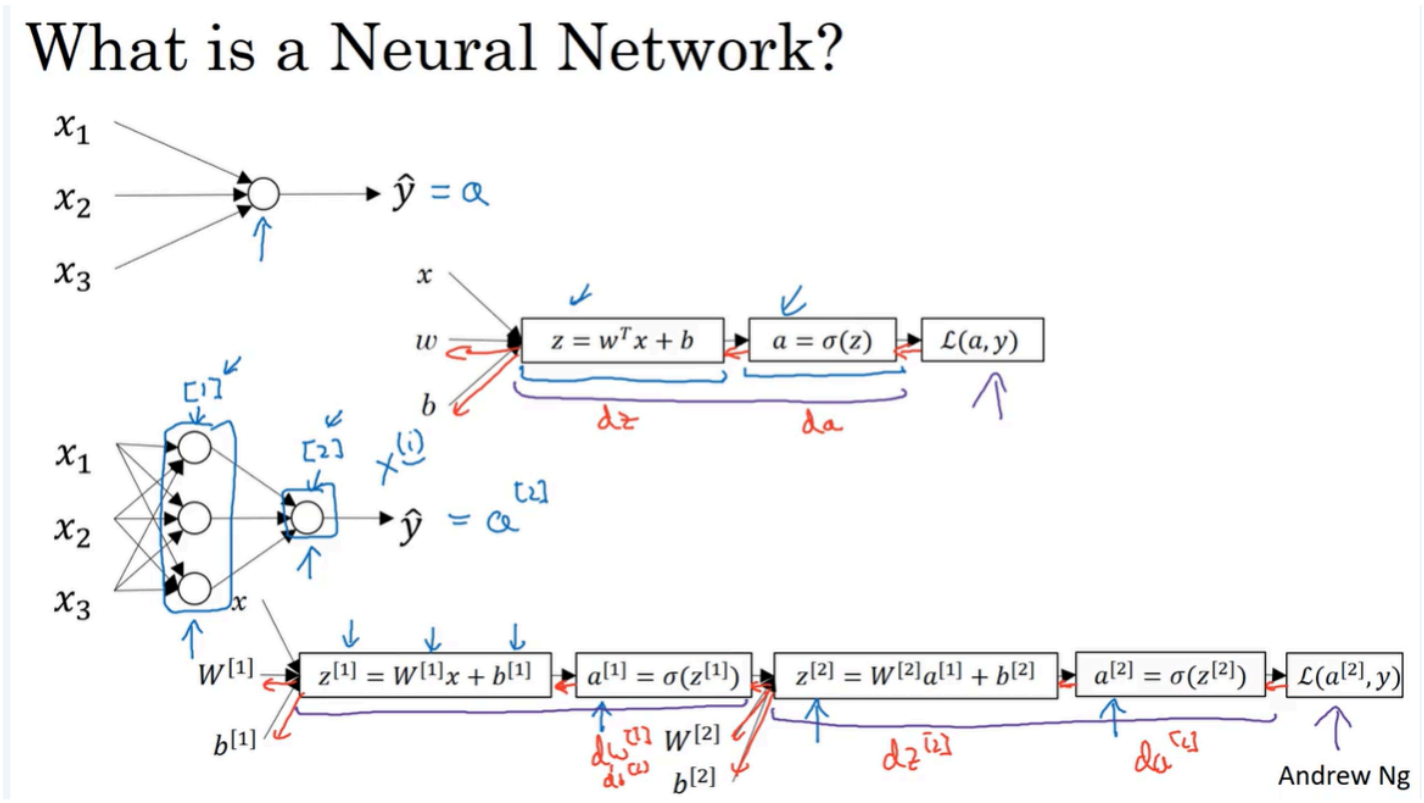
Shallow Neural Network

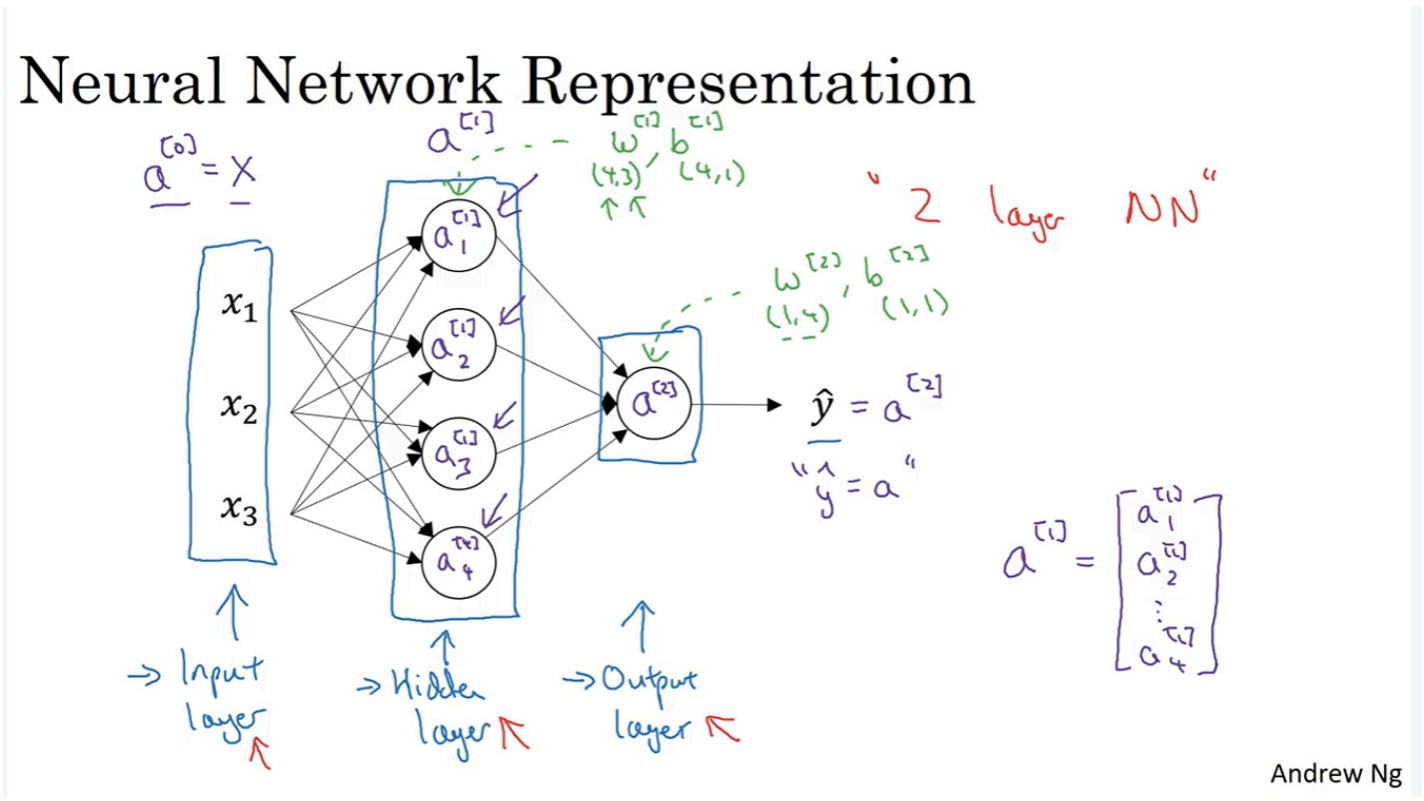
Neural Networks Overview



So, new notation that we'll introduce is that we'll use superscript square bracket one to refer to quantities associated with this stack of nodes, it's called a layer ($b^{[1]}$ represents b in layer 1) . Then later, we'll use superscript square bracket two to refer to quantities associated with that node. That's called another layer of the neural network.

