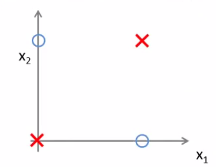
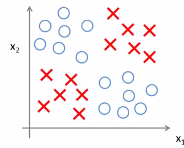
***Neural Networks – Applications***

**I. Examples and Intuitions**

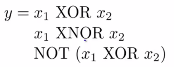
* Consider the following problem where we have features x1 + x2, both binary variables



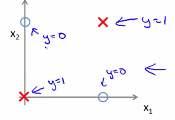
* We have 2 positive + 2 negative examples (think of this as a simplified version of a more complex learning problem w/ a bunch of positive examples in the upper right + lower left + a bunch of negative examples denoted by the circles)



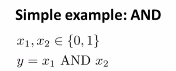
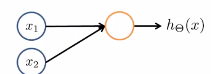
* What we'd like to do is learn a *non-linear* decisionboundary that separates the positive + negative examples.
* Concretely, this NN is really computing the *type* of label y:



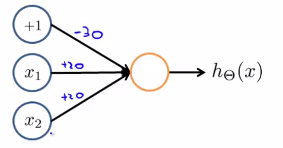
* XNOR = alternative notation for NOT x1 or x2.
* **x1 XOR x2** is True *only if EXACTLY ONE of x1 or x2 is equal to 1.*
* It turns out that these specific examples work out a little bit better if we use **XNOR** instead.
* i.e. have both positive examples = True or both = False



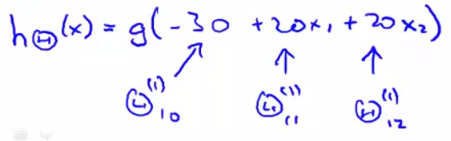
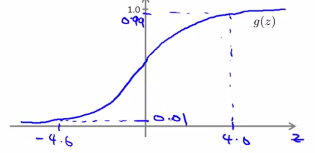
* We're going to figure out if we can get a NN to fit to this sort of training set.
* In order to build up to a NN that fits the XNOR example, we're going to start w/ a slightly simpler network that fits the AND function.

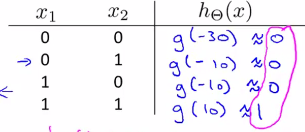
* Concretely, let's say we have binary inputs x1 + x2 that are again binaries + our target y = x1 AND x2.
* So, can we get a 1-unit NN to compute this logical AND function?
* In order to do so, draw in the bias unit as well + assign some values to the weights/parameters of this NN.



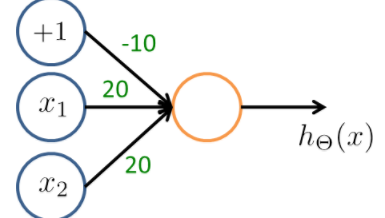
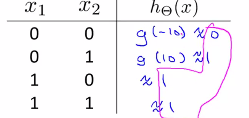
* So we’re assigning a value of -30 to the value associated w/ x0 ( = 1) + a parameter value of +20 is multiplied to x1 and x2.
* Concretely, it's the same as saying hypothesis h(x) is:

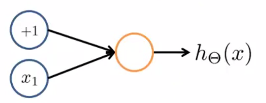
* This little single neuron network will compute the following for 4 possible input values for x1 + x2



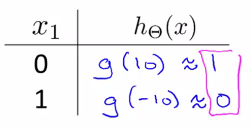
* If we look in the pink column, this is exactly the logical AND function
* This is computing h(x) which outputs 1 *if and only if* x2 + x1 are both = 1.
* So, by writing out a truth table like this, we manage to figure the logical function our NN computes.

