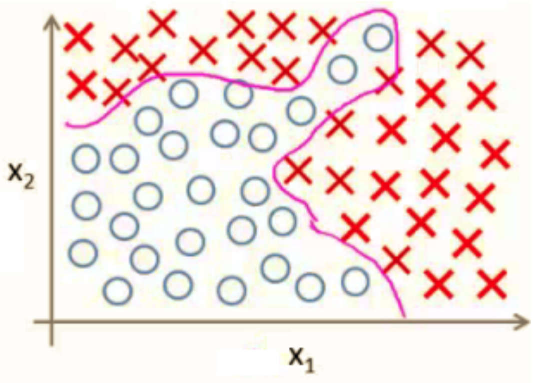
# 08: Neural Networks - Representation

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

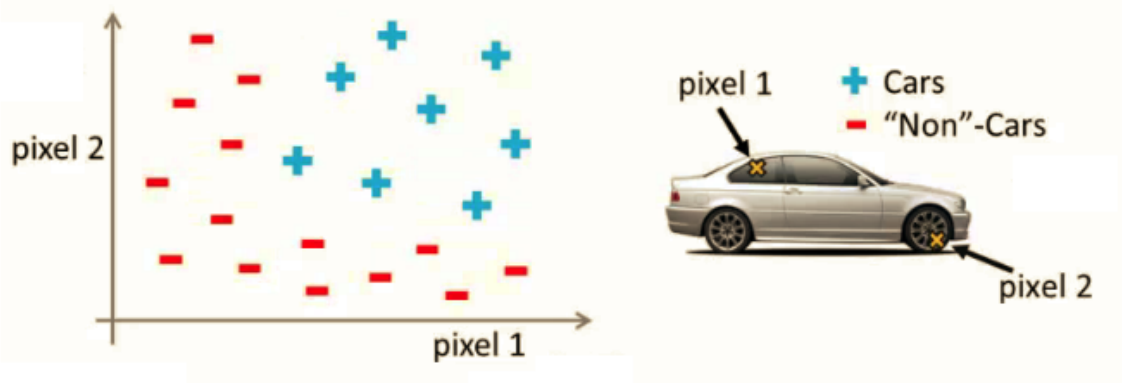


- e.g. our housing example
  - 100 house features, predict odds of a house being sold in the next 6 months
  - Here, if you included all the quadratic terms (second order)
    - There are lots of them ($x_1^2$ ,$x_1x_2$, $x_1x_4$ ..., $x_1x_{100}$)
    - For the case of n = 100, you have about 5000 features
    - 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