The superscript square brackets, like we have here, are not to be confused with the superscript round brackets which we use to refer to individual training examples $x^{(i)}$ () training example i.

Neural Network Representation



One funny thing about notational conventions in neural networks is that this network that you've seen here is called a two layer neural network.

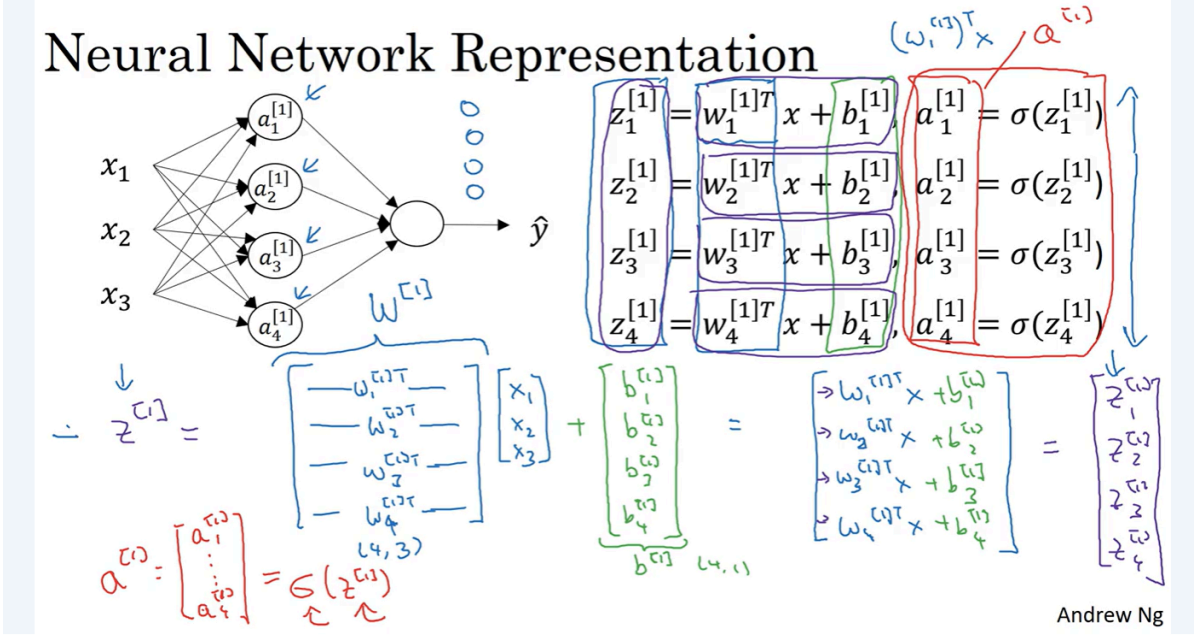
And the reason is that when we count layers in neural networks, we don't count the input layer. So the hidden layer is layer one and the output layer is layer two. In our notational convention, we're calling the input layer layer zero, so technically maybe there are three layers in this neural network. Because there's the input layer, the hidden layer, and the output layer.

But in conventional usage, if you read research papers and elsewhere in the course, you see people refer to this particular neural network as a two layer neural network, because we don't count the input layer as an official layer.

Finally, something that we'll get to later is that the hidden layer and the output layers will have parameters associated with them.

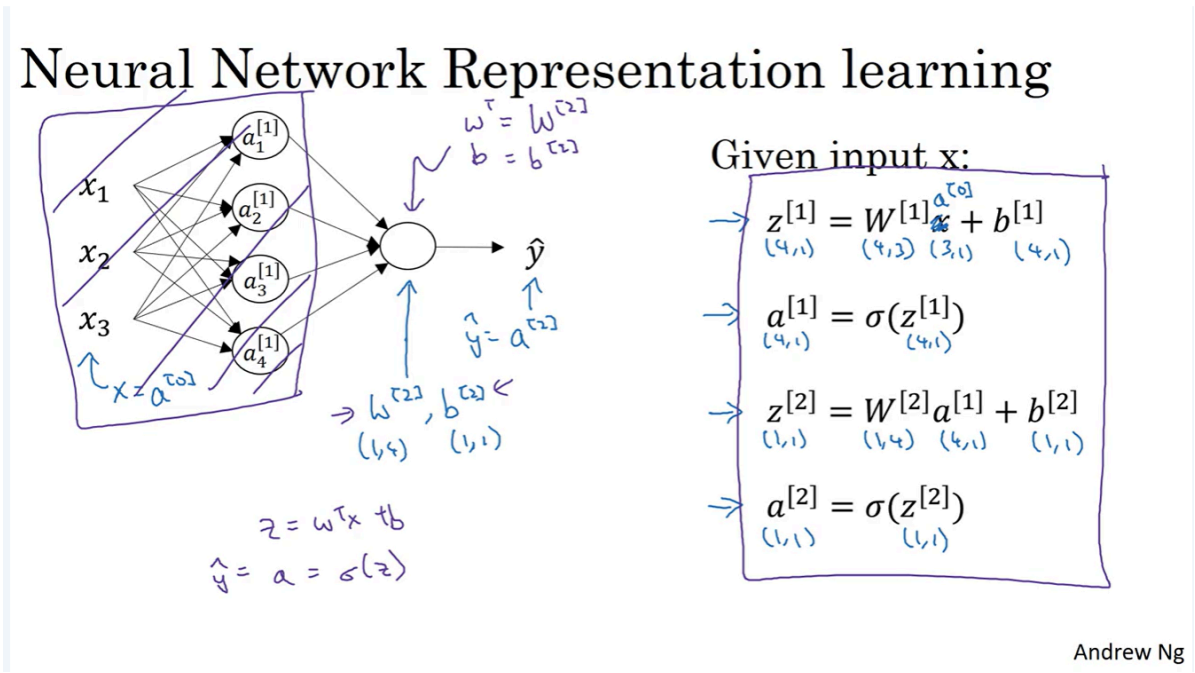
So the hidden layer will have associated with it parameters w and b. And I'm going to write superscripts square bracket 1 to indicate that these are parameters associated with layer one with the hidden layer. We'll see later that w will be a (4 × 3) matrix and b will be a 4 by 1 vector in this example. Where the first coordinate four comes from the fact that we have four nodes of our hidden units and a layer, and three comes from the fact that we have three input features. We'll talk later about the dimensions of these matrices. And it might make more sense at that time. But in some of the output layers has associated with it also, $w^{[2]}$ and $b^{[2]}$. And it turns out the dimensions of these are (1 × 4) for w and (1 × 1) for b. And these 1 x 4 is because the hidden layer has four hidden units, the output layer has just one unit.

