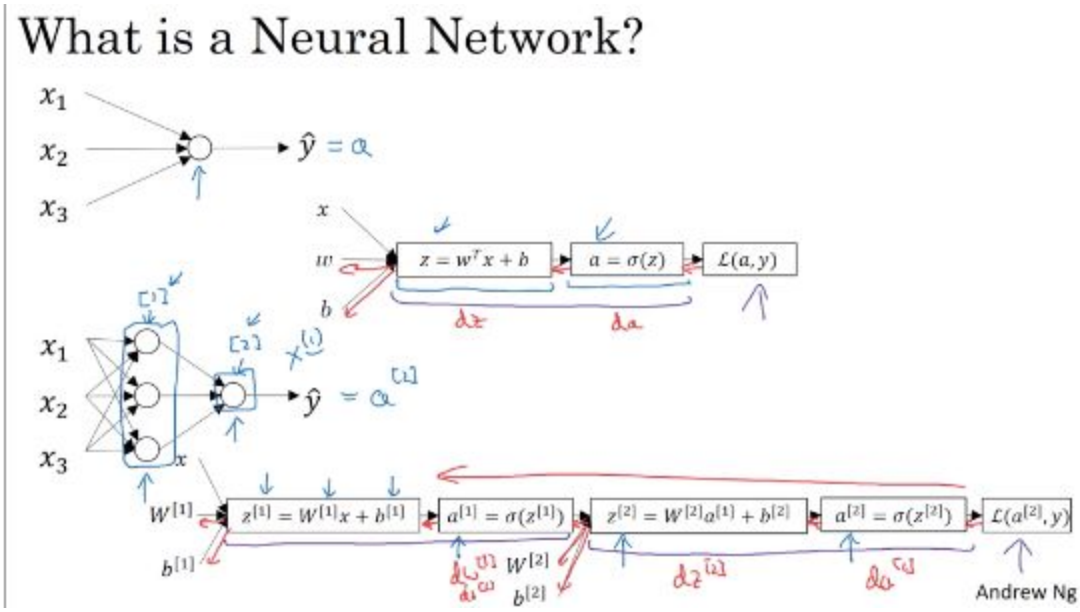
Andrew Ng's Deep Learning Class

Week 3: Shallow Neural Networks

Neural Networks Overview

- This is a small overview of what we are going to learn this week. Last week we looked at logistic regression and its computation graph where we had 2 steps: one step for computing z and another step for computing a using $\sigma(z)$.



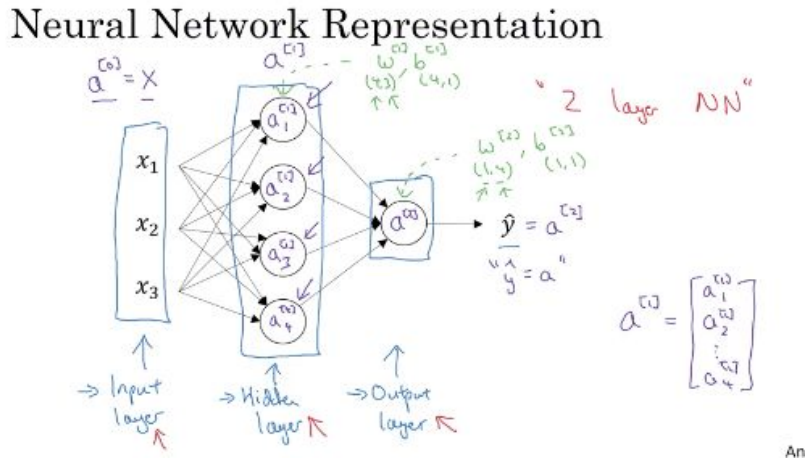
Neural Networks perform those 2 steps in one step in a neuron. We introduce new notation: $[1], [2], ..., [n]$ refers to the layers of a neural network. From left to right we compute $z^{[1]}$ *and* $a^{[1]}$ for the first layer, $z^{[2]}$ *and* $a^{[2]}$ for the second layer and so on until we get to the loss function $L(a^{[2]}, y)$. In this case we assumed the neural network had 2 layers. From right to left we compute the derivatives.