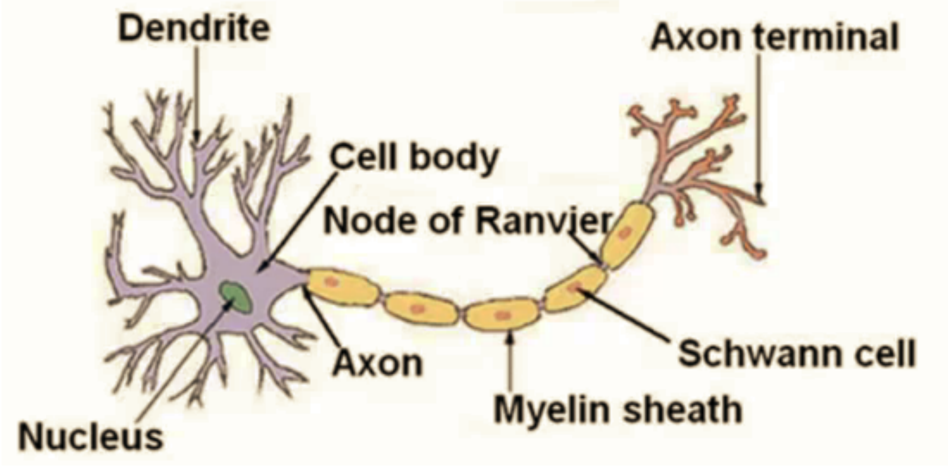
### Neurons and the brain

- **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

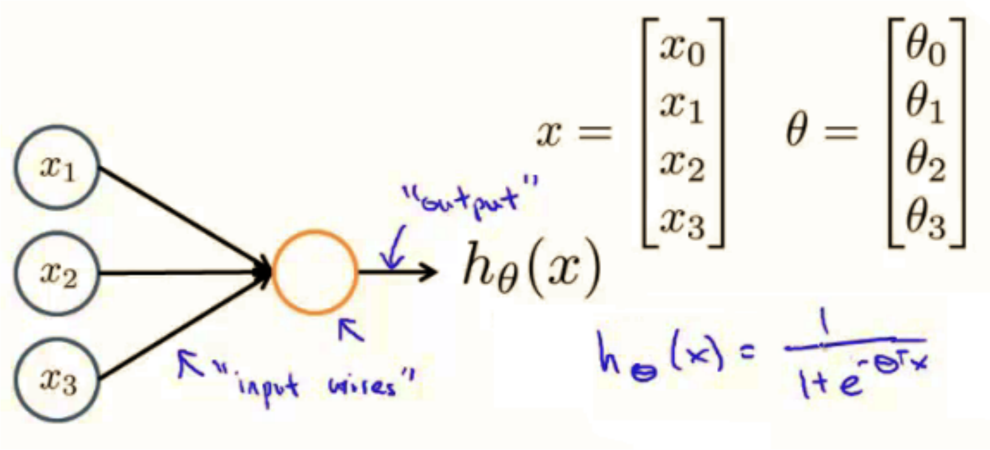


- Three things to notice
  - Cell body
  - Number of input wires (dendrites)
  - Output wire (axon)
- Simple level

  - 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

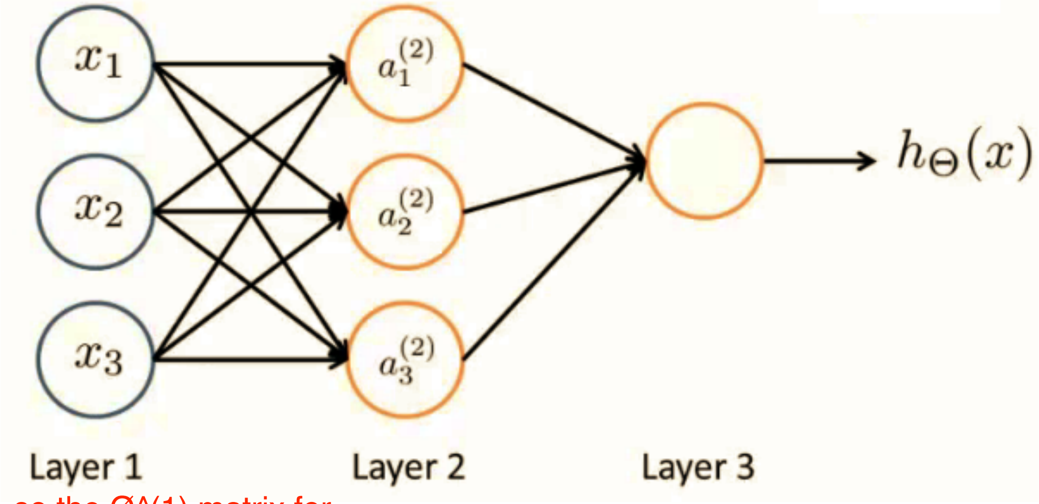
### Artificial neural network - representation of a neurone

- 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



$x_1$
$x_2$
$x_3$
"output"
$h_\theta(x)$
"input wires"
$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$
$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$
$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$

- 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



so the Ø^(1) matrix for the inputs will have 3 rows in this case

- 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

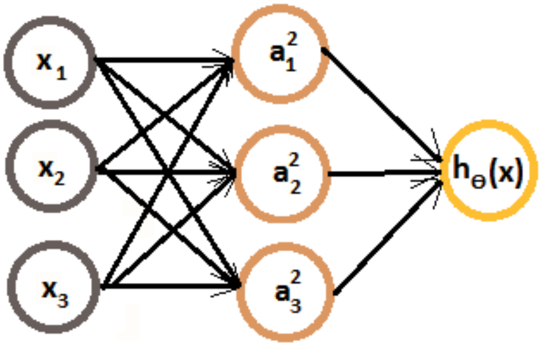
### Neural networks - notation

- **$a_i^{(j)}$ - 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 $\Theta^j$ will be of dimensions [$s_{j+1}$ 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

