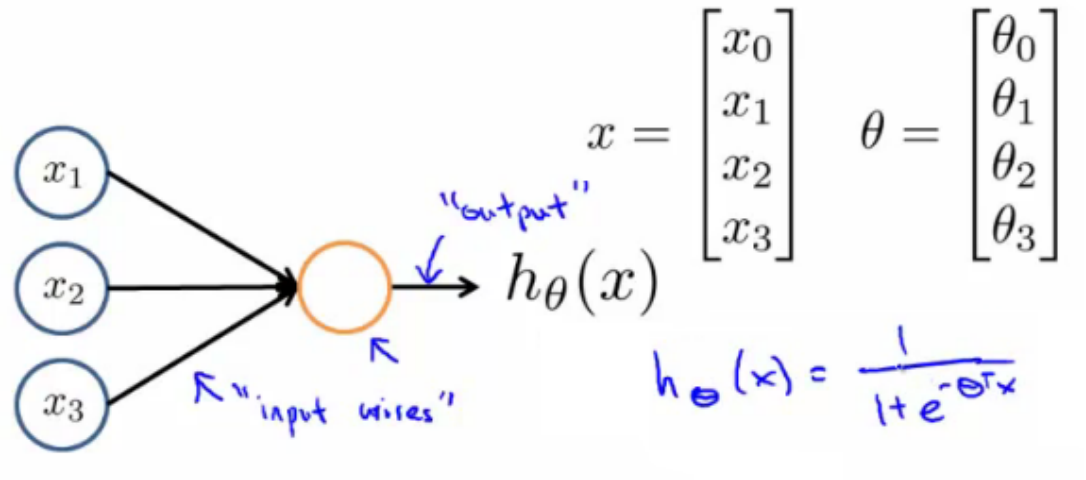
- How do we represent neural networks (NNs)?
    - Neural networks were developed as a way to simulate networks of neurones
- What does a neurone look like



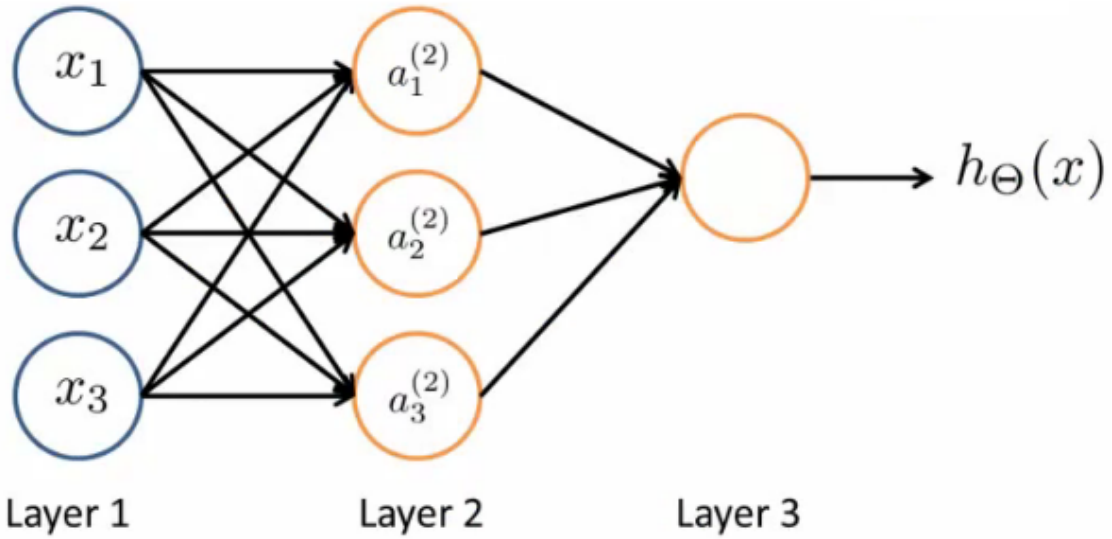
- Three things to notice
    - Cell body
    - Number of input wires (dendrites)
    - Output wire (axon)
- Simple level
    - Neurone gets one or more inputs through dendrites
    - Does processing
    - Sends output down axon
- Neurons communicate through electric spikes
    - Pulse of electricity via axon to another neurone

Artificial neural network - representation of a neurone

- In an artificial neural network, a neurone is a logistic unit
    - Feed input via input wires
    - Logistic unit does computation
    - Sends output down output wires
- That logistic computation is just like our previous logistic regression hypothesis calculation



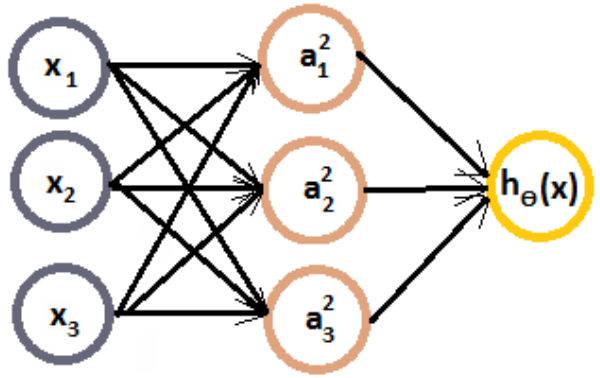
- Very simple model of a neuron's computation
  - Often good to include an x_0 input - the **bias unit**
    - This is equal to 1
- This is an artificial neurone with a sigmoid (logistic) activation function
  - Θ vector may also be called the **weights** of a model
- The above diagram is a single neurone
  - Below we have a group of neurones strung together



- Here, input is x_1, x_2 and x_3
  - We could also call input activation on the first layer - i.e. (a_1^1, a_2^1 and a_3^1)
  - Three neurones in layer 2 (a_1^2, a_2^2 and a_3^2)
  - Final fourth neurone which produces the output
    - Which again we *could* call a_1^3
- First layer is the **input layer**
- Final layer is the **output layer** - produces value computed by a hypothesis
- Middle layer(s) are called the **hidden layers**
  - You don't observe the values processed in the hidden layer
  - Not a great name
  - Can have many hidden layers

www **Neural networks - notation**

- **$a_i^{(j)}$ - activation of unit *i* in layer *j***
  - So, a_1^2 - is the **activation** of the 1st unit in the second layer
  - By activation, we mean the value which is computed and output by that node
- **$\Theta^{(j)}$ - matrix of parameters controlling the function mapping from layer *j* to layer *j* + 1**
  - Parameters for controlling **mapping** from one layer to the next
  - If network has
    - s_j units in layer *j* and
    - s_{j+1} units in layer j + 1
    - Then Θ^j will be of dimensions $[s_{j+1} \text{ X } s_j + 1]$
      - Because
        - s_{j+1} is equal to the number of units in layer (j + 1)
        - is equal to the number of units in layer j, plus an additional unit
  - Looking at the Θ matrix
    - Column length is the number of units in the following layer
    - Row length is the number of units in the current layer + 1 (because we have to map the bias unit)
    - So, if we had two layers - 101 and 21 units in each
      - Then Θ^j would be = [21 x 102]
- What are the computations which occur?
  - We have to calculate the activation for each node
  - That activation depends on
    - The input(s) to the node
    - The parameter associated with that node (from the Θ vector associated with that layer)
- Below we have an example of a network, with the associated calculations for the four nodes below