Neural Network Representation

- We'll start by looking at a neural network with a single hidden layer. Such a network is known as 2 layer network (the 2 layers are the hidden layer and the output layer). Now we go over the inputs and outputs:
  - The *input* layer has a list of x examples and is denoted as $a^{[0]}$
  - The *hidden* layer has an activation $a^{[1]}$ and each neuron has an a values as follows: $a_1^{[1]}, a_2^{[1]}, a_3^{[1]}, a_4^{[1]}$

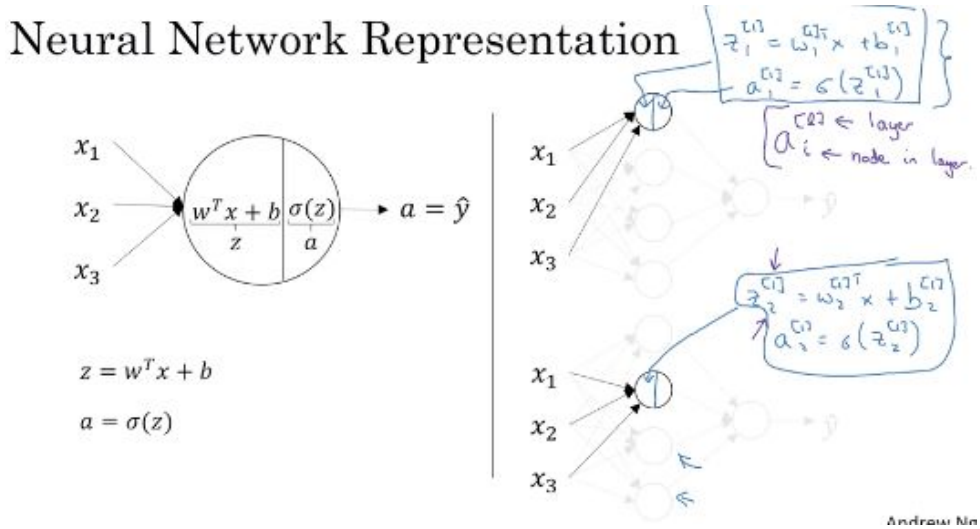
- The *output* layer has an activation $a^{[2]}$

$a^{[1]}$ has $w^{[1]}$ and $b^{[1]}$ parameters . $w^{[1]}$ has dimensions $(4,3)$ because there are 4 neurons and 3 X inputs. $b^{[1]}$ had dimensions $(4,1)$. Similarly, $a^{[2]}$ has parameters $w^{[2]}$ and $b^{[2]}$; $w^{[2]}$ has dimensions $(1,4)$ because there is only one neuron and there are 4 input parameters, $b^{[2]}$ has dimensions $(1,1)$.



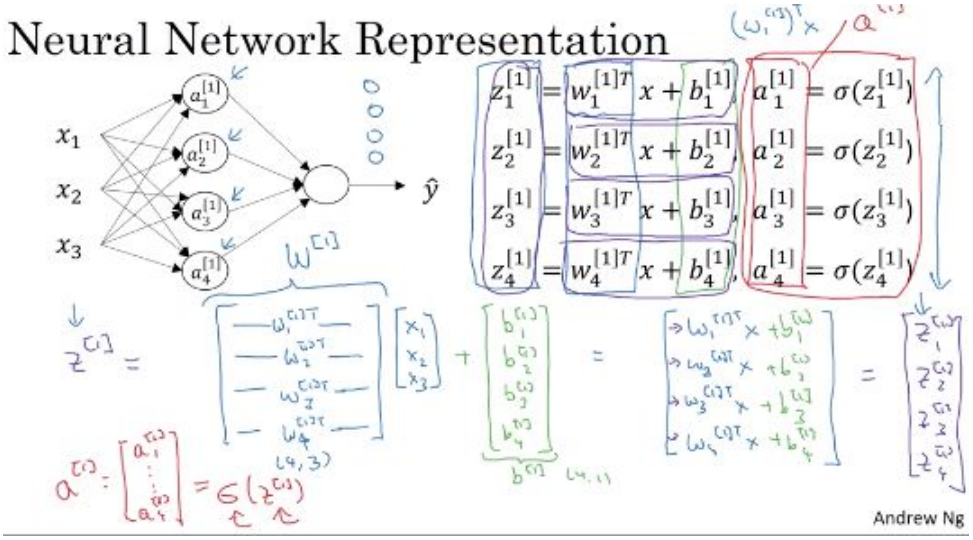
Computing a Neural Network's Output

- We are going to look in detail at what a neural network computes. On the left side we see how logistic regression is done. Neural Networks do the same thing but many times. On the right side we are going to look at the computations of of the first 2 neurons. The first neuron will compute $z_1^{[1]} = w_1^{[1]T} * X + b_1^{[1]}$ and $a_1^{[1]} = \sigma(z_1^{[1]})$. The second neuron will compute $z_2^{[1]} = w_2^{[1]T} * X + b_2^{[1]}$ and $a_2^{[1]} = \sigma(z_2^{[1]})$