The +1 comes from the addition in Θ^(j) of the "bias nodes," x0 and Θ[0]^(j). In other words the output nodes will not include the bias nodes while the inputs will.

s[j]+1 weights to be applied to each of the s[j+1] units in the next layer

- 
    - 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 $\Theta^j$ 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

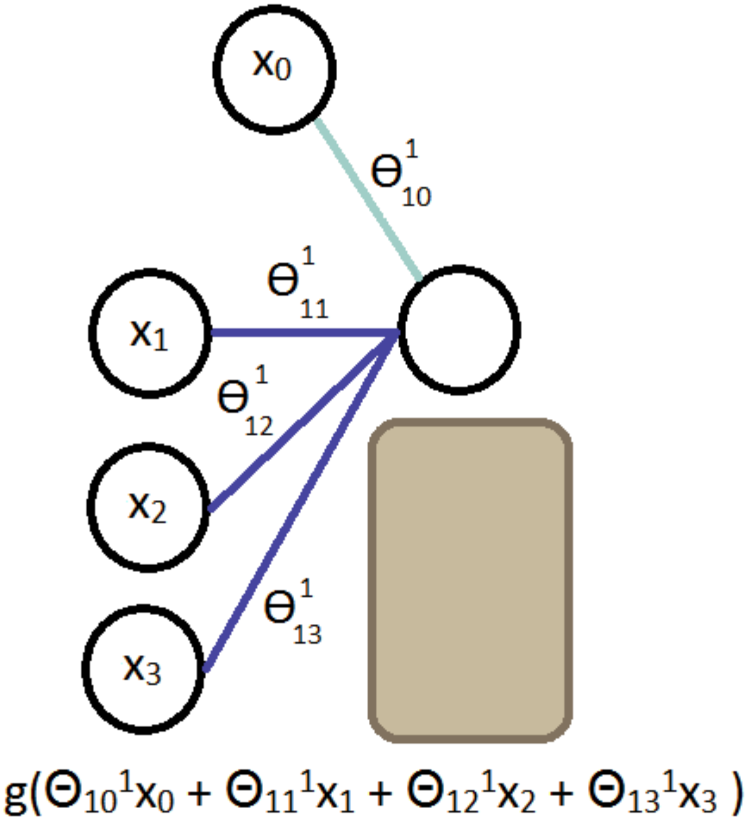


ø ^ j[af] means:
weight f (from prev. ‘outputting’ units of layer j-1) for activation unit# a in layer j+1

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

- 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)
            - **$\Theta_{ji}^l$**
                - 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



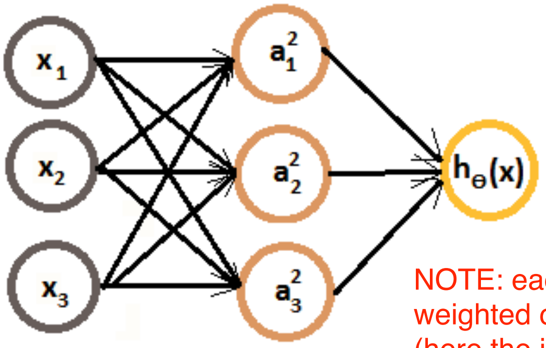
$$g(\Theta_{10}{}^1x_0 + \Theta_{11}{}^1x_1 + \Theta_{12}{}^1x_2 + \Theta_{13}{}^1x_3\ )$$

- For example
    - $\Theta_{13}{}^{1}$ = 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



NOTE: each unit in layer l forms the vector of features in to be weighted on in layer l+1.
(here the input xs are the prev. layer units to the layer 2 units)