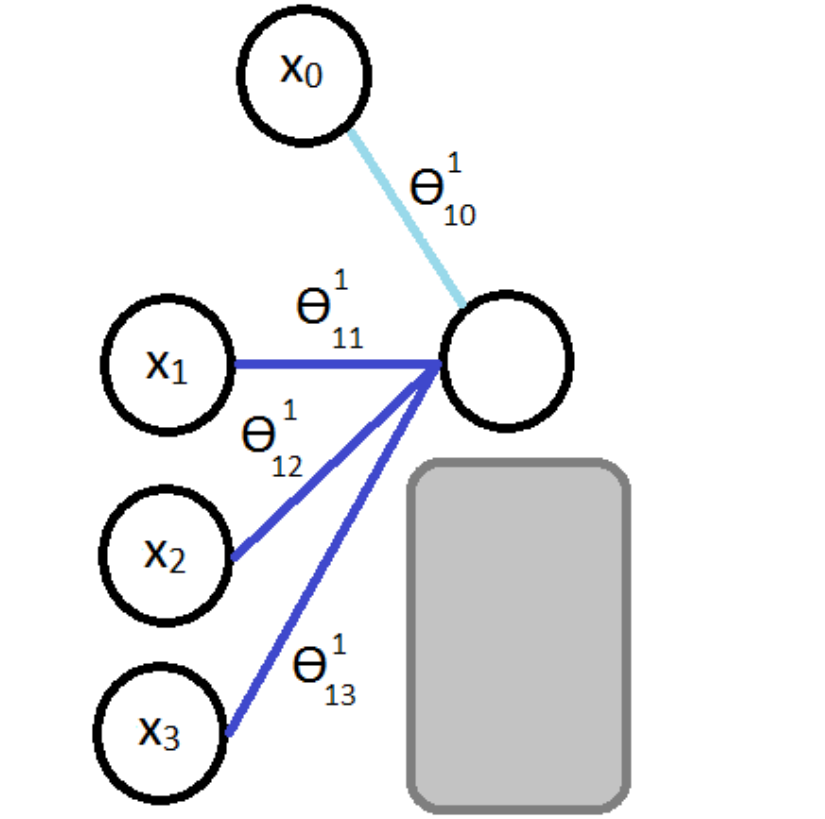


$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

- As you can see

- We calculate each of the layer-2 activations based on the input values with the bias term (which is equal to 1)
  - i.e. x_0 to x_3
- We then calculate the final hypothesis (i.e. the single node in layer 3) using exactly the same logic, except in input is not x values, but the activation values from the preceding layer

- The activation value on each hidden unit (e.g. a_1^2) is equal to the sigmoid function applied to the linear combination of inputs
  - Three input units
    - So $\Theta^{(1)}$ is the matrix of parameters governing the mapping of the input units to hidden units
      - $\Theta^{(1)}$ here is a [3 x 4] dimensional matrix
  - Three hidden units
    - Then $\Theta^{(2)}$ is the matrix of parameters governing the mapping of the hidden layer to the output layer
      - $\Theta^{(2)}$ here is a [1 x 4] dimensional matrix (i.e. a row vector)
  - One output unit
- Something conceptually important (that I hadn't really grasped the first time) is that
  - **Every input/activation goes to every node in following layer**
    - Which means each "layer transition" uses a matrix of parameters with the following significance
      - For the sake of consistency with later nomenclature, we're using j,i and l as our variables here (although later in this section we use j to show the layer we're on)
      - $\boldsymbol{\Theta}_{ji}^{l}$
        - j (first of two subscript numbers)= ranges from 1 to the number of units in layer l+1
        - i (second of two subscript numbers) = ranges from 0 to the number of units in layer l
        - l is the layer you're moving FROM
      - This is perhaps more clearly shown in my slightly over the top example below



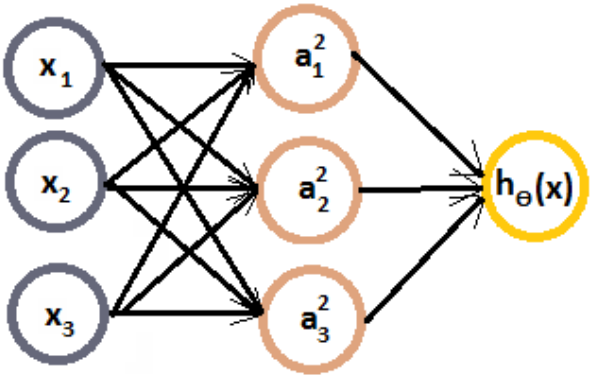
$$g(\Theta_{10}^{1}x_0 + \Theta_{11}^{1}x_1 + \Theta_{12}^{1}x_2 + \Theta_{13}^{1}x_3)$$

- For example
  - Θ_{13}^{1} = means
    - **1** - we're mapping to node 1 in layer l+1
    - **3** - we're mapping from node 3 in layer l
    - **1** - we're mapping from layer 1

Model representation II

Here we'll look at how to carry out the computation efficiently through a vectorized implementation. We'll also consider
why NNs are good and how we can use them to learn complex non-linear things

- Below is our original problem from before
  - Sequence of steps to compute output of hypothesis are the equations below



$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

- Define some additional terms
  - $z_1^2 = \Theta_{10}^1 x_0 + \Theta_{11}^1 x_1 + \Theta_{12}^1 x_2 + \Theta_{13}^1 x3$
  - Which means that
    - $a_1^2 = g(z_1^2)$
  - NB, superscript numbers are the layer associated
- Similarly, we define the others as
  - z_2^2 and z_3^2
  - These values are just a linear combination of the values
- If we look at the block we just redefined
  - We can vectorize the neural network computation
  - So lets define
    - x as the feature vector x
    - z^2 as the vector of z values from the second layer

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$$

- z^2 is a 3x1 vector
- We can vectorize the computation of the neural network as as follows in two steps
  - $z^2 = \Theta^{(1)}x$
    - i.e. $\Theta^{(1)}$ is the matrix defined above
    - x is the feature vector
  - $a^2 = g(z^{(2)})$
    - To be clear, z^2 is a 3x1 vecor
    - a^2 is also a 3x1 vector

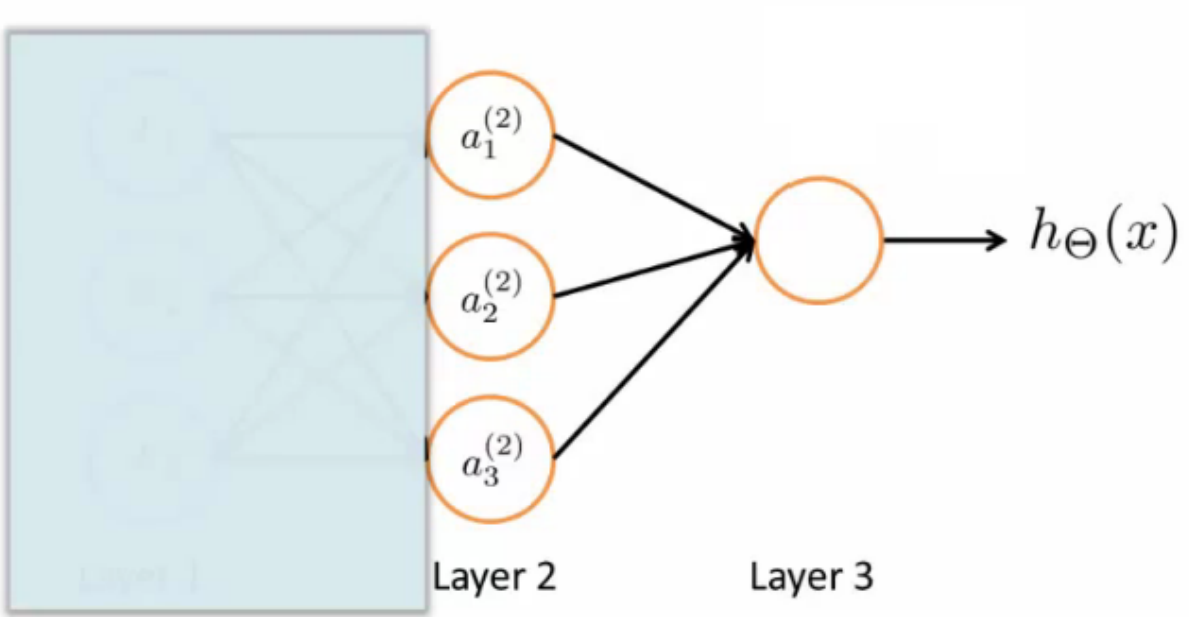
        - g() applies the sigmoid (logistic) function element wise to each member of the z^2 vector
- To make the notation with input layer make sense;
    - $a^1 = x$
        - a^1 is the activations in the input layer
        - Obviously the "activation" for the input layer is just the input!
    - So we define x as a^1 for clarity
        - So
            - a^1 is the vector of inputs
            - a^2 is the vector of values calculated by the $g(z^2)$ function
- Having calculated then z^2 vector, we need to calculate a_0^2 for the final hypothesis calculation

