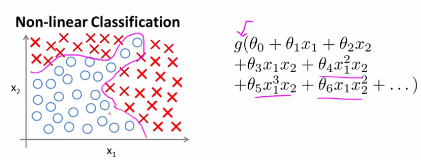
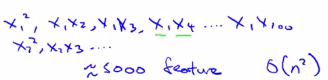
***Neural Networks***

**I. NON-LINEAR HYPOTHESIS**

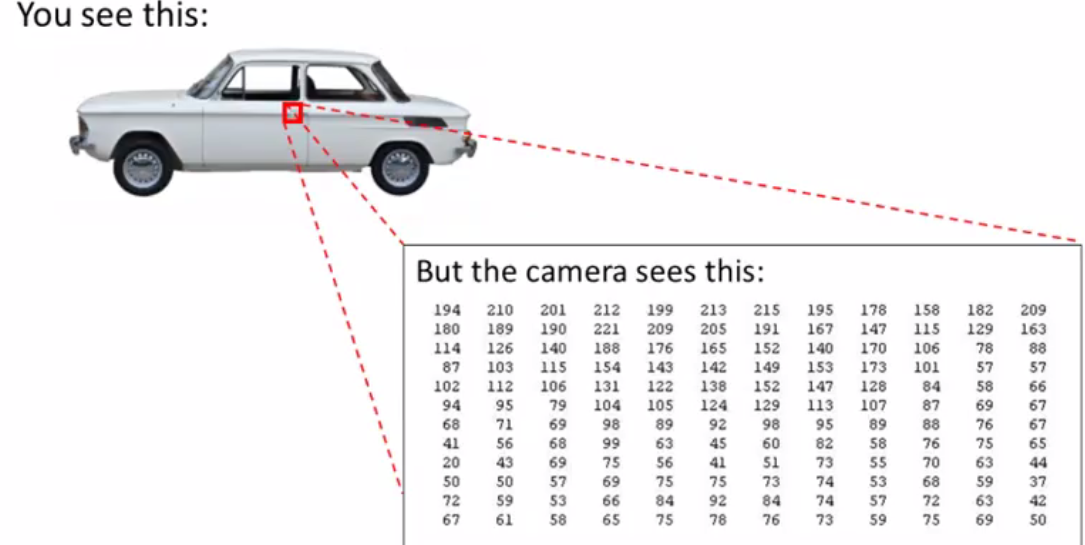
* **Neutral networks** is actually a pretty old idea, but had fallen out of favor for a while.
* But today, it is the state of the art technique for many different ML problems.
* So why do we need another learning algorithm? We already have linear regression + logistic regression
* Examples of ML problems where we need to learn complex, NON-LINEAR hypotheses.
* Supervised learning classification problem where you have a training set + you want to apply logistic regression



* Need to apply logistic regression w/ nonlinear features (where g is the sigmoid function) + lots of polynomial terms
* If you include enough polynomial terms, maybe you can get a hypotheses that separates positive + negative
* *This particular method only works well when you have, say, 2 features b/c you can then include all those polynomial terms of x1 + x2.*
* But many interesting ML problems have a lot more features than just 2.
* For housing prediction, suppose you have a housing classification problem rather than a regression problem
* You have 100 different features of a house + want to predict the odds your house will be sold w/in the next 6 months, so that will be a classification problem.
* For a problem like this, if you were to include all the quadratic (2nd-order polynomial) terms (terms that are a product of other terms starting w/ x1\*x1):