Computing a Neural Network's Output

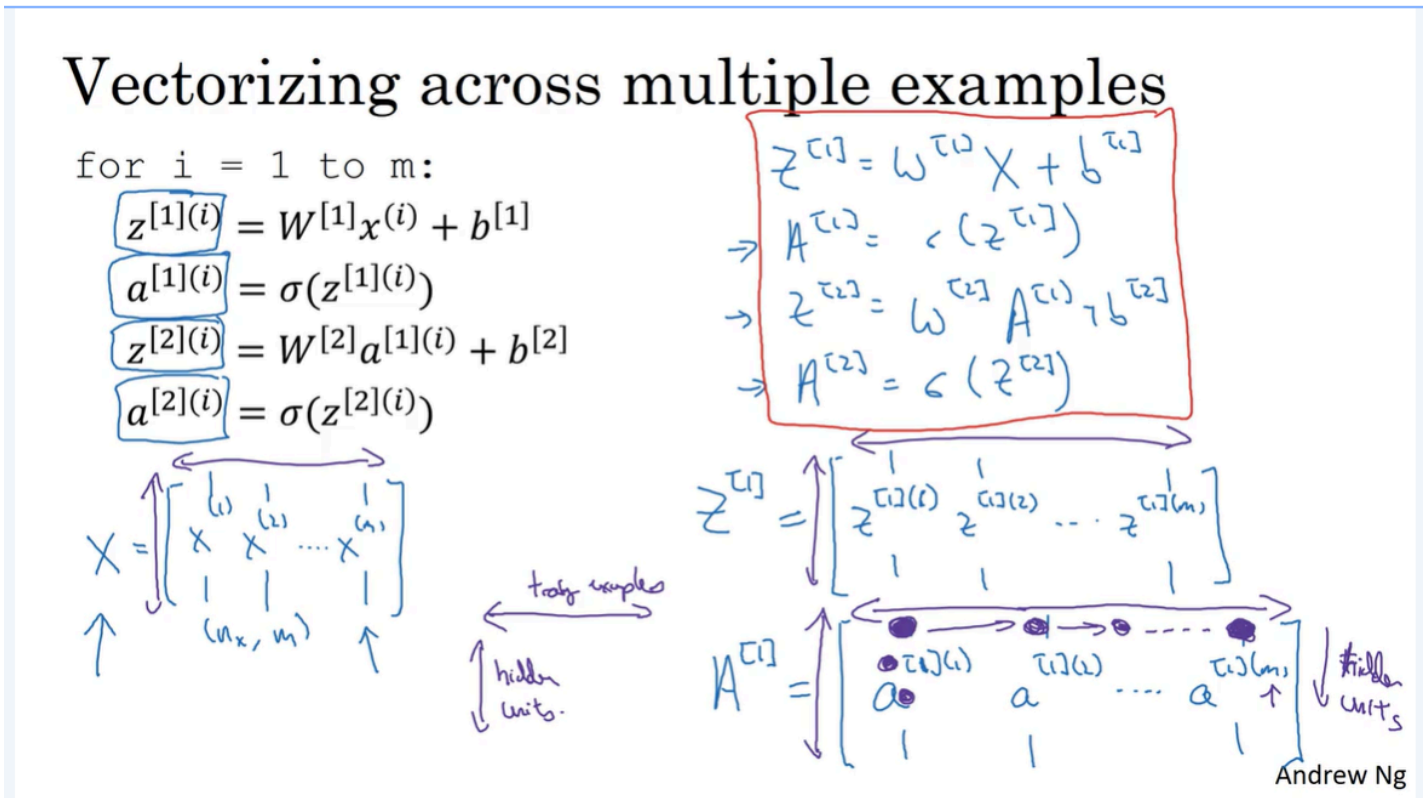


Report an issue

It’s like the feedforward method

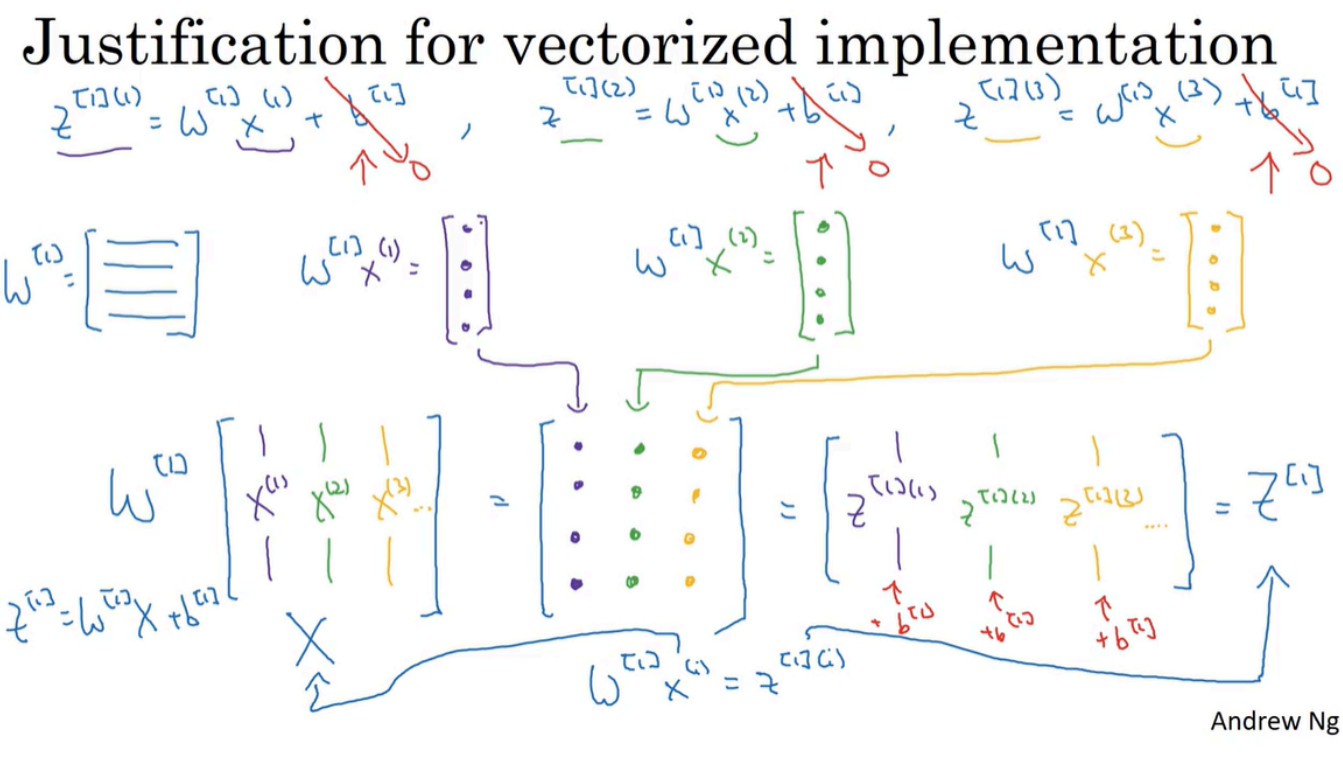


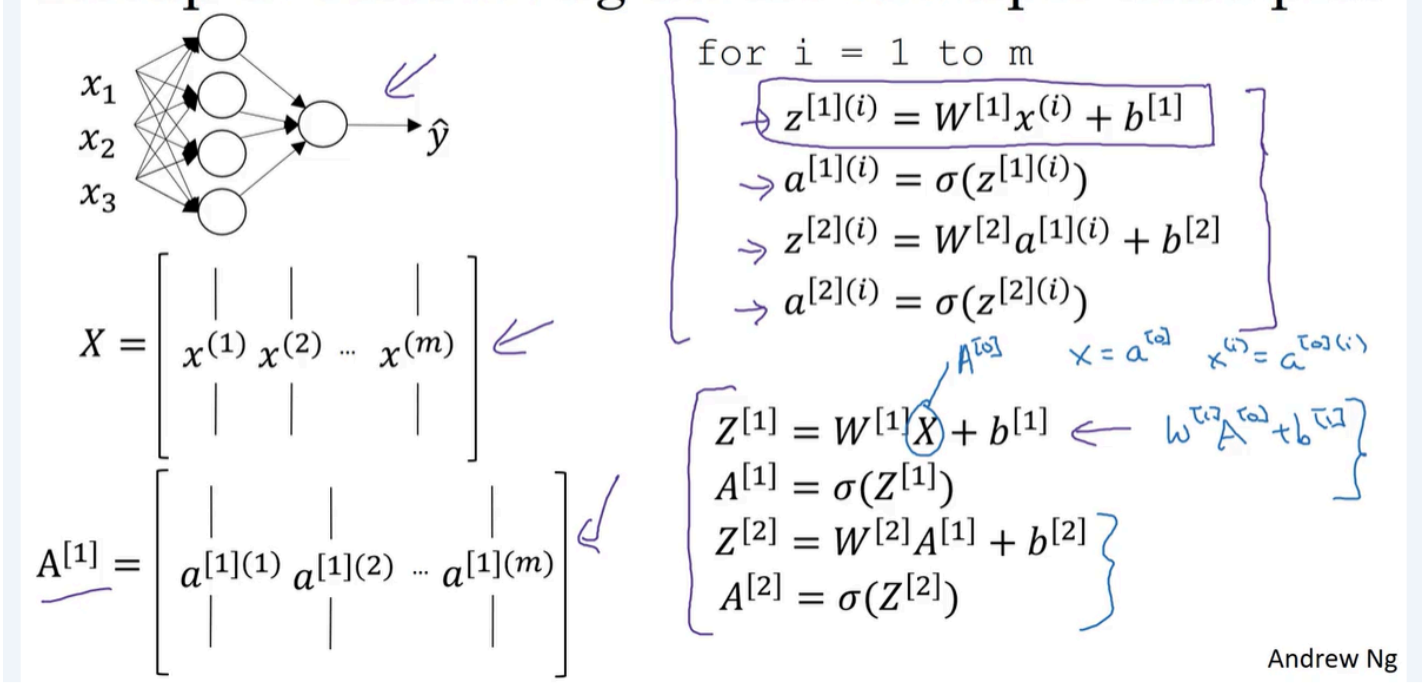
Vectorizing Across Multiple Examples



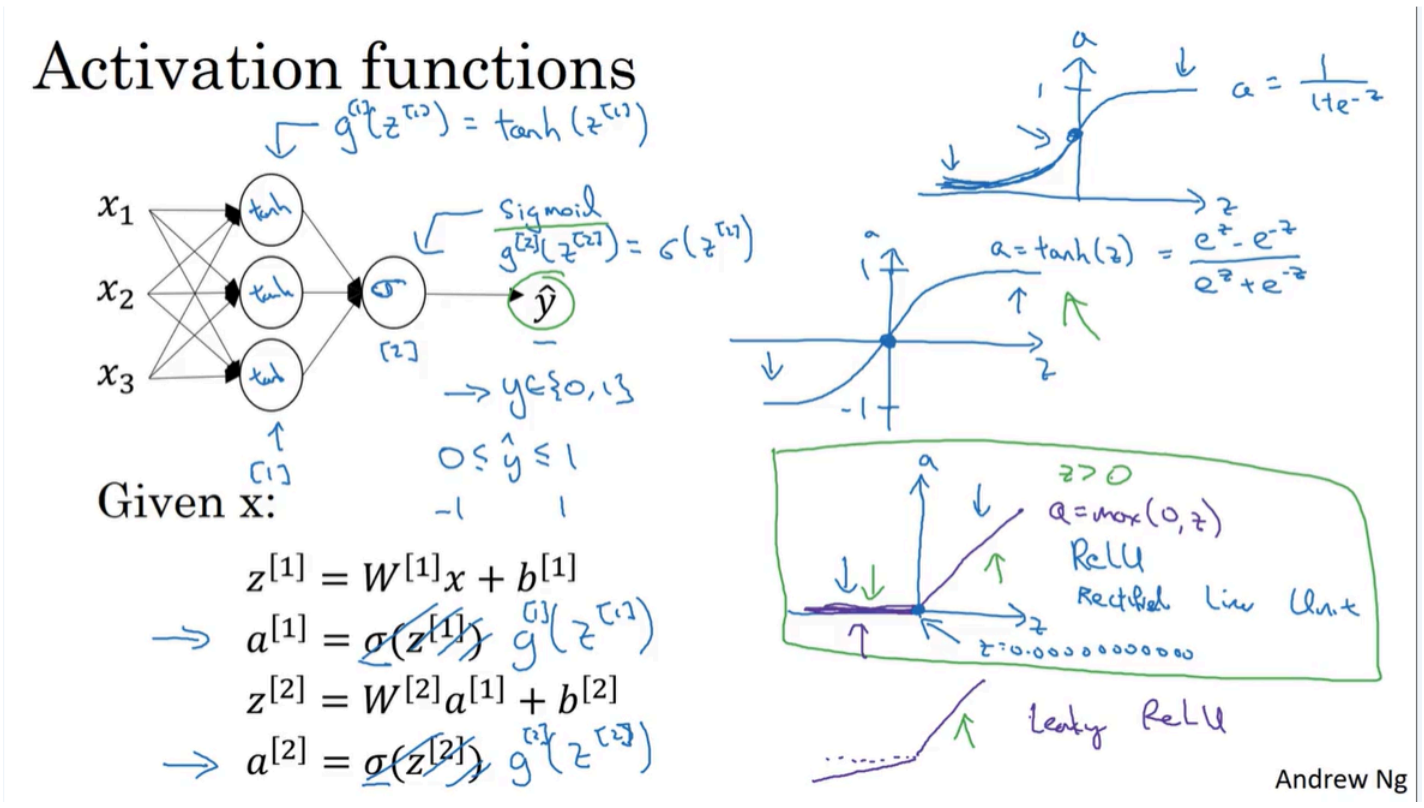
Okay so the horizontally the matrix A goes over different training examples. And vertically the different indices in the matrix A corresponds to different hidden units. And a similar intuition holds true for the matrix Z as well as for X where horizontally corresponds to different training examples.