$$h_\Theta(x) = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

- To take care of the extra bias unit add $a_0^2 = 1$
    - So add a_0^2 to a^2 making it a 4x1 vector
- So,
    - $z^3 = \Theta^2 a^2$
        - This is the inner term of the above equation
    - $h_\Theta(x) = a^3 = g(z^3)$
- This process is also called **forward propagation**
    - Start off with activations of input unit
        - i.e. the x vector as input
    - Forward propagate and calculate the activation of each layer sequentially
    - This is a vectorized version of this implementation

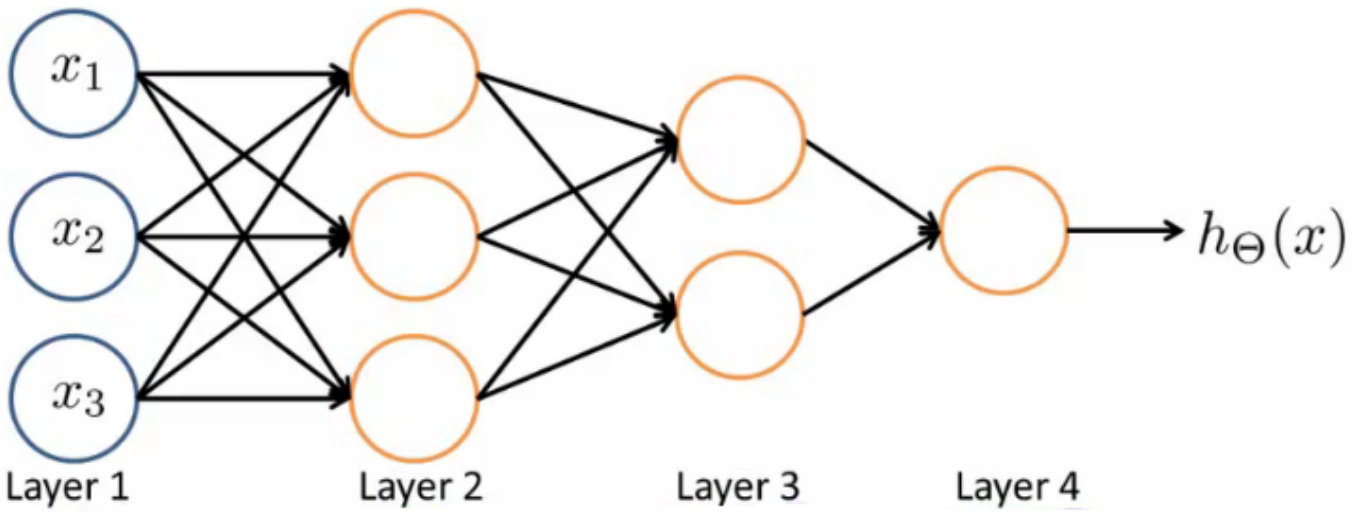
Neural networks learning its own features

- Diagram below looks a lot like logistic regression



- Layer 3 is a logistic regression node
    - The hypothesis output = $g(\Theta_{10}^2 a_0^2 + \Theta_{11}^2 a_1^2 + \Theta_{12}^2 a_2^2 + \Theta_{13}^2 a_3^2)$

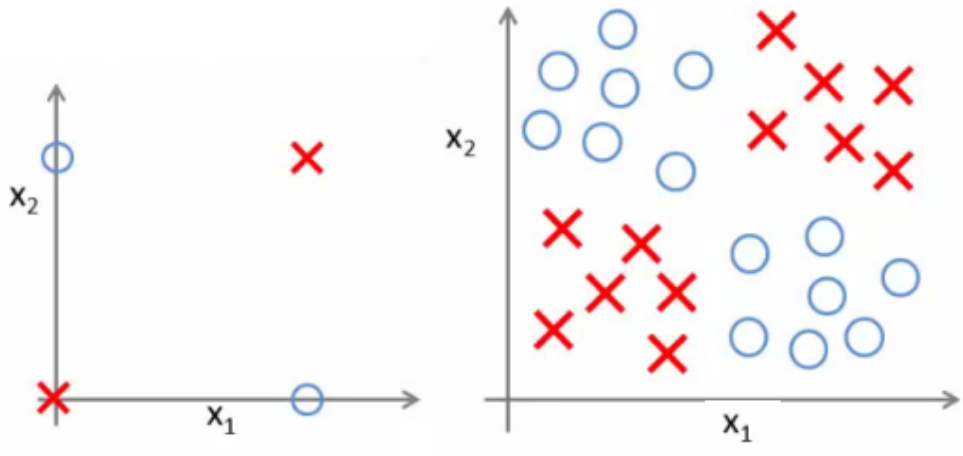
    - This is just logistic regression
        - The only difference is, instead of input a feature vector, the features are just values calculated by the hidden layer
- The features a_1^2, a_2^2, and a_3^2 are calculated/learned - not original features
- So the mapping from layer 1 to layer 2 (i.e. the calculations which generate the a^2 features) is determined by another set of parameters - Θ^1
    - So instead of being constrained by the original input features, a neural network can learn its own features to feed into logistic regression
    - Depending on the Θ^1 parameters you can learn some interesting things
        - Flexibility to learn whatever features it wants to feed into the final logistic regression calculation
            - So, if we compare this to previous logistic regression, you would have to calculate your own exciting features to define the best way to classify or describe something
            - Here, we're letting the hidden layers do that, so we feed the hidden layers our input values, and let them learn whatever gives the best final result to feed into the final output layer
- As well as the networks already seen, other architectures (topology) are possible
    - More/less nodes per layer
    - More layers
    - Once again, layer 2 has three hidden units, layer 3 has 2 hidden units by the time you get to the output layer you get very interesting non-linear hypothesis



- Some of the intuitions here are complicated and hard to understand
    - In the following lectures we're going to go though a detailed example to understand how to do non-linear analysis

Neural network example - computing a complex, nonlinear function of the input

- Non-linear classification: XOR/XNOR
    - x_1, x_2 are binary



- Example on the right shows a simplified version of the more complex problem we're dealing with (on the left)
- We want to learn a non-linear decision boundary to separate the positive and negative examples

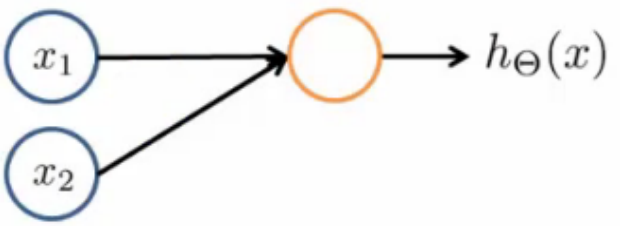
$y = x_1$ XOR x_2
x_1 XNOR x_2

Where XNOR = NOT (x_1 XOR x_2)