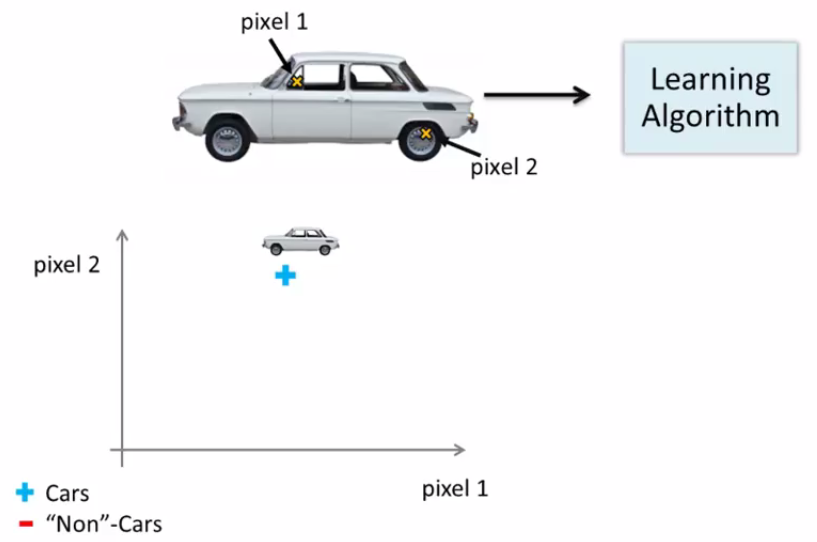
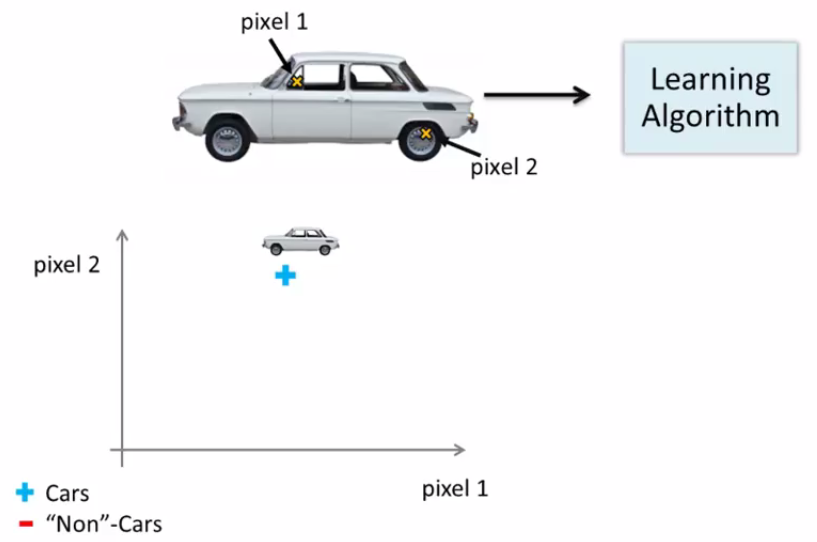


* We end up with about 5000 features.
* Asymptotically, the number of quadratic features grows roughly as **order n squared: O(n^2)** where n is the number of the original features,
* Its actually closer to n^2 / 2
* So including all the quadratic features doesn't seem like it's a good idea, b/c that’s a lot of features + you might up overfitting the training set, + it can also be computationally expensive
* 1 thing you could do is include only a subset of the quadratic features (only the squared terms for example), the number of features is much smaller 🡪 only 100 such quadratic features
* But this is not enough features to fit the data set like we have above
* In fact, if you include only these quadratic features together, you CAN actually fit very interesting hypotheses, such as axis lines of ellipses, but you certainly cannot fit a more complex data set like above.
* If you were to include the cubic, or 3rd-order polynomials of the features, we get O(n^3) 🡪 170,000 cubic features
* Including higher-order polynomial features when an original feature set is large really dramatically blows up a feature space
* Doesn't seem like a good way to come up w/ additional features w/ which to build non-linear classifiers when n is large.
* For many ML problems, n *WILL* be pretty large. Here's an example.
* Consider the problem of **computer vision** + you want to use ML to train a classifier to examine an image + tell us if the image is a car.
* To understand why CPU vision, where you + I see a car, the CPU sees a matrix/grid of pixel intensity values that signifies the brightness of each pixel in the image.
* So CPU vision problems to look at this matrix of pixel intensity values + tell us if these numbers represent, say, the door handle of a car.