Explanation for Vectorized Implementation





Activation Functions



When you build your neural network, one of the choices you get to make is what activation function to use in the hidden layers as well as at the output units of your neural network. So far, we've just been using the sigmoid activation function, but sometimes other choices can work much better. Let's take a look at some of the options. In the forward propagation steps for the neural network, we had these two steps where we use the sigmoid function here. So that sigmoid is called an activation function. And here's the familiar sigmoid function, a = 1/(1 + e^-z). So in the more general case, we can have a different function g(z).

Which I'm going to write here where g could be a nonlinear function that may not be the sigmoid function. So for example, the sigmoid function goes between zero and one. An activation function that almost always works better than the sigmoid function is the tangent function or the hyperbolic tangent function. So this is z, this is a, this is a = tanh(z). And this goes between +1 and -1. The formula for the tanh function is e to the z minus e to-z over their sum.

$$tanh = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

And it's actually mathematically a shifted version of the sigmoid function.
So as a sigmoid function just like that but shifted so that it now crosses the zero zero point on the scale. So it goes between -1 and +1. And it turns out that for hidden units, if you let the function g(z)=tanh(z). This almost always works better than the sigmoid function because with values between plus one and minus one, the mean of the activations that come out of your hidden layer are closer to having a zero mean. And so just as sometimes when you train a learning

algorithm, you might center the data and have your data have zero mean using a tanh instead of a sigmoid function. Kind of has the effect of centering your data so that the mean of your data is close to zero rather than maybe 0.5. And this actually makes learning for the next layer a little bit easier.

We'll say more about this in the second course when we talk about optimization algorithms as well. But one takeaway is that I pretty much never use the sigmoid activation function anymore. The tan h function is almost always strictly superior. The one exception is for the output layer because if y is either zero or one, then it makes sense for y hat to be a number that you want to output that's between zero and one rather than between -1 and 1. So the one exception where I would use the sigmoid activation function is when you're using binary classification. In which case you might use the sigmoid activation function for the upper layer. So g(z2) here is equal to sigmoid of z2.

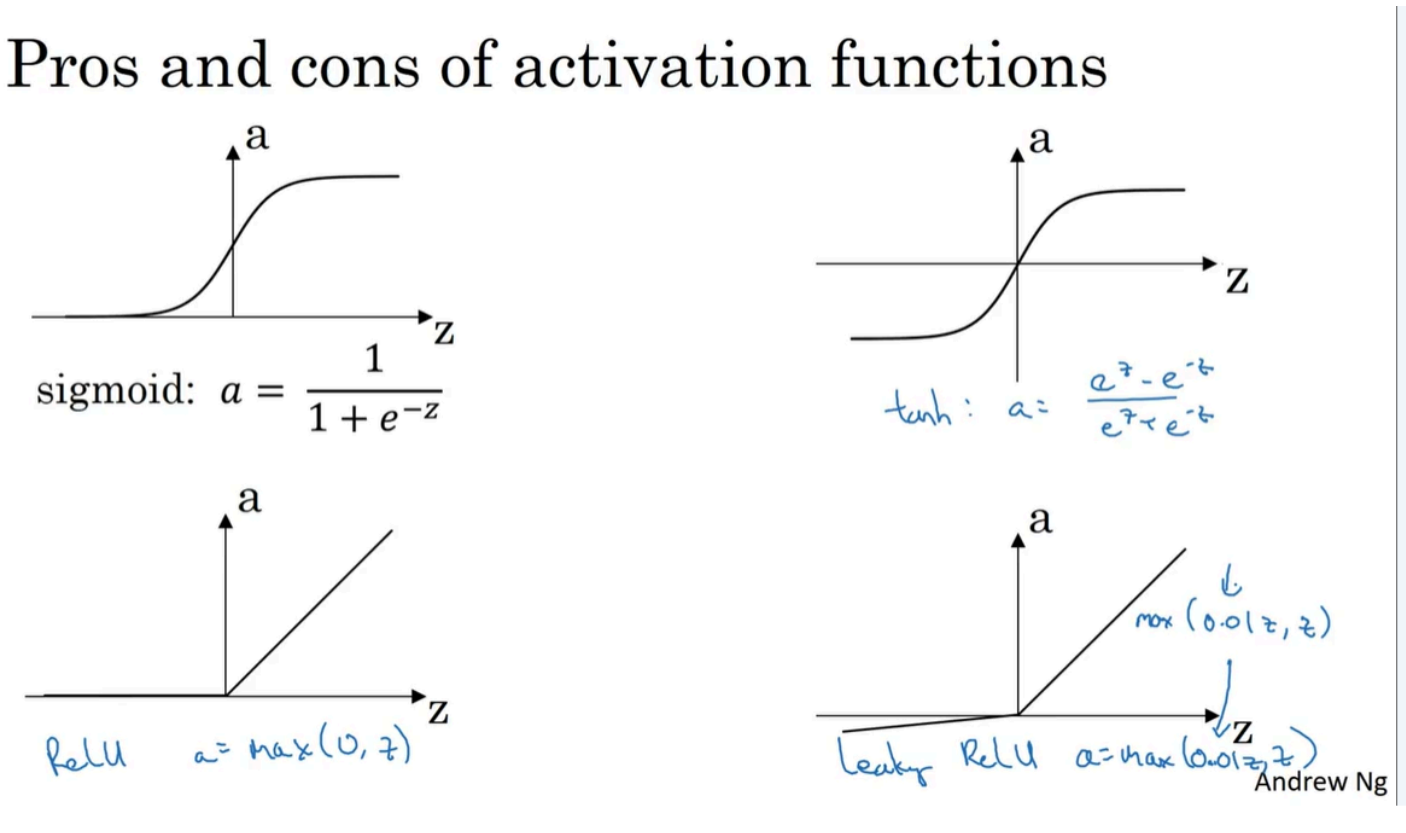
And so what you see in this example is where you might have a tan h activation function for the hidden layer and sigmoid for the output layer. So the activation functions can be different for different layers. And sometimes to denote that the activation functions are different for different layers, we might use these square brackets superscripts as well to indicate that gf square bracket one may be different than gf square bracket two, right. Again, square bracket one superscript refers to this layer and superscript square bracket two refers to the output layer. Now, one of the downsides of both the sigmoid function and the tan h function is that if z is either very large or very small, then the gradient of the derivative of the slope of this function becomes very small. So if z is very large or z is very small, the slope of the function either ends up being close to zero and so this can slow down gradient descent. So one other choice that is very popular in machine learning is what's called the **rectified linear unit (ReLU)**.