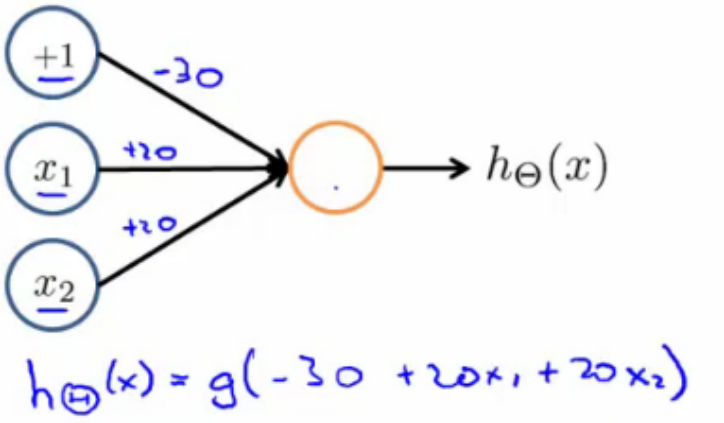
- Positive examples when both are true and both are false
  - Let's start with something a little more straight forward...
  - Don't worry about how we're determining the weights (Θ values) for now - just get a flavor of how NNs work

Neural Network example 1: AND function

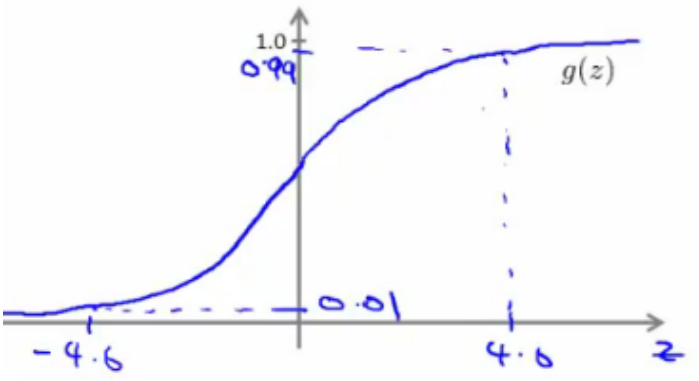
- Simple first example



- Can we get a one-unit neural network to compute this logical AND function? (*probably*...)
  - Add a bias unit
  - Add some weights for the networks
    - What are weights?
      - Weights are the parameter values which multiply into the input nodes (i.e. Θ)



- Sometimes it's convenient to add the weights into the diagram
  - These values are in fact just the Θ parameters so
    - $\Theta_{10}^{1} = -30$
    - $\Theta_{11}^{1} = 20$
    - $\Theta_{12}^{1} = 20$
  - To use our original notation
- Look at the four input values



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	$g(10) \approx 1$

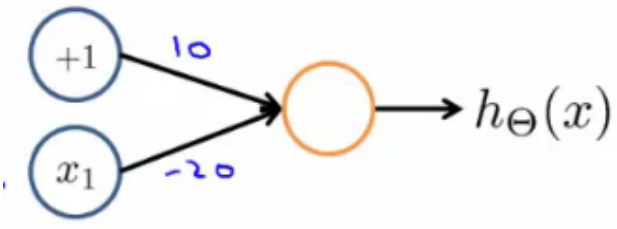
$h_\Theta(x) \approx x_1$ AND x_2

Sigmoid function (reminder)

- So, as we can see, when we evaluate each of the four possible input, only (1,1) gives a positive output

Neural Network example 2: NOT function

- How about negation?

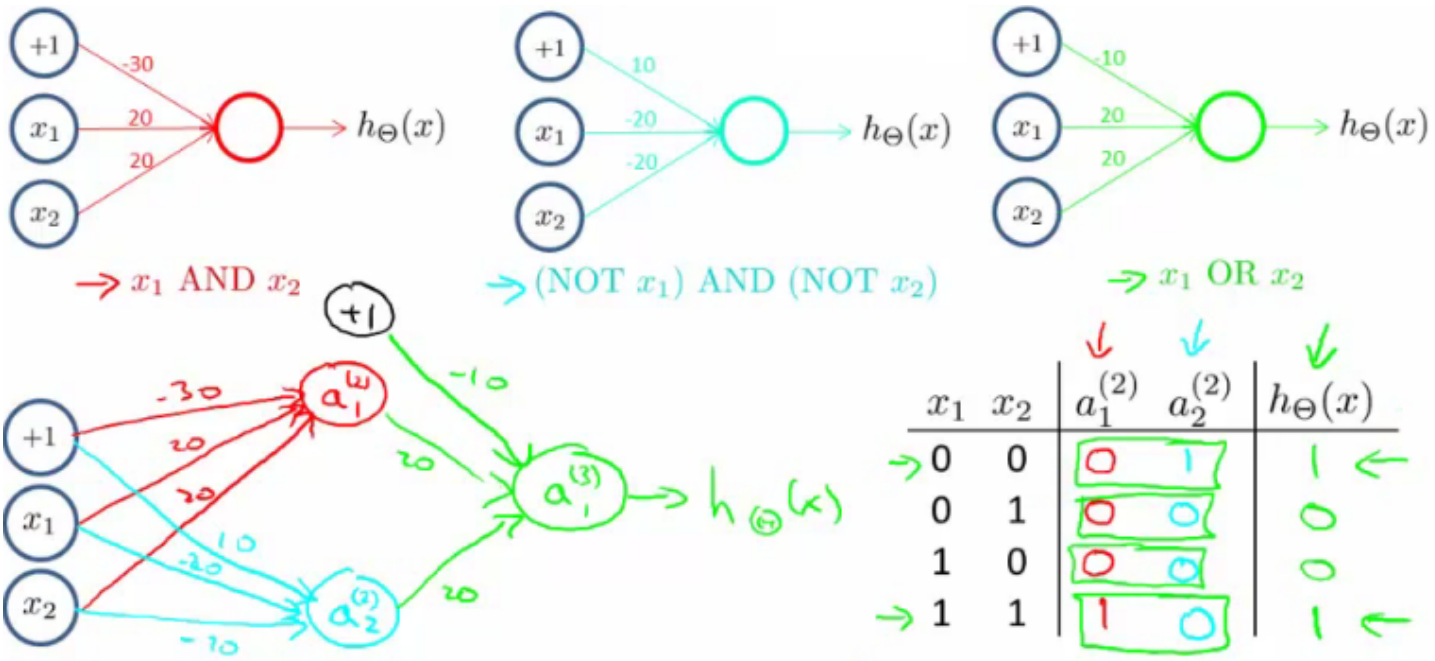


x_1	$h_\Theta(x)$
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

- Negation is achieved by putting a large negative weight in front of the variable you want to negative

Neural Network example 3: XNOR function

- So how do we make the XNOR function work?
  - XNOR is short for NOT XOR
    - i.e. NOT an exclusive or, so either go big (1,1) or go home (0,0)
  - So we want to structure this so the input which produce a positive output are
    - AND (i.e. both true)
      OR
    - Neither (which we can shortcut by saying not only one being true)
- So we combine these into a neural network as shown below;



- Simplez!

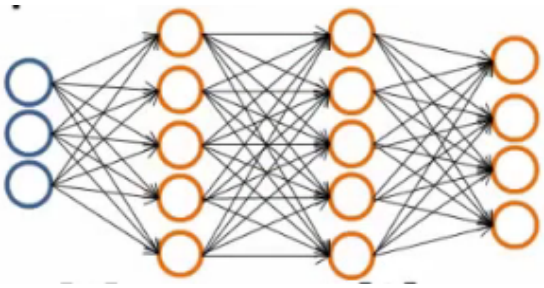
Neural network intuition - handwritten digit classification

- Yann LeCun = machine learning pioneer
- Early machine learning system was postcode reading
  - Hilarious music, impressive demonstration!