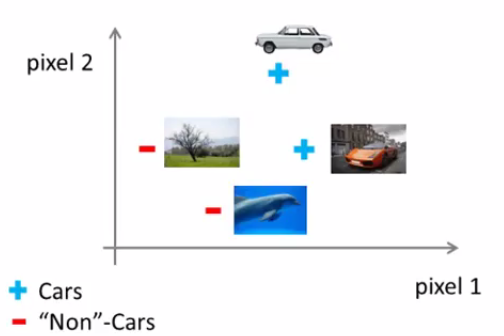


* Concretely, when we use ML to build a car detector, we come up w/ a *labelled* training set w/ examples of cars + examples of things that are not cars
* Then we give our training set to the learning algorithm, train, a classifier, and test it by showing a new image + asking "What is this new thing?" + hopefully it will recognize that that is a car.

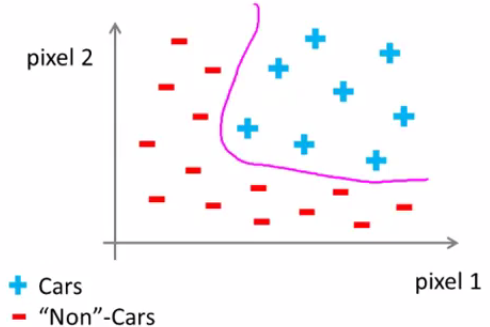
* To understand why we need **nonlinear hypotheses**, take a look at some of the images of cars + maybe non-cars we might feed to our learning algorithm.
* Pick a couple of pixel locations in an images + plot this car image at a certain point, depending on the intensities of pixel 1 and pixel 2:

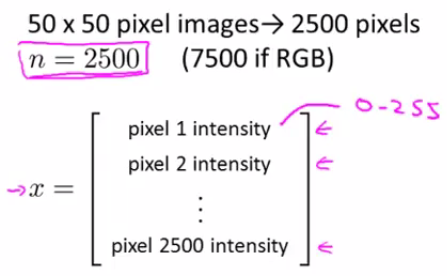
* Do this w/ a few other images: a different example of a car +some non-cars

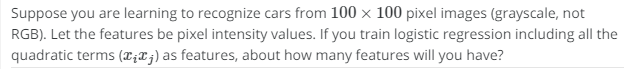


* If we do this for more + more examples, using pluses to denote cars + minuses to denote non-cars, we'll find that cars + non-cars end up lying in different regions of the space
* What we need therefore is some sort of non-linear hypotheses to try to separate out the 2 classes.



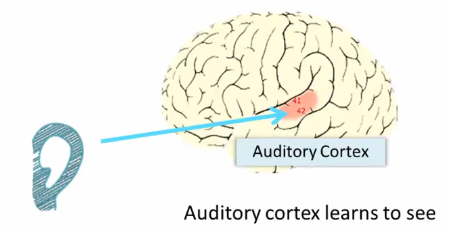
* Suppose we were to use just 50x50 pixel images (pretty small), we’d have 2500 pixels
* The dimension of our feature space will be n = 2500 where our feature vector x is a list of all the pixel intensities (brightness of pixel 1, brightness of pixel 2, down to the brightness of the last pixel)



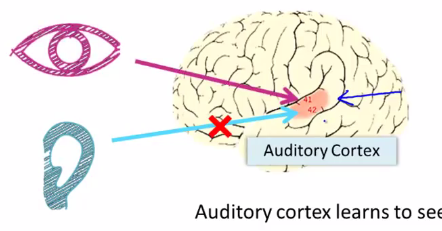
* In a typical CPU representation, each of these may be values between 0-255, if it gives us the grayscale value.
* If using RGB images (separate red, green + blue values), we would have n = 7500.
* If we were to try to learn a nonlinear hypothesis by including all quadratic features like before, w/ 2500 pixels, we’d end up w/ a total of 3 million features (2500^2 / 2)
* So, simple logistic regression together w/ adding in quadratic or cubic features is just not a good way to learn complex nonlinear hypotheses when n is large, b/c you just end up w/ too many features.
* 
* **50 million (5\*10^7)**