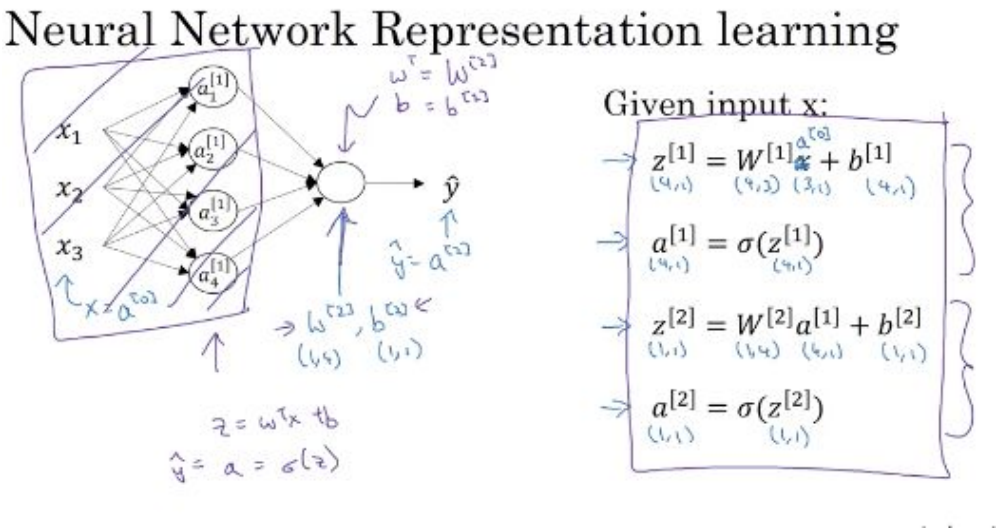


- Now let's take the equations from the first 2 neurons and continue with the rest. On the top right we show the the equations for all neurons. Now we can do a vectorized version of them: we start by 'stacking' $w_1^{[1]T}, w_2^{[1]T}, w_3^{[1]T}, w_4^{[1]T}$ to form the vector $w^{[1]}$ which is

then multiplied by the vector $x = [x_1, x_2, x_3]$ and then we add the vector $b = [b_1^{[1]}, b_2^{[1]}, b_3^{[1]}, b_4^{[1]}]$ and the result is the vector $z^{[1]} = [z_1^{[1]}, z_2^{[1]}, z_3^{[1]}, z_4^{[1]}]$



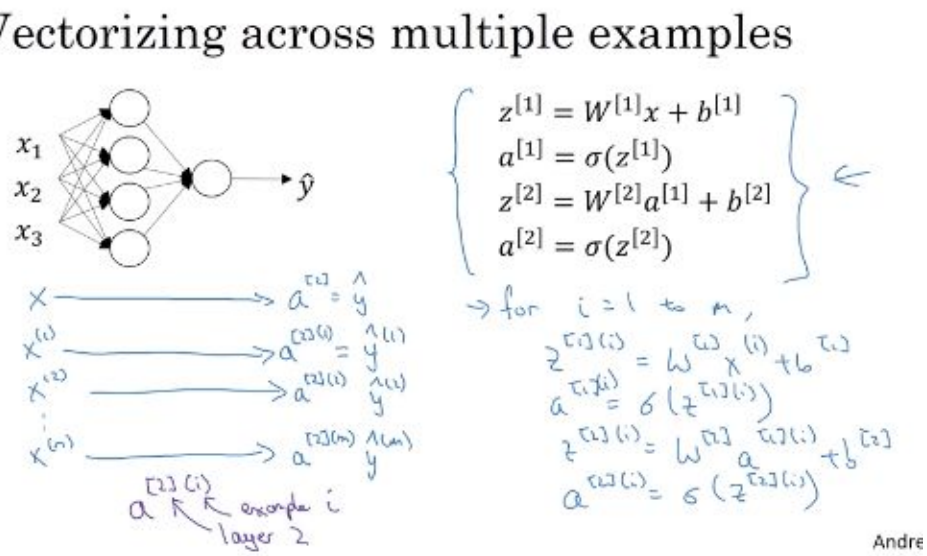
- Taking that a bit further we can now see all the vectorized equations we need to compute the output of a 2 layer NN. They are shown on the right below and are:
  - $z^{[1]} = W^{[1]} * x + b^{[1]}$. Remember we also know x as $a^{[0]}$ so we can also write this equation as $z^{[1]} = W^{[1]} * a^{[0]} + b^{[1]}$
  - $a^{[1]} = \sigma(z^{[1]})$
  - $z^{[2]} = W^{[2]} * a^{[1]} + b^{[2]}$
  - $a^{[2]} = \sigma(z^{[2]})$