Multiclass classification

- Multiclass classification is, unsurprisingly, when you distinguish between more than two categories (i.e. more than 1 or 0)
- With handwritten digital recognition problem - 10 possible categories (0-9)
  - How do you do that?
  - Done using an extension of one vs. all classification
- Recognizing pedestrian, car, motorbike or truck
  - Build a neural network with four output units
  - Output a vector of four numbers

        - 1 is 0/1 pedestrian
        - 2 is 0/1 car
        - 3 is 0/1 motorcycle
        - 4 is 0/1 truck
    - When image is a pedestrian get [1,0,0,0] and so on
- Just like one vs. all described earlier
    - Here we have four logistic regression classifiers



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

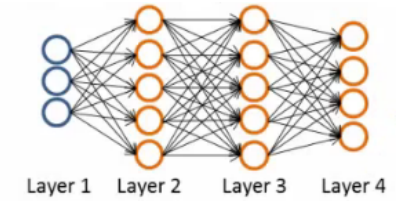
- Training set here is images of our four classifications
    - While previously we'd written y as an integer {1,2,3,4}
    - Now represent y as
- $y^{(i)}$ one of $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

09: Neural Networks - Learning

Previous Next Index

Neural network cost function

- NNs - one of the most powerful learning algorithms
  - Is a learning algorithm for fitting the derived parameters given a training set
  - Let's have a first look at a neural network cost function
- Focus on application of NNs for classification problems
- Here's the set up
  - Training set is $\{(x^1, y^1), (x^2, y^2), (x^3, y^3) \ldots (x^n, y^m)$
  - L = number of layers in the network
    - In our example below L = 4
  - s_l = number of units (not counting bias unit) in layer l



- So here
  - l = 4
  - $s_1 = 3$
  - $s_2 = 5$
  - $s_3 = 5$
  - $s_4 = 4$

Types of classification problems with NNs

- Two types of classification, as we've previously seen
- **Binary classification**
  - 1 output (0 or 1)
  - So single output node - value is going to be a real number
  - k = 1
    - NB k is number of units in output layer
  - $s_L = 1$
- **Multi-class classification**
  - k distinct classifications
  - Typically k is greater than or equal to three
  - If only two just go for binary
  - $s_L = k$
  - So y is a k-dimensional vector of real numbers

$y \in \mathbb{R}^K$ 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

Cost function for neural networks

- The (regularized) logistic regression cost function is as follows;

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

- For neural networks our cost function is a generalization of this equation above, so instead of one output we generate k outputs

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

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

- Our cost function now outputs a k dimensional vector

- $h_\Theta(x)$ is a k dimensional vector, so $h_\Theta(x)_i$ refers to the ith value in that vector
- Costfunction J(Θ) is
  - [-1/m] times a sum of a similar term to which we had for logic regression
  - But now this is also a sum from k = 1 through to K (K is number of output nodes)
    - Summation is a sum over the k output units - i.e. for each of the possible classes
    - So if we had 4 output units then the sum is k = 1 to 4 of the logistic regression over each of the four output units in turn
  - This looks really complicated, but it's not so difficult
    - We don't sum over the bias terms (hence starting at 1 for the summation)
      - Even if you do and end up regularizing the bias term this is not a big problem
    - Is just summation over the terms

Woah there - lets take a second to try and understand this!

- There are basically two halves to the neural network logistic regression cost function

First half

$$-\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1-y_k^{(i)}) \log(1-(h_\Theta(x^{(i)}))_k)\right]$$

- This is just saying
  - For each training data example (i.e. 1 to m - the first summation)
    - Sum for each position in the output vector
- This is an average sum of logistic regression

Second half

$$\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

- This is a massive regularization summation term, which I'm not going to walk through, but it's a fairly straightforward triple nested summation
- This is also called a **weight decay** term
- As before, the lambda value determines the important of the two halves

- The regularization term is similar to that in logistic regression
- So, we have a cost function, but *how* do we minimize this bad boy?!

Summary of what's about to go down

The following section is, I think, the most complicated thing in the course, so I'm going to take a second to explain the general idea of what we're going to do;

- We've already described **forward propagation**
  - This is the algorithm which takes your neural network and the initial input into that network and pushes the input through the network
    - It leads to the generation of an output hypothesis, which may be a single real number, but can also be a vector
- We're now going to describe **back propagation**
  - Back propagation basically takes the output you got from your network, compares it to the real value (y) and calculates how wrong the network was (i.e. how wrong the parameters were)
  - It then, using the error you've just calculated, back-calculates the error associated with each unit from the preceding layer (i.e. layer *L* - 1)
  - This goes on until you reach the input layer (where obviously there is no error, as the activation is the input)
  - These "error" measurements for each unit can be used to calculate the **partial derivatives**
    - Partial derivatives are the bomb, because gradient descent needs them to minimize the cost function
  - We use the partial derivatives with gradient descent to try minimize the cost function and update all the Θ values
  - This repeats until gradient descent reports convergence

- A few things which are good to realize from the get go
  - There is a Θ matrix for each layer in the network
    - This has each node in layer l as one dimension and each node in l+1 as the other dimension
  - Similarly, there is going to be a Δ matrix for each layer
    - This has each node as one dimension and each training data example as the other

Back propagation algorithm

- We previously spoke about the neural network cost function
- Now we're going to deal with **back propagation**

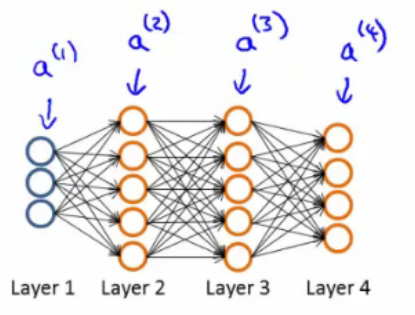
- Algorithm used to minimize the cost function, as it **allows us to calculate partial derivatives**!

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

- The cost function used is shown above
  - We want to find parameters Θ which minimize J(Θ)
  - To do so we can use one of the algorithms already described such as
    - Gradient descent
    - Advanced optimization algorithms
- To minimize a cost function we just write code which computes the following
  - **J(Θ)**
    - i.e. the cost function itself!
    - Use the formula above to calculate this value, so we've done that
  - **Partial derivative terms**
    - So now we need some way to do that
      - This is not trivial! Θ is indexed in three dimensions because we have separate parameter values for each node in each layer going to each node in the following layer
      - i.e. each layer has a Θ matrix associated with it!
        - We want to calculate the partial derivative Θ with respect to a single parameter

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

    - Remember that the partial derivative term we calculate above is a REAL number (not a vector or a matrix)
      - Θ is the input parameters
        - Θ^1 is the matrix of weights which define the function mapping from layer 1 to layer 2
        - Θ_{10}^1 is the real number parameter which you multiply the bias unit (i.e. 1) with for the bias unit input into the first unit in the second layer
        - Θ_{11}^1 is the real number parameter which you multiply the first (real) unit with for the first input into the first unit in the second layer
        - Θ_{21}^1 is the real number parameter which you multiply the first (real) unit with for the first input into the second unit in the second layer
        - As discussed, Θ_{ij}^l i
          - i here represents the unit in layer l+1 you're mapping to (destination node)
          - j is the unit in layer l you're mapping from (origin node)
          - l is the layer your mapping from (to layer l+1) (origin layer)
          - NB
            - *The terms destination node, origin node and origin layer are terms I've made up!*
    - So - this partial derivative term is
      - The partial derivative of a 3-way indexed dataset with respect to a real number (which is one of the values in that dataset)
  - **Gradient computation**
    - One training example
    - Imagine we just have a single pair (x,y) - entire training set
    - How would we deal with this example?
    - The forward propagation algorithm operates as follows
      - **Layer 1**
        - $a^1 = x$
        - $z^2 = \Theta^1 a^1$
      - **Layer 2**
        - $a^2 = g(z^2)$ (add a_0^2)
        - $z^3 = \Theta^2 a^2$
      - **Layer 3**
        - $a^3 = g(z^3)$ (add a_0^3)
        - $z^4 = \Theta^3 a^3$
      - **Output**
        - $a^4 = h_\Theta(x) = g(z^4)$



- This is the vectorized implementation of forward propagation
  - Lets compute activation values sequentially (below just re-iterates what we had above!)

$$
\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)}) \quad (\text{add } a_0^{(2)}) \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
a^{(3)} &= g(z^{(3)}) \quad (\text{add } a_0^{(3)}) \\
z^{(4)} &= \Theta^{(3)} a^{(3)} \\
a^{(4)} &= h_\Theta(x) = g(z^{(4)})
\end{aligned}
$$

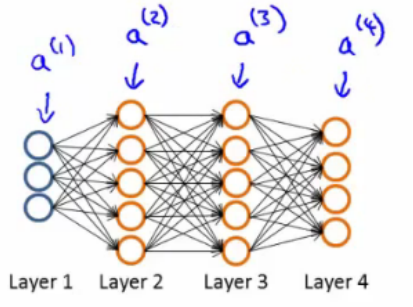
What is back propagation?