**II. NEURONS AND THE BRAIN**

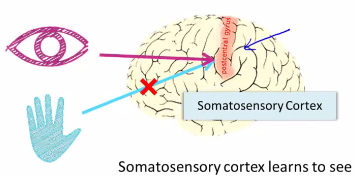
* **Neural Networks** are a pretty old algorithm originally motivated by the goal of having machines can mimic the brain + they work really well for different ML problems
* The origins of Neural Networks were algorithms that try to mimic the brain w/ the idea that if we want to build learning systems, why not mimic the most amazing learning machine we know about?
* NN’s came to be very widely used throughout the 1980's + 1990's
* For various reasons, popularity diminished in the late 90's, but more recently, NN’s have had a major recent resurgence.
* 1 reason for this resurgence is that NN’s are somewhat more computationally expensive algorithms
* It was only somewhat more recent that CPU’s became fast enough to really run large scale NN’s
* B/c of that as well as a few other technical reasons, modern NNs today are the state of the art technique for many applications.
* When you think about mimicking the brain, the human brain does many amazing things, right?
* It can learn to see process images, to hear, to process our sense of touch, to do calculus
* It does so many different + amazing things, so it seems like if you want to mimic the brain you have to write lots of different pieces of software to mimic all of these different fascinating things the brain does for us
* But there’s this fascinating hypothesis that the way the brain does all these different things is not w/ a thousand *different* programs, but instead, the brain does w/ *just a single learning algorithm*.
* This is just a hypothesis but here’s some evidence for this:



* The ear takes sound signals + routes them to the auditory cortex to allow you to understand them
* Neuroscientists have done experiments where you cut the wire from the ears to the auditory cortex + re-wire a brain so that the signal from the eyes to the optic nerve eventually gets routed to the auditory cortex.



* *If you do this it turns out, the auditory cortex will learn to see* (in every sense of the word “see” as we know it)
* So, if you do this to the animals, the animals can perform visual discrimination tasks + as they look at images, make appropriate decisions based on the images w/ that piece of brain tissue.



* Here's another example.
* The somatosensory cortex is how you process your sense of touch.
* If you do a similar re-wiring process to the one above, then the somatosensory cortex will learn to see.
* B/c of this and other similar experiments, these are called **neuro-rewiring experiments**.
* There's this sense that if the same piece of physical brain tissue can process sight OR sound OR touch, then maybe there is 1 learning algorithm that can process sight or sound or touch.
* Then, instead of needing to implement a thousand different programs/algorithms to do the thousand things the brain does, maybe what we need is to figure out some approximation to whatever the brain's learning algorithm is
* We’d implement that + the brain would learn how to process different types of data by itself

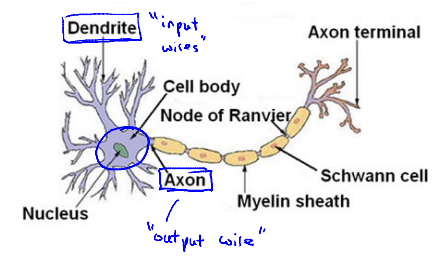
* To a surprisingly large extent, it seems as if we can plug in almost any sense to almost any part of the brain +, w/in reason, the brain will learn to deal with it.
* Here are a few more examples:



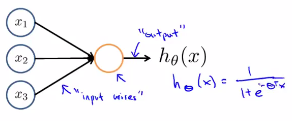
* On the upper left is an example of learning to see with your tongue.
* This is actually a system called BrainPort that’s undergoing FDA trials to help blind people see
* The way it works is, you strap a camera to your forehead, facing forward, that takes a low-resolution grayscale image of what's in front of you
* You then run a wire to an array of electrodes at you place on your tongue, such that each pixel gets mapped to a location on your tongue where maybe a high voltage corresponds to a dark pixel and a low voltage corresponds to a bright pixel
* W/ this sort of system, you + I will be able to learn to see w/ out tongues in 10’s of minutes
* 2nd example = human echo location or human sonar.
* There’s are 2 ways you can do this 🡪 either snap your fingers or click your tongue.
* There are blind people today that are actually being trained in schools to do this + learn to interpret the pattern of sounds bouncing off the environment (sonar).
* There is a kid who tragically had his eyeballs removed due to cancer, but by snapping his fingers, he can walk around + never hit anything, ride a skateboard, + shoot a basketball
* 3rd example: The Haptic Belt
* Have a strap around your waist w/ a ring of buzzers + always have the northern-most one buzzing to give a human a sense of direction similar to how birds can sense where north is
* 4th example: if you plug a 3rd eye into a frog, the frog will learn to use that eye as well.
* It's pretty amazing as to what extent you can plug in almost any sensor to the brain + the brain's learning algorithm will just figure out how to learn from + deal w/ that data.
* There's a sense that if we can figure out what the brain's learning algorithm is + implement some approximation of that algorithm on a CPU, maybe that would be our best shot at making real progress towards building truly intelligent machines + AI