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

- Define some additional terms
    - $z_1^2 = \Theta_{10}{}^1x_0 + \Theta_{11}{}^1x_1 + \Theta_{12}{}^1x_2 + \Theta_{13}{}^1x3$
    - 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
    - $z^2 = \Theta^{(1)}x$
        - row vector?
        - 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 vector, since z encompasses the hypoth. summations
    - $a^2$ 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;
    - $a^1 = x$ then in general, z^(j) = Θ^(j–1) * a^(j–1) OR z^(j+1)= Ø^(j) * a^(j)
        - $a^1$ 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

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

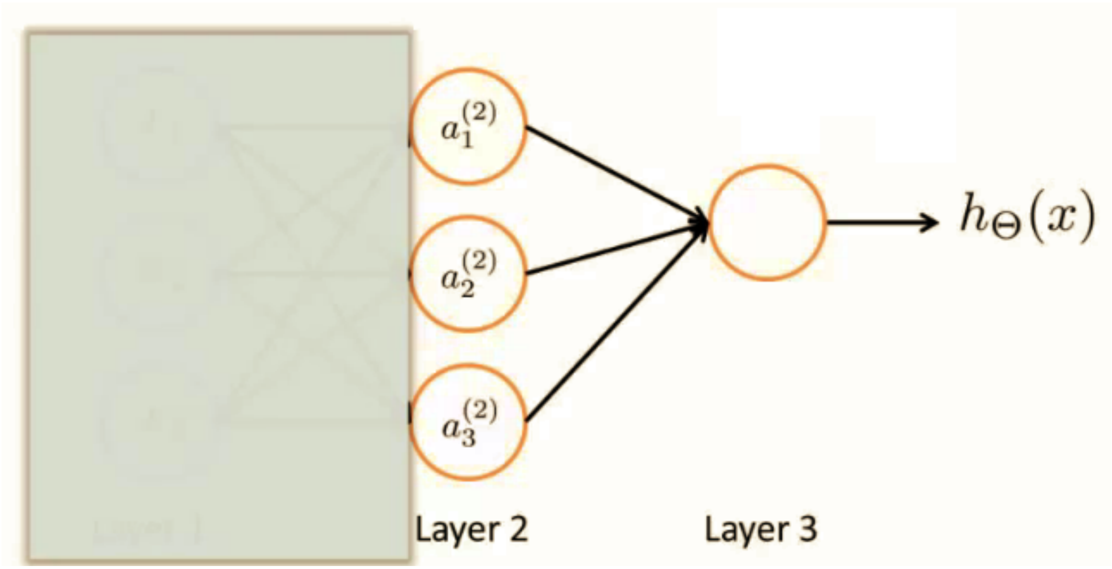
This last theta matrix Θ^(j) will have only one row which is multiplied by one column a^(j) so that our result is a single number.

- 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,
    - $z^3 = \Theta^2a^2$
        - 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

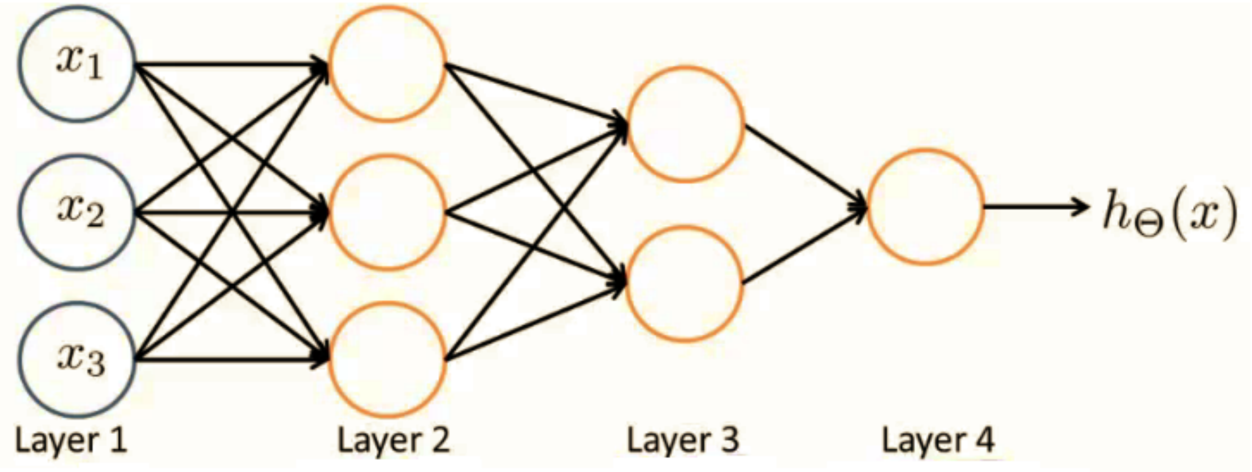
Notice that in this last step, between layer j and layer j+1, we are doing exactly the same thing as we did in logistic regression. Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses.