- Use it to compute the partial derivatives
- Before we dive into the mechanics, let's get an idea regarding the intuition of the algorithm
  - For each node we can calculate (δ_j^l) - this is **the error of node j in layer l**
    - If we remember, a_j^l is the activation of node j in layer l
    - Remember the activation is a totally calculated value, so we'd expect there to be some error compared to the "real" value
      - The delta term captures this error
      - But the problem here is, "what is this 'real' value, and how do we calculate it?!"
        - The NN is a totally artificial construct
        - The only "real" value we have is our actual classification (our y value) - so that's where we start
- If we use our example and look at the fourth (output) layer, we can first calculate
  - $\delta_j^4 = a_j^4 - y_j$
    - [Activation of the unit] - [the actual value observed in the training example]
    - We could also write a_j^4 as $h_\Theta(x)_j$
      - Although I'm not sure why we would?
  - This is an individual example implementation
- Instead of focussing on each node, let's think about this as a vectorized problem
  - $\delta^4 = a^4 - y$
    - So here δ^4 is the vector of errors for the 4th layer
    - a^4 is the vector of activation values for the 4th layer
- With δ^4 calculated, we can determine the error terms for the other layers as follows;

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

- Taking a second to break this down
  - Θ^3 is the vector of parameters for the 3->4 layer mapping
  - δ^4 is (as calculated) the error vector for the 4th layer
  - $g'(z^3)$ is the first derivative of the activation function g evaluated by the input values given by z^3
    - You can do the calculus if you want (...), but when you calculate this derivative you get
    - $g'(z^3) = a^3 .* (1 - a^3)$
  - So, more easily
    - **$\delta^3 = (\Theta^3)^T \delta^4 .* (a^3 .* (1 - a^3))$**
  - **.*** is the element wise multiplication between the two vectors
    - Why element wise? Because this is essentially an extension of individual values in a vectorized implementation, so element wise multiplication gives that effect
    - We highlighted it just in case you think it's a typo!

Analyzing the mathematics



- And if we take a second to consider the vector dimensionality (with our example above [3-5-5-4])
  - Θ^3 = is a matrix which is [4 X 5] (if we don't include the bias term, 4 X 6 if we do)
    - $(\Theta^3)^T$ = therefore, is a [5 X 4] matrix
  - δ^4 = is a 4x1 vector
  - So when we multiply a [5 X 4] matrix with a [4 X 1] vector we get a [5 X 1] vector
  - Which, low and behold, is the same dimensionality as the a^3 vector, meaning we can run our pairwise multiplication

- For δ^3 when you calculate the derivative terms you get
  a^3 **.** * $(1 - a^3)$
- Similarly For δ^2 when you calculate the derivative terms you get
  a^2 **.** * $(1 - a^2)$
  - So to calculate δ^2 we do
    $\delta^2 = (\Theta^2)^T \delta^3$ **.** * $(a^2$ **.** * $(1 - a^2))$
- There's no δ^1 term
  - Because that was the input!

Why do we do this?

- We do all this to get all the δ terms, and we want the δ terms because through a very complicated derivation you can use δ to get the partial derivative of Θ with respect to individual parameters (if you ignore regularization, or regularization is 0, which we deal with later)
- $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$ = $a_j^l \delta_i^{(l+1)}$
- By doing back propagation and computing the delta terms you can then compute the **partial derivative terms**
  - We need the partial derivatives to minimize the cost function!

Putting it all together to get the partial derivatives!

- What is really happening - lets look at a more complex example
- Training set of m examples
  Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
- **First**, set the delta values
  Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j)
  - Set equal to 0 for all values
  - Eventually these Δ values will be used to compute the partial derivative
    - Will be used as accumulators for computing the partial derivatives
- **Next**, loop through the training set
  For $i = 1$ to m
  - i.e. for each example in the training set (dealing with each example as (x,y)
  - Set a^1 (activation of input layer) = x^i
  - **Perform forward propagation** to compute a^l for each layer (l = 1,2, … L)
    - i.e. run forward propagation
  - **Then**, use the output label for the specific example we're looking at to calculate δ^L where $\delta^L = a^L - y^i$
    - So we initially calculate the delta value for the output layer
    - Then, using **back propagation** we move back through the network from layer L-1 down to layer
  - Finally, use Δ to accumulate the partial derivative terms
    $\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
  - Note here
    - l = layer
    - j = node in that layer
    - i = the error of the affected node in the target layer

- You can vectorize the Δ expression too, as

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

- **Finally**
  - After executing the body of the loop, exit the for loop and compute

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

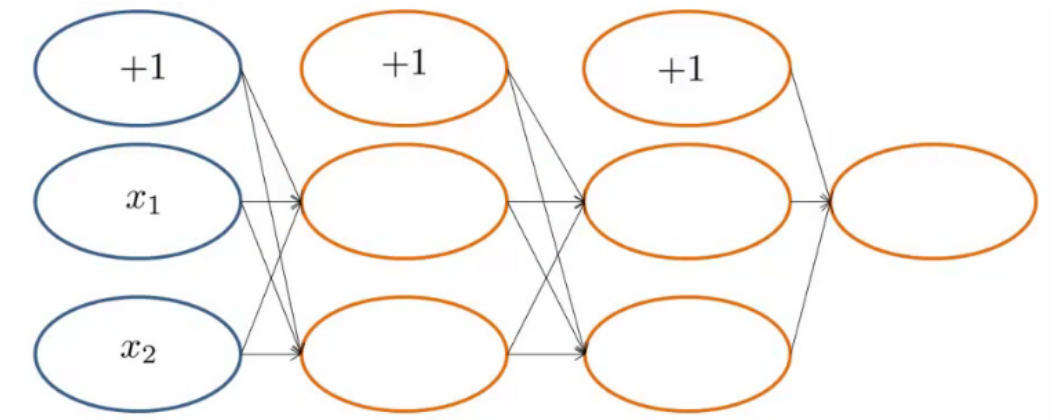
$$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} \text{ if } j = 0$$

    - When j = 0 we have no regularization term
- At the end of ALL this
  - You've calculated all the *D* terms above using Δ
    - NB - each D term above is a real number!
  - We can show that each D is equal to the following
    - $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$
  - We have calculated the partial derivative for each parameter
    - We can then use these in gradient descent or one of the advanced optimization algorithms
- Phew!
  - What a load of hassle!