**III. MODEL REPRESENTATION**

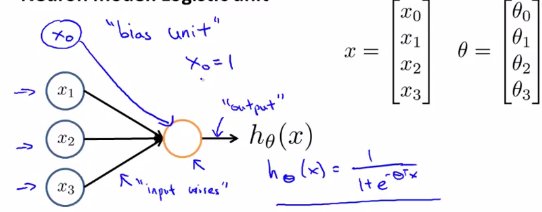
* NNs were developed as simulations of neurons/networks of neurons in the brain.
* To explain the hypothesis, let's start by looking at what a single neuron in the brain looks like.



* A brain packed full of neurons (cells in the brain)
* The neuron has a **cell body** containing the nucleus + a number of input wires = the **dendrites,** which receive inputs from other locations.
* Neurons also have an output wire called an **Axon**, used to send signals to other neurons
* So, at a simplistic level, a neuron is a computational unit that gets a number of inputs through input wires, does some computation, + sends outputs via its axon to other nodes/neurons in the brain.
* Neurons communicate w/ each other via little pulses of electricity (spikes)
* If a neuron wants a send a message, it sends a spike to another neuron via its axon that is connected to the dendrites of the 2nd neuron, which accepts this incoming message, does some computation, + in turn, decides to send out a message from its axon to more neurons
* This is the process by which all human thought happens, neurons doing computations + passing messages to other neurons as a result of inputs they've gotten.
* This is how our senses + muscles work as well.
* If you want to move a muscle, your neuron sends a spike to your muscle to cause it to contract
* Some senses like sight must have the eyes send a message via pulses of electricity to a neuron in the brain.
* In an artificial NN that we've implemented on a CPU, we're going to use a very simple model of what a neuron does 🡪 *going to model a neuron as just a logistic unit*.



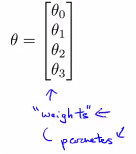
* Imagine the yellow circle as playing a role analogous to the body of a neuron
* We then feed the neuron a few inputs via its “dendrites”, the ANN does some computation, + outputs some value on its output wire
* This is a very simple, maybe a vastly oversimplified model of the computations a neuron does
* An x0 node is sometimes called the **bias unit** or the **bias neuron**, but b/c x0 is already = 1, we don’t *need* to draw it (depends on whatever is more notational-ly convenient for that example)



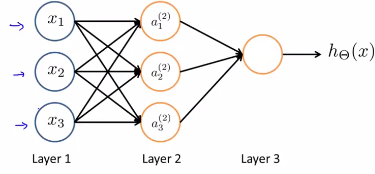
* When we talk about NNs, sometimes we'll say “this is an artificial neuron w/ a **Sigmoid/logistic activation function”.**
* This activation function in NN terminology is just another term for the function for the non-linearity



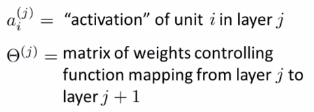
* Whereas so far I've been calling ϴ the “parameters of the model”,
* But in the NN literature, sometimes you might hear people talk about **weights** of a model, + weights just means *exactly the same thing as parameters of a model*.



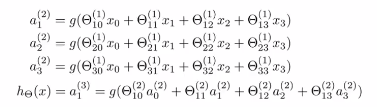
* A neural *NETWORK* is a *group* of these neurons strung together.