### Neural networks learning its own features

- Diagram below looks a lot like logistic regression



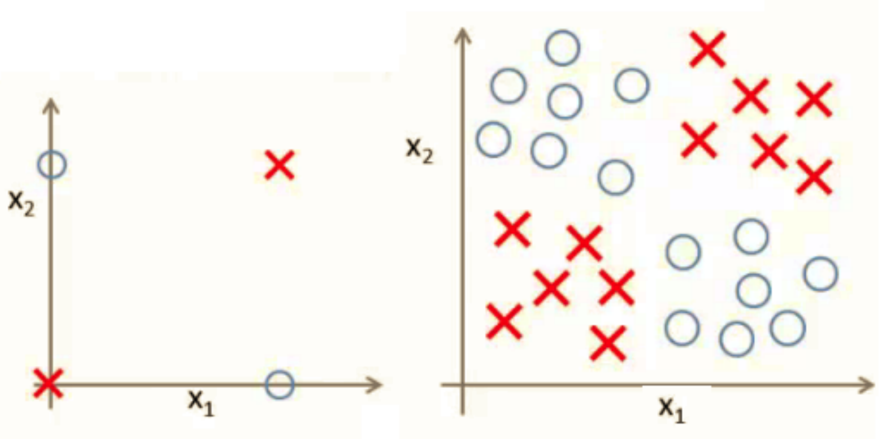
- Layer 3 is a logistic regression node
    - The hypothesis output = $g(\Theta_{10}^2 a_0^2 + \Theta_{11}^2 a_1^2 + \Theta_{12}^2 a_2^2 + \Theta_{13}^2 a_3^2)$
    - This is just logistic regression
        - The only difference is, instead of input a feature vector, the features are just values calculated 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 - $\Theta^1$
    - 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 $\Theta^1$ 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

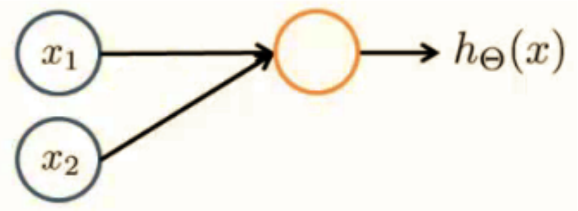
y = $x_1$ XOR $x_2$
    $x_1$ XNOR $x_2$

Where XNOR = NOT ($x_1$ XOR $x_2$)

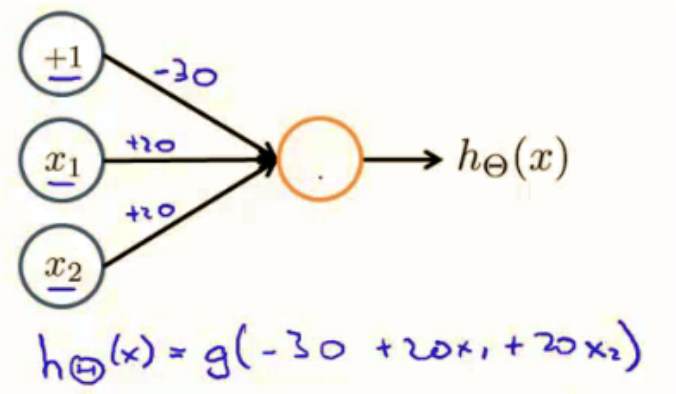
- Positive examples when both are true and both are false
  - Let's start with something a little more straight forward...
  - Don't worry about how we're determining the weights (Θ values) for now - just get a flavor of how NNs work