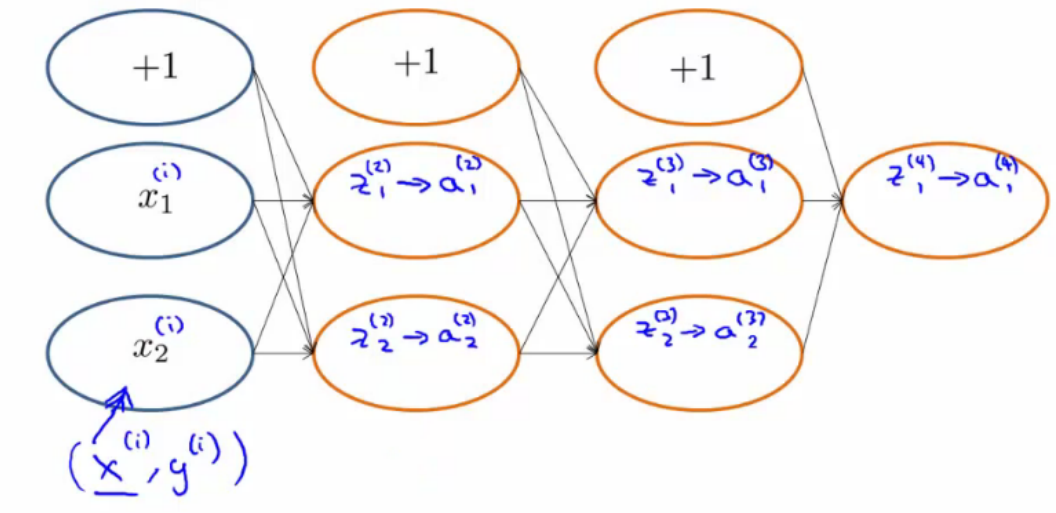
Back propagation intuition

- Some additionally back propagation notes
  - In case you found the preceding unclear, which it shouldn't be as it's fairly heavily modified with my own explanatory notes
- Back propagation is hard(ish...)
  - But don't let that discourage you
  - It's hard in as much as it's confusing - it's not difficult, just complex
- Looking at mechanical steps of back propagation

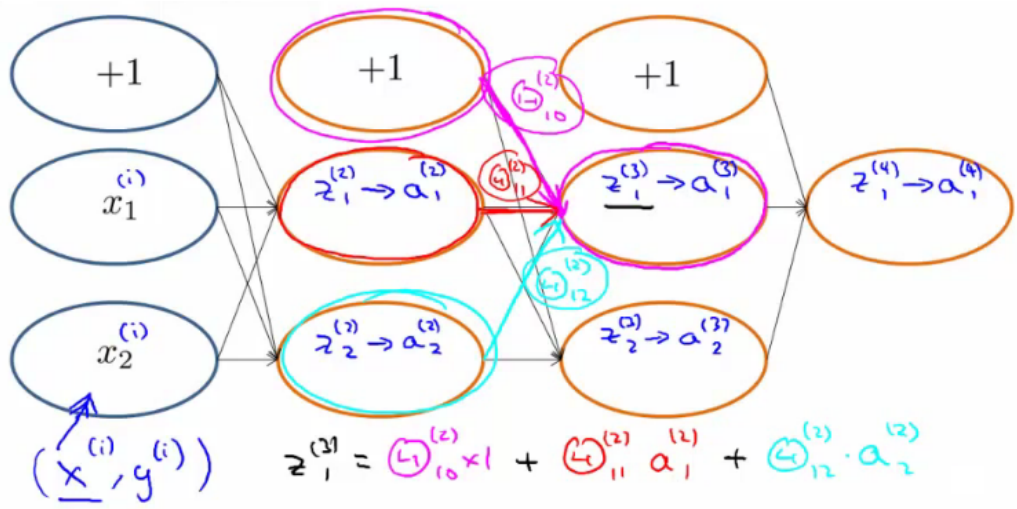
Forward propagation with pictures!



- Feeding input into the input layer (x^i, y^i)
  - Note that x and y here are vectors from 1 to n where n is the number of features
    - So above, our data has two features (hence x_1 and x_2)
- With out input data present we use **forward propagation**



- The sigmoid function applied to the z values gives the activation values
  - Below we show exactly how the z value is calculated for an example



Back propagation

- With forwardprop done we move on to do back propagation
- Back propagation is doing something very similar to forward propagation, but backwards
  - Very similar though
- Let's look at the cost function again...
  - Below we have the cost function if there is a single output (i.e. binary classification)

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

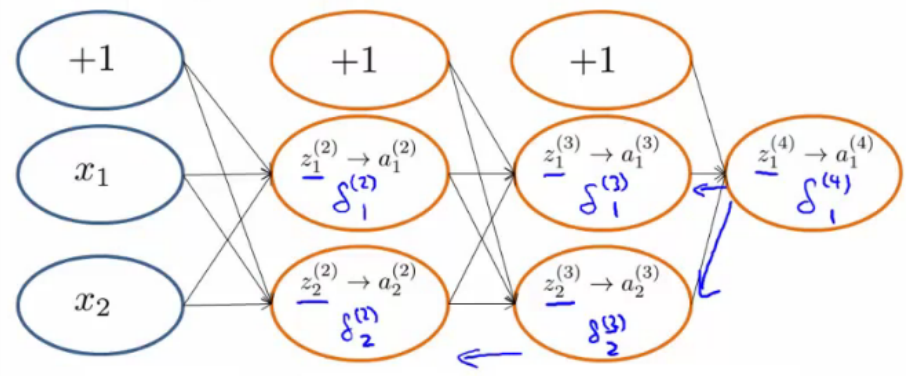
- This function cycles over each example, so the cost for one example really boils down to this

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

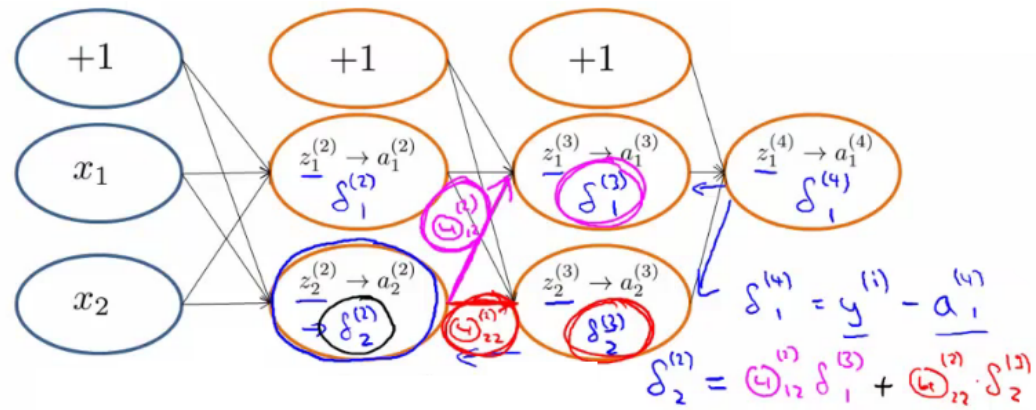
- Which, we can think of as a sigmoidal version of the squared difference (check out the derivation if you don't believe me)
  - So, basically saying, "how well is the network doing on example *i* "?
- We can think about a δ term on a unit as the "error" of cost for the activation value associated with a unit
  - More formally (*don't worry about this...*), δ is

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost(i)}$$

    - Where cost is as defined above
    - Cost function is a function of y value and the hypothesis function
- So - for the output layer, back propagation sets the δ value as [a - y]
  - Difference between activation and actual value
- We then propagate these values backwards;



- Looking at another example to see *how* we actually calculate the delta value;



- So, in effect,
  - Back propagation calculates the δ, and those δ values are the weighted sum of the next layer's delta values, weighted by the parameter associated with the links
  - Forward propagation calculates the activation (a) values, which
- Depending on how you implement you may compute the delta values of the bias values
  - However, these aren't actually used, so it's a bit inefficient, but not a lot more!

Implementation notes - unrolling parameters (matrices)

- Needed for using advanced optimization routines

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

- Is the MATLAB/octave code
  - But theta is going to be matrices
- fminunc takes the costfunction and initial theta values
  - These routines assume theta is a parameter vector
  - Also assumes the gradient created by costFunction is a vector
- For NNs, our parameters are matrices
  - e.g.