* We have input units x1-x3 + 3 neurons a1(2), a2(2), a3(2)
* Can add in just x0 and a0 for our bias units that always output a value of 1
* Finally we have a 3rd node, the final layer, which outputs the value that hϴ(x) computes.
* In a NN, the **first layer/input layer** is where we Input our features + the **final layer/output layer** has a neuron that outputs the final value computed by a hypothesis.
* Layer 2 in between is called the **hidden layer**
* This isn't great terminology, but the idea is that in supervised learning, you get to see the inputs + see the correct outputs, whereas in a hidden layer, there’s values you don't get to observe in the training set.
* It's not x + it's not y, so we call those hidden.
* Basically, anything that isn't an input layer and isn't an output layer is called a hidden layer.
* To explain these specific computations represented by a NN, here's a little bit more notation.

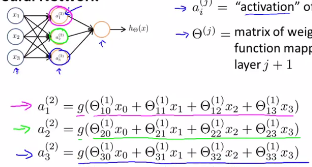


* a1(2) = activation of the 1st unit in layer 2, our hidden layer.
* **Activation** = the value computed + outputted by a specific neuron
* In addition, an NN is parametrized by these matrixes, ϴ(j) = a matrix of weights controlling the function mapping from 1 layer to the next
* Here are the computations that are represented by our NN diagram:



* The 1st hidden unit has its value computed as follows: a1(2) = the logistic/sigmoid activation function **g** applied to a sort of linear combination of inputs.

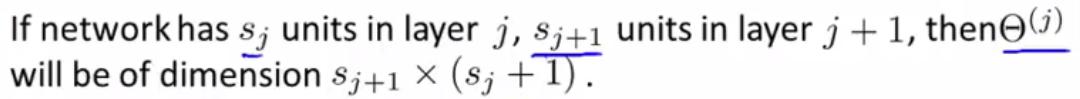




* So we have 3 input units/neurons + 3 hidden units, so the dimension of ϴ1 (matrix of parameters governing the mapping from our 3 input units to our 3 hidden units) will be 3x4



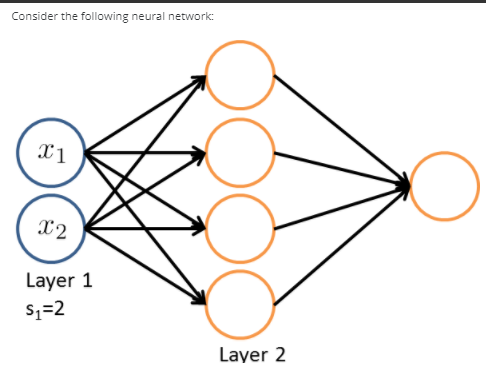
* More generally for the matrix ϴj (which governs the function mapping from layer j to layer (j + 1):



* So we talked about what the three hidden units do to compute their values.
* Finally, in the output layer, we have 1 more unit which computer hϴ(x)



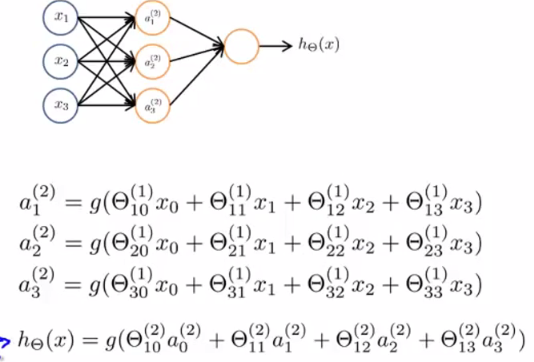
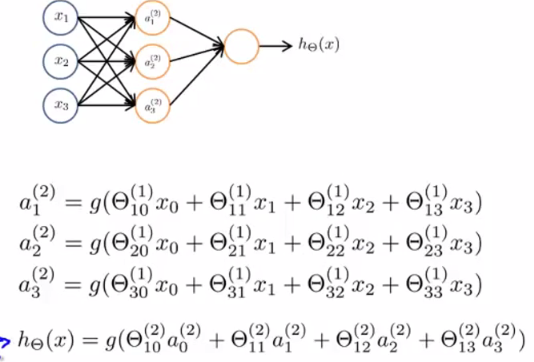
* Notice we have ϴ w/ a superscript 2 here, b/c ϴ2 = the matrix of parameters/weights that controls the function that maps from the hidden units (layer 2 units) to the lone layer 3 unit
* To summarize, what we've done is shown an artificial NN which defines a function hϴ(x) that hopefully maps from x's input values to some provisions y.
* These hypothesis are parameterized by parameters denoted w/ a capital ϴ so that as we vary ϴ, we get different hypothesis + we get different functions mapping from x to y.