### Neural Network example 1: AND function

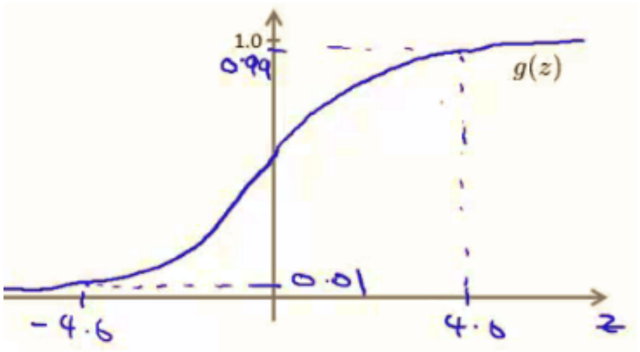
- Simple first example

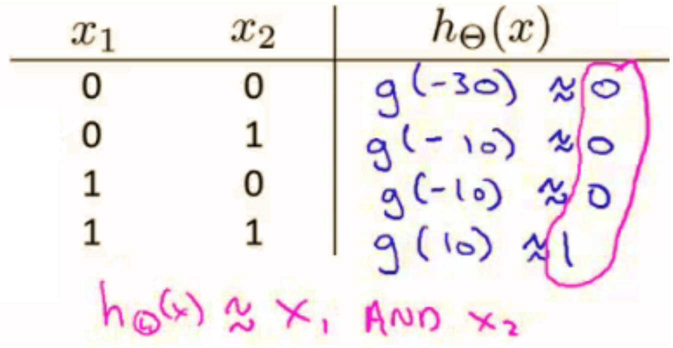


- Can we get a one-unit neural network to compute this logical AND function? (*probably*...)
  - Add a bias unit
  - Add some weights for the networks
    - What are weights?
      - Weights are the parameter values which multiply into the input nodes (i.e. Θ)



- 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 | $g(10) \approx 1$ |

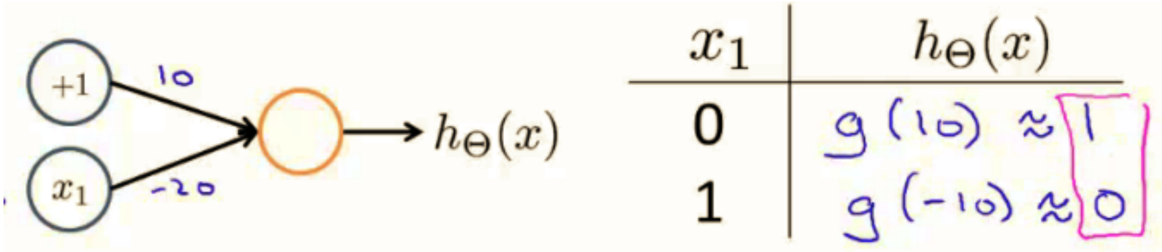
$h_\Theta(x) \approx x_1$ AND $x_2$

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?