So the value function looks like this and the formula is a = max(0,z). So the derivative is one so long as z is positive and derivative or the slope is zero when z is negative. If you're implementing this, technically the derivative when z is exactly zero is not well defined. But when you implement this in the computer, the odds that you get exactly z equals 000000000000 is very small. So you don't need to worry about it. In practice, you could pretend a derivative when z is equal to zero, you can pretend is either one or zero. And you can work just fine.

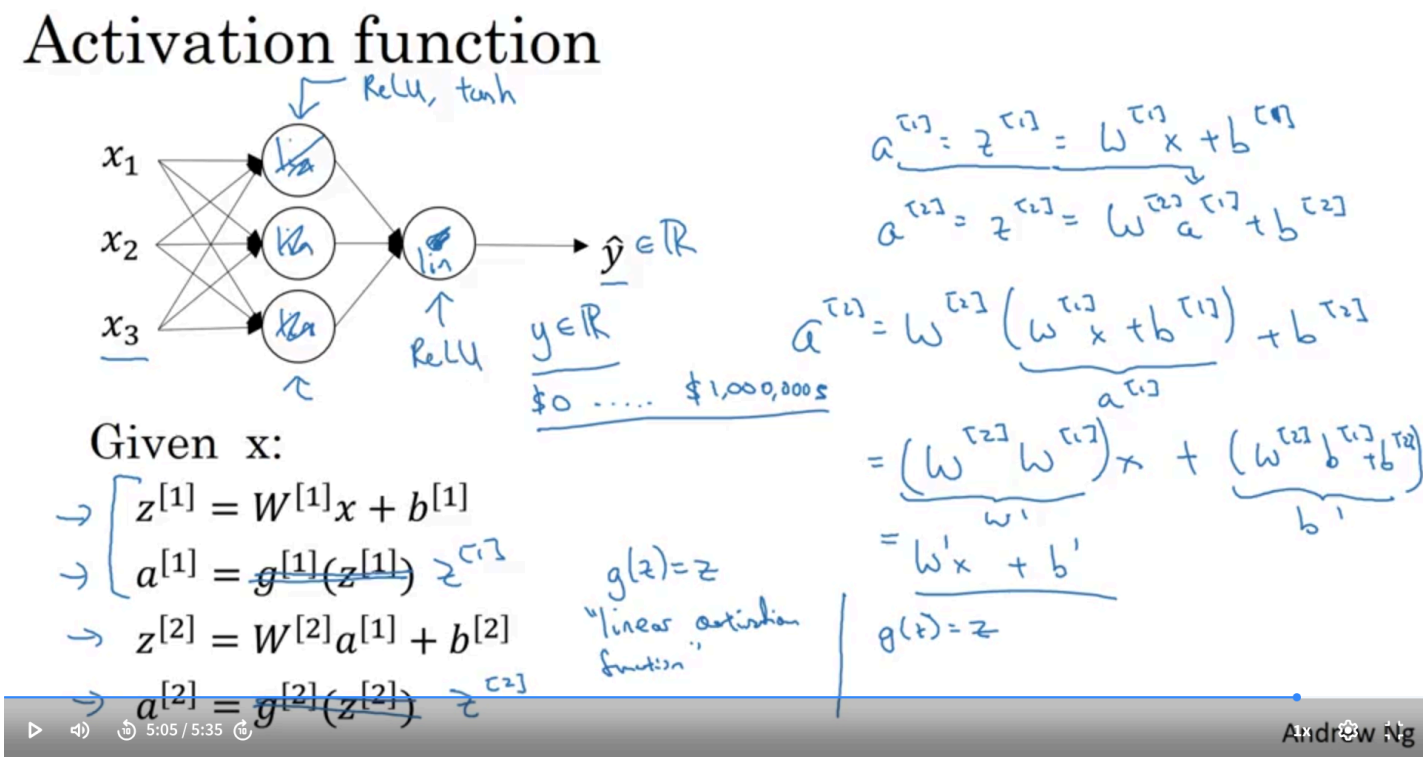
So the fact is not differentiable. The fact that, so here's some rules of thumb for choosing activation functions. If your output is zero one value, if you're using binary classification, then the sigmoid activation function is very natural choice for the output layer. And then for all other units value or the rectified linear unit is increasingly the default choice of activation function. So if you're not sure what to use for your hidden layer, I would just use the ReLU activation function, is what you see most people using these days. Although sometimes people also use the tan h activation function. One disadvantage of the value is that the derivative is equal to zero when z is negative.

In practice this works just fine. But there is another version of the value called the Leaky ReLU. We'll give you the formula on the next slide but instead of it being zero when z is negative, it just takes a slight slope like so. So this is called Leaky ReLU. This usually works better than the ReLU activation function. Although, it's just not used as much in practice. Either one should be fine.

Although, if you had to pick one, I usually just use the ReLU. And the advantage of both the ReLU and the Leaky ReLU is that for a lot of the space of z, the derivative of the activation function, the slope of the activation function is very different from zero. And so in practice, using the ReLU activation function, your neural network will often learn much faster than when using the tan h or the sigmoid activation function. And the main reason is that there's less of this effect of the slope of the function going to zero, which slows down learning. And I know that for half of the range of z, the slope for ReLU is zero. But in practice, enough of your hidden units will have z greater than zero. So learning can still be quite fast for most training examples.



Why do you need Non-Linear Activation Functions?

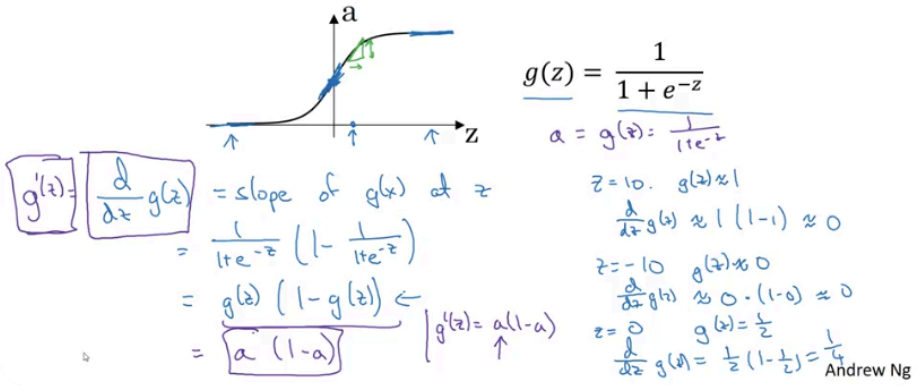


And it turns out that if you use a linear activation function or alternatively, if you don't have an activation function, then no matter how many layers your neural network has, all it's doing is just computing a linear activation function. So you might as well not have any hidden layers. Some of the cases that are briefly mentioned, it turns out that if you have a linear activation function here and a sigmoid function here, then this model is no more expressive than standard logistic regression without any hidden layer.

But then the hidden units should not use the activation functions. They could use ReLU or tanh or Leaky ReLU or maybe something else. So the one place you might use a linear activation function is usually in the output layer. But other than that, using a linear activation function in the hidden layer except for some very special circumstances relating to compression that we're going to talk about using the linear activation function is extremely rare.