  🡺 **4x3**

* So this gives us a mathematical definition of how to represent the hypothesis in the NN.

**IV. MODEL REPRESENTATION II**

* So we have the sequence of steps we need in order to compute the output of a hypotheses:



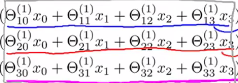
* We compute the **activation values** of the 3 hidden units + use those to compute the final output of our hϴ(x)



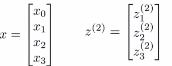
* We can define the underlined term here as z1(2) to end up w/:



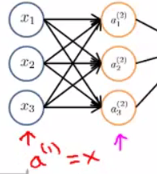
* Superscript 2 means these are values associated w/ layer 2
* So we’d then have z2(2) and z3(2) for the next 2 activation values
* These z values are just a weighted linear combination of the input values x0-x3 that go into a particular neuron.



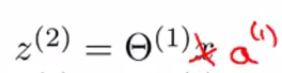
* If you look at this block of numbers, you may notice that it corresponds suspiciously similar to the matrix vector multiplication of x1 \* x.
* Using this observation we're able to vectorize this computation of the NN.
* Concretely, define the feature vector x as usual to be the vector of x0, x1, x2, x3 where x0 = always 1 + z(2) is the vector of these z-values z(2)1 z(2)2, z(2)3



* Notice z2 this is a 3-dimensional vector.
* Now vectorize the computation of a(2)1, a(2)2, a(2)3 as follows in 2 steps
*  
* 1: Compute z2 as ϴ1\*x to give us
* 2: a2 = g(z2) where z2 is the 3-dimensional vector + a2 is also a 3 -dimensional vector
* Thus this activation **g** applies the sigmoid function element-wise to each of the z2's elements.
* To make our notation a little more consistent w/ what we'll do later, in the input layer we have the inputs x, but we can also think of this as the activations of the 1st layer
* i.e. Define a1 to be equal to x.



* Can now replace x in z2



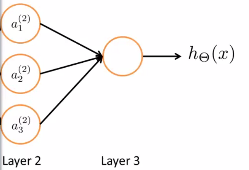
* Now, w/ what I've written so far, I've got myself values for a(2)1, a(2)2, a(2)3, but I need 1 more value, this a(2)0 that corresponds to the bias unit in the hidden layer
* Take care of this extra bias unit, add an extra a(2)0 (which is = 1), which results in a2 being a 4-dimensional feature vector, b/c we just added this extra a0



* Finally, to compute the actual value output of our hypotheses, we then simply need to compute z3.
* z3 = to this inner term:



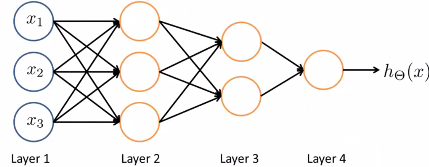
* Then finally my hϴ(x) output is a3 = the activation of my 1 + only unit in the output layer
* You can write it as a3 or as a(1)3 = g(z3)
* This process of computing hϴ(x) is also called **FORWARD PROPAGATION** = we start off w/ the activations of the input units + then forward-propagate that to the hidden layer + compute its activations + then forward propagate them + compute the activation of the output layer
* This process of computing the activations from the input, then hidden, then output layer, is our **forward propagation**
* What we just did is worked out a vector-wise implementation of this procedure.
* If you implement it using these new equations, we’d have an efficient way of computing hϴ(x).
* This forward propagation view also helps us to understand what NNs might be doing + why they might help us to learn interesting nonlinear hypotheses.
* Consider the following NN w/ a covered up left path for now