Neural Network (L=4):

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

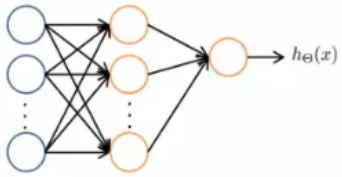
$D^{(1)}, D^{(2)}, D^{(3)}$ - matrices (`D1, D2, D3`)

Example

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

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

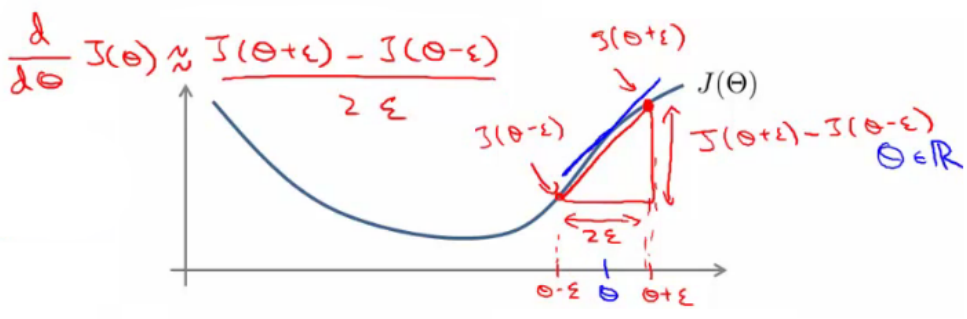
$D^{(1)} \in \mathbb{R}^{10\times 11}, D^{(2)} \in \mathbb{R}^{10\times 11}, D^{(3)} \in \mathbb{R}^{1\times 11}$



- Use the `thetaVec = [ Theta1(:); Theta2(:); Theta3(:)];` notation to unroll the matrices into a long vector
- To go back you use
  - `Theta1 = resape(thetaVec(1:110), 10, 11)`

Gradient checking

- Backpropagation has a lot of details, small bugs can be present and ruin it :-(
  - This may mean it looks like J(Θ) is decreasing, but in reality it may not be decreasing by as much as it should
- So using a numeric method to check the gradient can help diagnose a bug
  - Gradient checking helps make sure an implementation is working correctly
- **Example**
  - Have an function J(Θ)
  - Estimate derivative of function at point Θ (where Θ is a real number)
  - How?
    - Numerically
      - Compute Θ + ε
      - Compute Θ - ε
      - Join them by a straight line
      - Use the slope of that line as an approximation to the derivative



- Usually, epsilon is pretty small (0.0001)
  - If epsilon becomes REALLY small then the term BECOMES the slopes derivative
- The is the two sided difference (as opposed to one sided difference, which would be J(Θ + ε) - J(Θ) /ε
- If Θ is a vector with n elements we can use a similar approach to look at the partial derivatives

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

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

$$\vdots$$

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

- So, in octave we use the following code the numerically compute the derivatives

```
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;
```

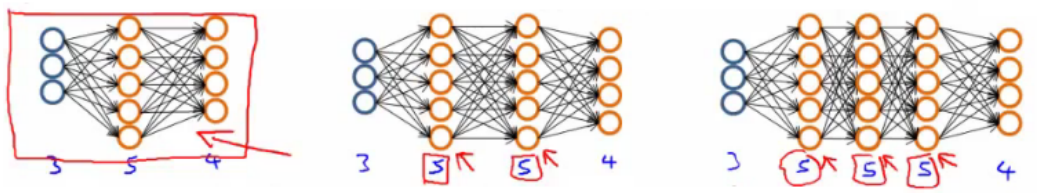
- So on each loop thetaPlus = theta except for thetaPlus(i)
  - Resets thetaPlus on each loop
- Create a vector of partial derivative approximations
- Using the vector of gradients from backprop (DVec)
  - Check that gradApprox is basically equal to DVec
  - Gives confidence that the Backproc implementation is correc
- Implementation note
  - Implement back propagation to compute DVec
  - Implement numerical gradient checking to compute gradApprox
  - Check they're basically the same (up to a few decimal places)
  - Before using the code for learning turn off gradient checking
    - Why?
      - GradAprox stuff is very computationally expensive
      - In contrast backprop is much more efficient (just more fiddly)

Random initialization

- Pick random small initial values for all the theta values
  - If you start them on zero (which does work for linear regression) then the algorithm fails - all activation values for each layer are the same
- So chose random values!
  - Between 0 and 1, then scale by epsilon (where epsilon is a constant)

Putting it all together

- **1) - pick a network architecture**
  - Number of
    - **Input units** - number of dimensions x (dimensions of feature vector)
    - **Output units** - number of classes in classification problem
    - **Hidden units**
      - Default might be
        - 1 hidden layer
      - Should probably have
        - Same number of units in each layer
        - Or 1.5-2 x number of input features
      - Normally
        - More hidden units is better
        - But more is more computational expensive
  - We'll discuss architecture more later



- **2) - Training a neural network**
  - **2.1)** Randomly initialize the weights
    - Small values near 0
  - **2.2)** Implement forward propagation to get $h_\Theta(x)^i$ for any x^i
  - **2.3)** Implement code to compute the cost function $J(\Theta)$
  - **2.4)** Implement back propagation to compute the partial derivatives

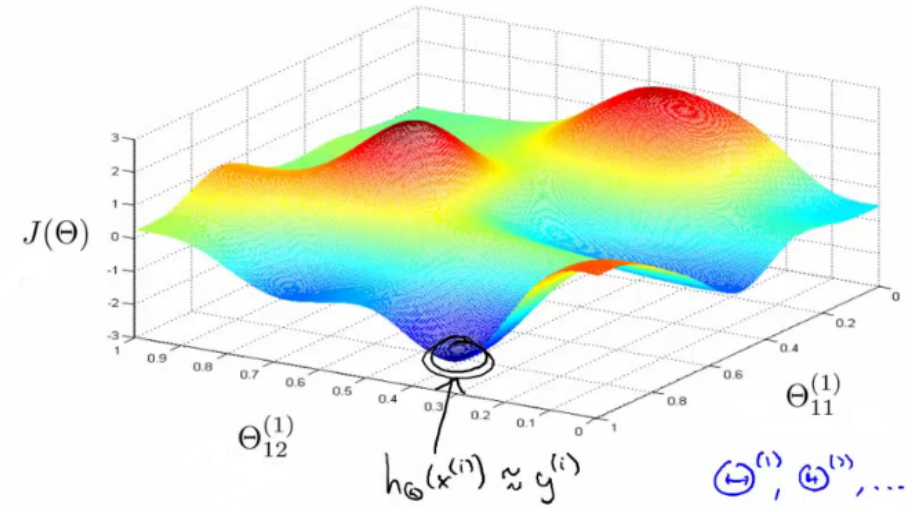
  - General implementation below

```
for i = 1:m {
     Forward propagation on (x^i, y^i) --> get activation (a) terms
     Back propagation on (x^i, y^i) --> get delta (δ) terms
     Compute Δ := Δ^l + δ^(l+1)(a^l)^T
}

With this done compute the partial derivative terms
```

  - Notes on implementation
    - Usually done with a for loop over training examples (for forward and back propagation)
    - *Can* be done without a for loop, but this is a much more complicated way of doing things
    - Be careful

- **2.5)** Use gradient checking to compare the partial derivatives computed using the above algorithm and numerical estimation of gradient of J(Θ)
  - Disable the gradient checking code for when you actually run it
- **2.6)** Use gradient descent or an advanced optimization method with back propagation to try to minimize J(Θ) as a function of parameters Θ
  - Here J(Θ) is non-convex
    - Can be susceptible to local minimum
    - In practice this is not usually a huge problem
    - Can't guarantee programs with find global optimum should find good local optimum at least



- e.g. above pretending data only has two features to easily display what's going on
  - Our minimum here represents a hypothesis output which is pretty close to y
  - If you took one of the peaks hypothesis is far from y
- Gradient descent will start from some random point and move downhill
  - Back propagation calculates gradient down that hill