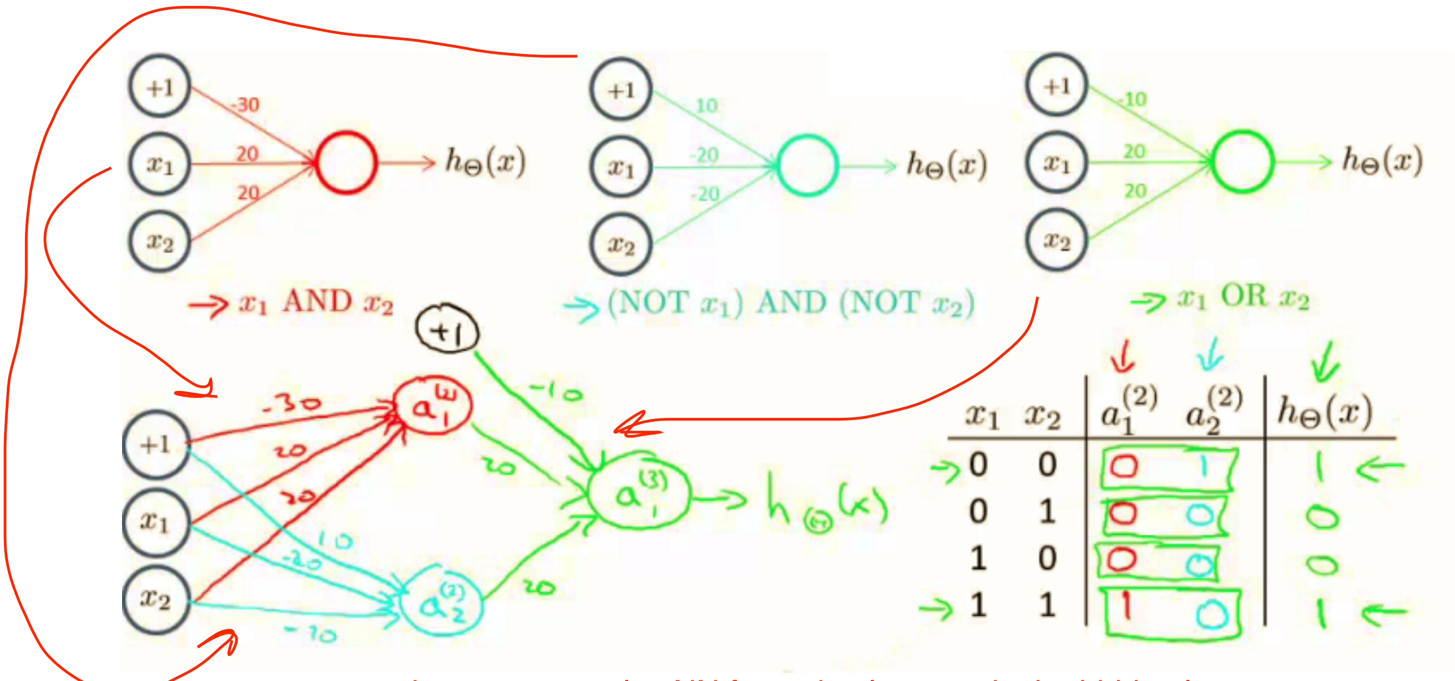


| $x_1$ | $h_\Theta(x)$ |
|---|---|
| 0 | $g(10) \approx 1$ |
| 1 | $g(-10) \approx 0$ |

- 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;



| $x_1$ | $x_2$ | $a_1^{(2)}$ | $a_2^{(2)}$ | $h_\Theta(x)$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 |

composing more complex NN from simpler ones in the hidden layer

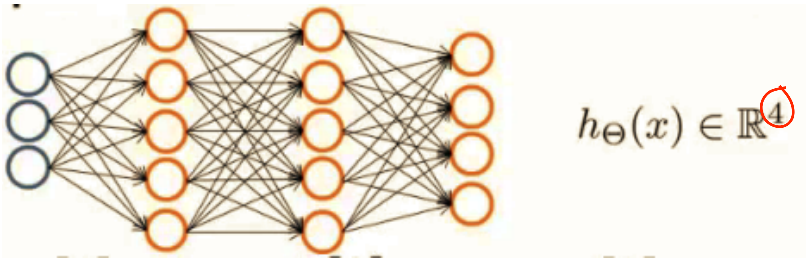
- Simplez!

**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



- 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 $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

NOTE: while the responses to inputs may be 'lone' 1s, the hypoth.(s) do not need to be
( eg. could have final activation outputs a^4[1]=0.5, a^4[2]=0.3, …, a^4[4]=0.6 and could say y = matrix such that y[j]=1 if a^4[j] is the highest value in matrix a^4
(ie. a^4[1] + … + a^4[4] does not need to equal 1) ).
That is, like in the (non-NN) logistic regression where we threshold on 0.5 to decide 1/0, the response may be discretely defined while the hypoth. output may not be.