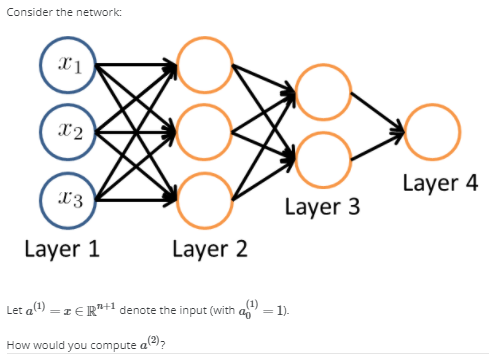
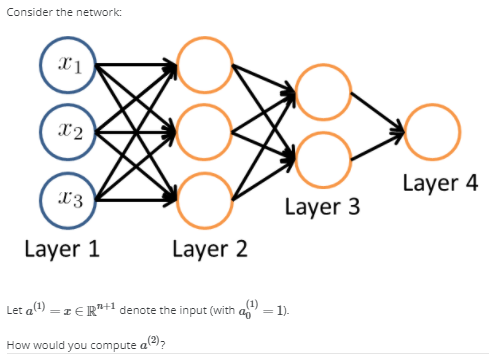
* What's left in looks a lot like logistic regression, where what we're doing is we're using a final node (layer 3) as the logistic regression unit to make a prediction h(ϴ)x
* Concretely, what h(ϴ)x is outputting is my sigmoid activation function g of ϴ0\*a0 + ϴ1\*a1 + ϴ2\*a2 + ϴ3\*a3
* Where values a1-a3 are those given by the 3 hidden units.
* To be actually consistent to earlier, notation, we need to fill in superscript 2's everywhere + have these indices = 1 b/c we have only 1 output unit

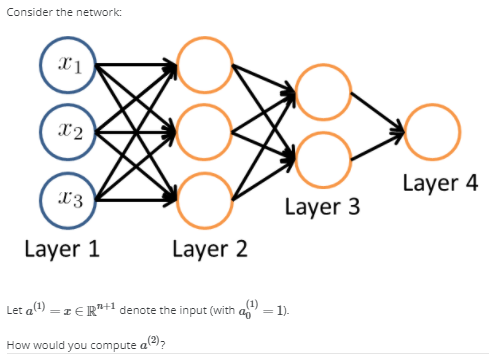


* If you focus on the blue parts of the notation, it looks awfully like a standard logistic regression model, except we now have a capital ϴ instead of lower case ϴ.
* This is just doing logistic regression but w/ features fed into it that are the values computed by the hidden layer.
* So, what this NN is doing is just like logistic regression, except that rather than using the *original features* x1-x3, it’s using *new features* a1-a3
* The cool thing about this is that the features a1-a3 themselves are *learned as functions of the input*s.
* Concretely, the function mapping from layer 1 to layer 2 is determined by some *OTHER set* of parameters, ϴ1.
* So it's as if the NN, instead of being constrained to feed the features x1-x3 to logistic regression, it gets to learn its *own* features a1-a3 to feed into the logistic regression
* As you can imagine, depending on what parameters it chooses for ϴ1, you can learn some pretty interesting + complex features and therefore end up w/ a better hypotheses than if you were constrained to use the raw features x1-x3 or constrained to choose polynomial terms
* Instead, this algorithm has the flexibility to try to learn whatever features it wants using a1-a3 in order to feed values into this last unit in essentially a logistic regression
* You can have NNs w/ other types of diagrams as well
* The way NNs are connected is called the **architecture** = refers to how different neurons are connected to each other



* This is an example of a different NN architecture
* In the 2nd layer, we have 3 hidden units computing some complex function from the input layer, + then the 3rd layer takes the 2nd layer's features + compute some more complex features so that by the time you get to the layer 4/output layer, you can have even more complex features than you were able to compute in layer 3 + end up w/ some very interesting nonlinear hypotheses.
* This network has two hidden layers b/c anything that's not an input or an output layer is hidden.
* **Feed forward propagation step** in a NN 🡪 start from the activations of the input layer + forward propagate that to the 1st hidden layer, then the 2nd hidden layer, + so on to the output layer.





* 