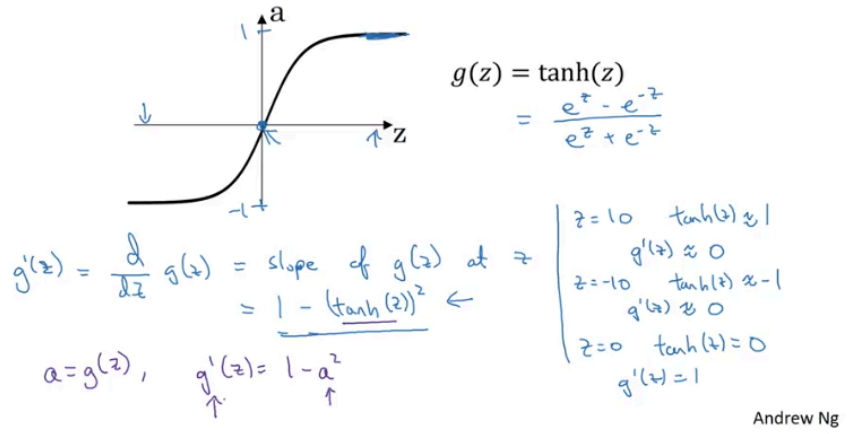
Derivatives of Activation Functions

Sigmoid activation function



$$g'(z) = g(z)(1 - g(z))$$

Tanh activation function



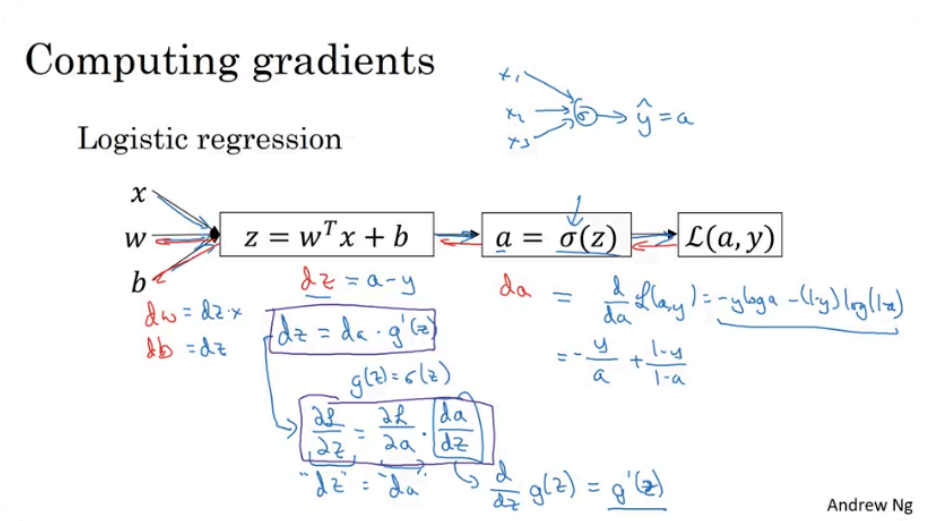
$$g(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

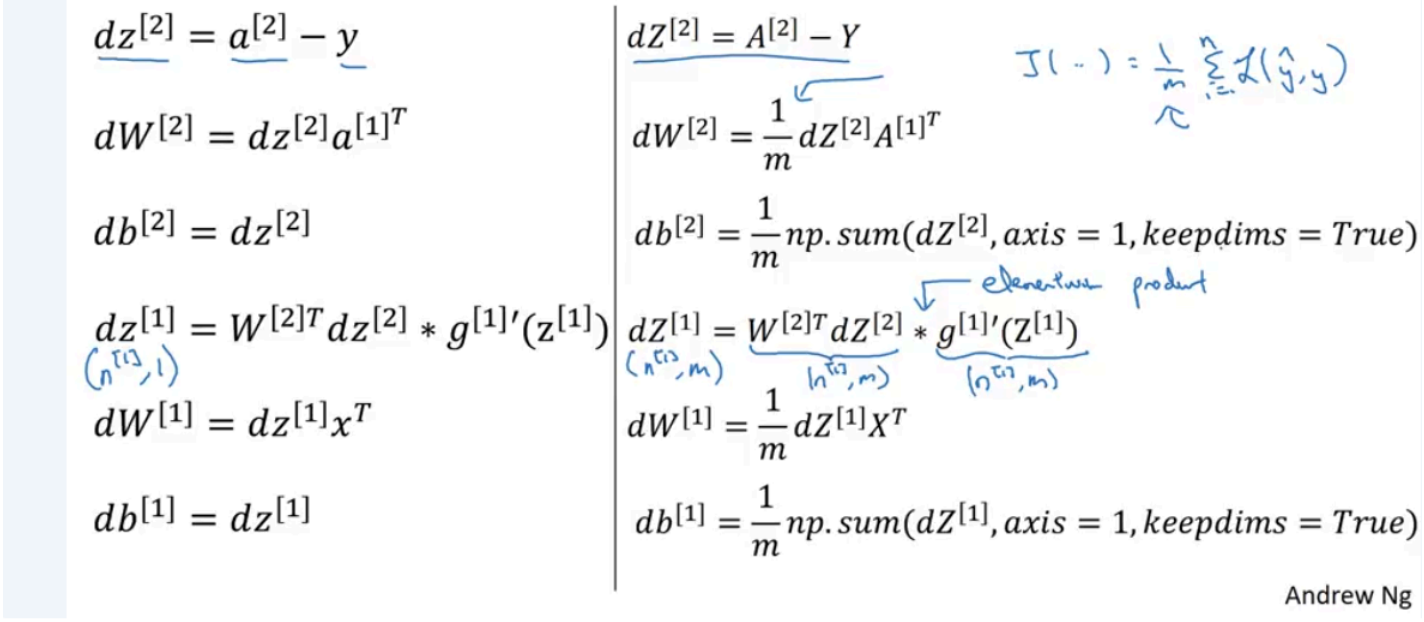
$$g'(z) = 1 - g^2(z)$$

Finally, here's how you compute the derivatives for the ReLU and Leaky ReLU activation functions. For the value g of z is equal to max of 0,z, so the derivative is equal to, turns out to be 0 , if z is less than 0 and 1 if z is greater than 0. It's actually undefined, technically undefined if z is equal to exactly 0. But if you're implementing this in software, it might not be a 100 percent mathematically correct, but it'll work just fine if z is exactly a 0, if you set the derivative to be equal to 1. It always had to be 0, it doesn't matter. If you're an expert in optimization, technically, g prime then becomes what's called a sub-gradient of the activation function g of z, which is why gradient descent still works. But you can think of it as that, the chance of z being exactly 0.000000.

It's so small that it almost doesn't matter where you set the derivative to be equal to when z is equal to 0. So, in practice, this is what people implement for the derivative of z. Finally, if you are training a neural network with a Leaky ReLU activation function, then g(z) is going to be max(0.01 z, z), and so, g′(z) of z is equal to 0.01 if z is less than 0 and 1 if z is greater than 0. Once again, the gradient is technically not defined when z is exactly equal to 0, but if you implement a piece of code that sets the derivative or that sets g prime to either 0.01 or or to 1, either way, it doesn't really matter. When z is exactly 0, your code will work just.