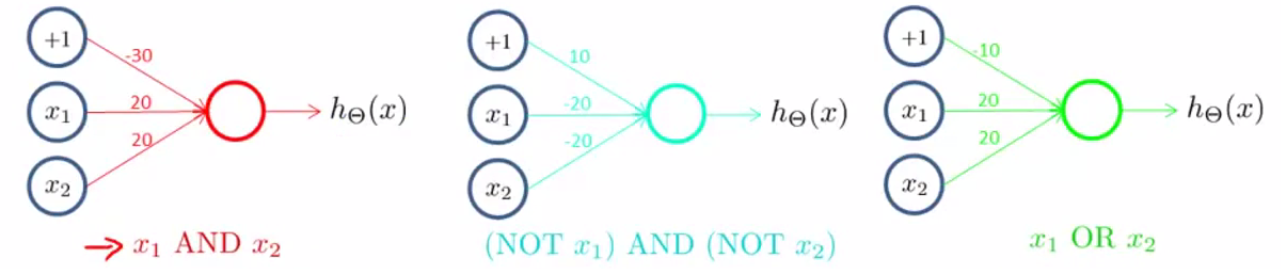
 

* This network computes the OR function.
* So single neurons in a NN can be used to compute logical functions like AND + OR, + so on.
* We can also have a NN to compute **negation** 🡪to compute the function NOT x1.



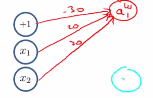
* We have only 1 input feature x1 + the bias unit +1.
* If I associate the bias unit w/ weight 10 + x1 w/ -20, then h(x) = g(10- 20x1) where g is the sigmoid function
* When x1 = 0, h(x) = g(10- 20\*0) = 10, which is approximately 1, + when x = 1, we have g(-10) which is approximately 0.



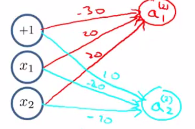
* This is essentially the NOT x1 function.
* For negations, the idea is to put a large negative weight in front of a variable you want to negate
* -20 multiplied by x1 is the general idea to negate x1
* If you want to compute a function like NOT x1 AND NOT x2, part of that will probably be putting large negative weights in front of x1 + x2, but it should be feasible to get a NN w/ just 1 output unit
* NOT x1 AND NOT x2 would be 1 if and only if x1 = x2 = 0.
* this logical function is = 1 if and only if both x1 + x2 = 0
* Now, taking the 3 pieces we have put together as the NN for computing x1 AND x2, the NN for computing NOT x1 AND NOT x2, + the NN computing for computing x1 OR x2, we should be able to compute x1 XNOR x2
* 



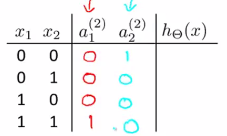
* x2 XNOR needs a non-linear decision boundary in order to separate the positive + negative examples
* 1st, take the inputs +1, x1, + x2 + create my the 1st hidden unit a1(2)



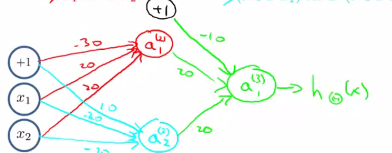
* Then create a 2nd hidden a2(2).



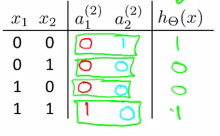
* For our truth table, for the red network, we know it was computing the x1 AND x2, and so this will be approximately 0 0 0 1, depending on the values of x1 + x2
* For a2(2), we know it was computing NOT x1 AND NOT x2 which outputs 1 0 0 0



* Finally, create the output node, a3(1), which has 1 output = h(x)



* Now fill in the truth table entries where the 1st entry is 0 OR 1 which here = 1, then 0 OR 0 which is = 0, then 0 OR 0 which = 0, and 1 OR 0 which = 1.



* Thus h(x) = 1 when either *both x1 and x2 = 0* OR when x1 + x2 both = 1
* Concretely, h(x) outputs 1 *exactly at these 2 locations + outputs 0 otherwise.*
* Thus, this NN w/ its input layer, 1 hidden layer, + one output layer, ends up w/ a nonlinear decision boundary that computes this XNOR function.



* The more general intuition is that in the input layer, we just have 4 inputs, + then we have a hidden layer which computes some slightly more complex functions of the inputs that shown here
* Then by adding yet another layer, we end up w/ an even more complex non-linear function.
* This is the intuition about why NN can compute pretty complicated functions 🡪 *W/ multiple layers we have relatively a simple function of the inputs in the 2nd layer, but a 3rd layer can build on that to compute more complex functions, + the layer after that can compute even MORE complex functions*.