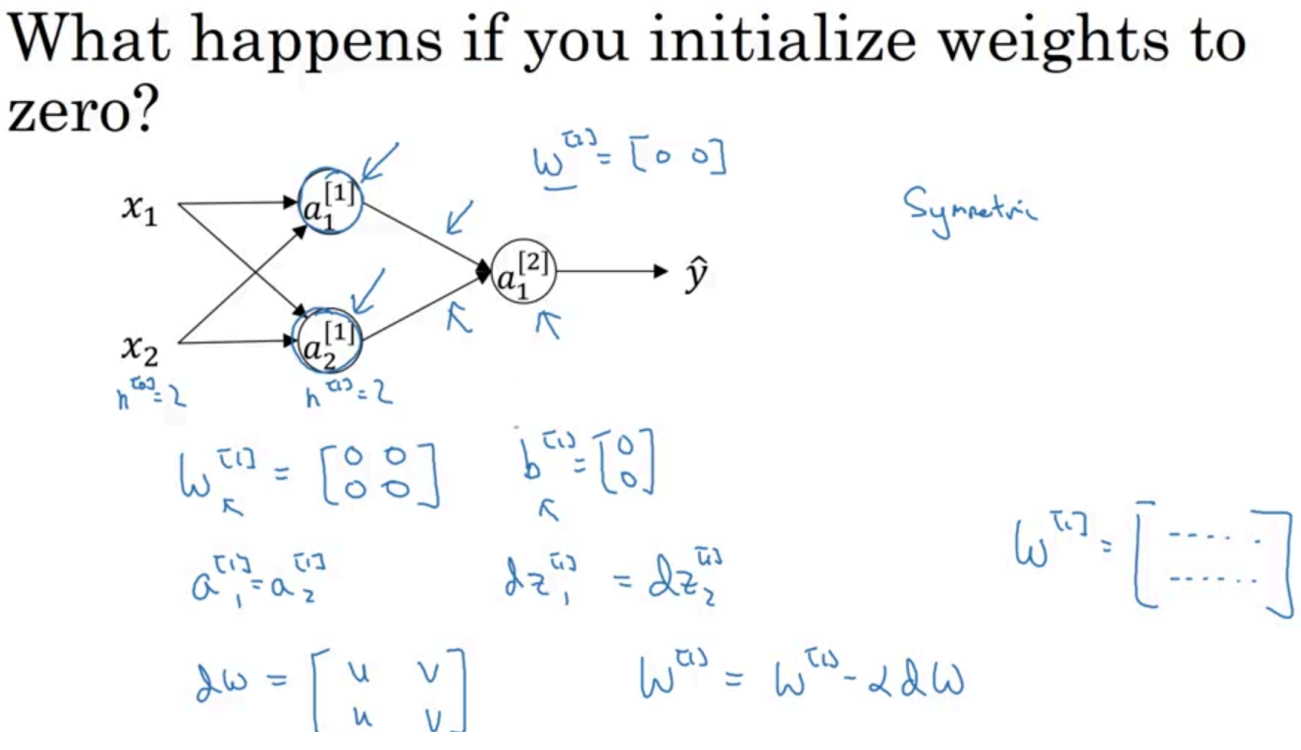
Gradient Descent for Neural Networks and Backpropagation Intuition





Random Initialization

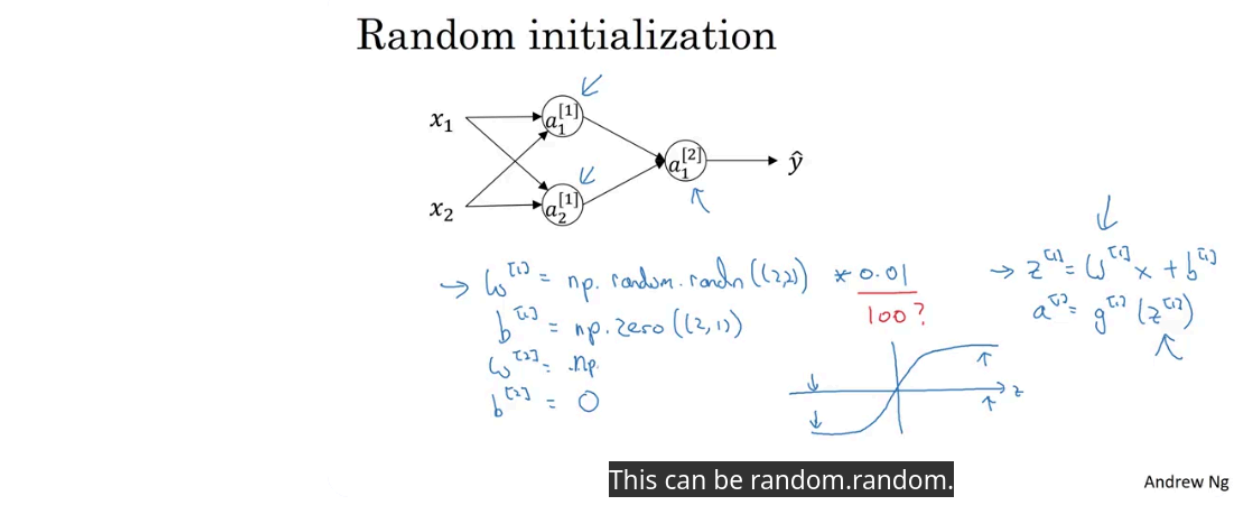
When you change your neural network, it's important to initialize the weights randomly. For logistic regression, it was okay to initialize the weights to zero. But for a neural network of initialize the weights to parameters to all zero and then applied gradient descent, it won't work. Let's see why.



But if you initialize the neural network this way, then this hidden unit and this hidden unit are completely identical. Sometimes you say they're completely symmetric, which just means that they're completing exactly the same function. And by kind of a proof by induction, it turns out that after every single iteration of training your two hidden units are still computing exactly the same function. Since plots will show that dw will be a matrix that looks like this. Where every row takes on the same value. So we perform a weight update.

So when you perform a weight update, w1 gets updated as w1- alpha times dw. You find that w1, after every iteration, will have the first row equal to the second row.

And of course, for larger neural networks, let's say of three features and maybe a very large number of hidden units, a similar argument works to show that with a neural network like this. Let me draw all the edges, if you initialize the weights to zero, then all of your hidden units are symmetric. And no matter how long you're upgrading the center, all continue to compute exactly the same function. So that's not helpful, because you want the different hidden units to compute different functions. The solution to this is to initialize your parameters randomly.



if w is too large, you're more likely to end up even at the very start of training, with very large values of z. Which causes your tanh or your sigmoid activation function to be saturated, thus slowing down learning.

If you don't have any sigmoid or tanh activation functions throughout your neural network, this is less of an issue. But if you're doing binary classification, and your output unit is a sigmoid function, then you just don't want the initial parameters to be too large.

Note for lecture:

$W^{[k]}$:

- number of rows is the number of units in the k-th layer
- number of column is the number of input of the layer