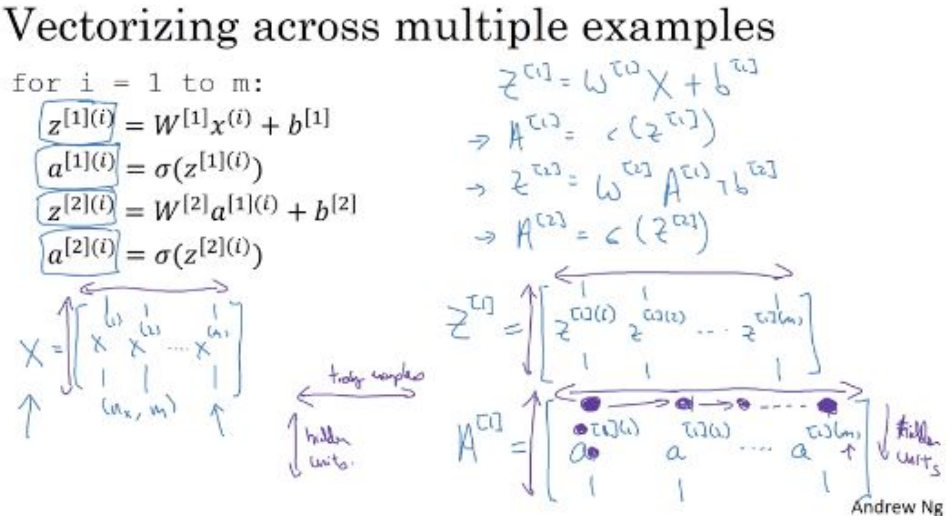
Vectorizing across multiple examples

- What we did before was for just one training example, x. Now we are going to see how we will do it for m training examples. Let's start with the un-vectorized version, shown

below. We start by saying $x \rightarrow a^{[2]} \rightarrow y^*$. The other training examples are then represented by $x^{(1)} \rightarrow a^{[2](1)} \rightarrow y^{*(1)}$, $x^{(2)} \rightarrow a^{[2](2)} \rightarrow y^{*(2)}$ until $x^{(m)} \rightarrow a^{[2](m)} \rightarrow y^{*(m)}$. The un-vectorized implementation is shown on the bottom right corner.



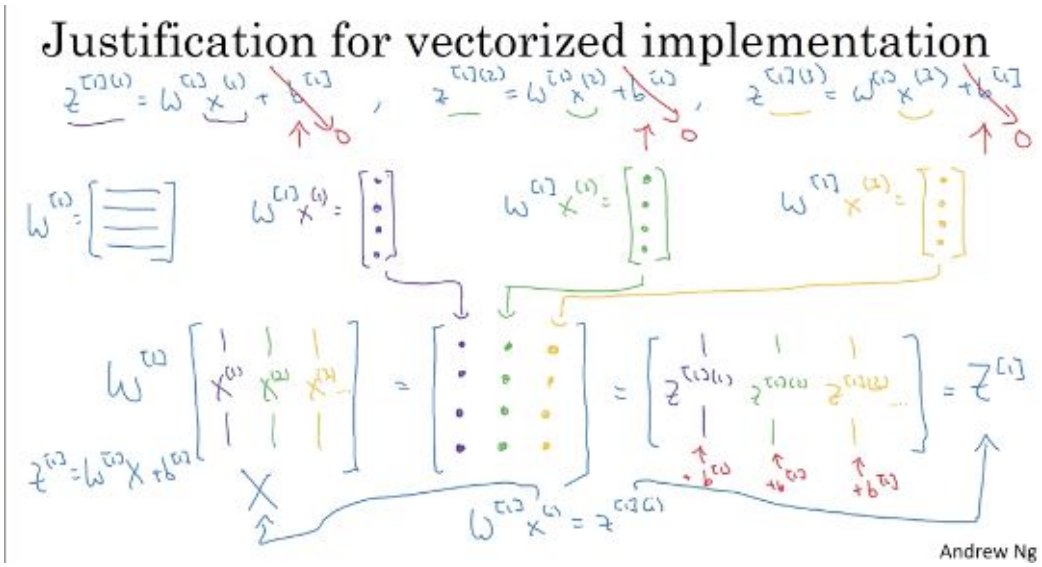
- The fully vectorized implementation across multiple examples is shown below. The equations are:
  - $Z^{[1]} = W^{[1]}X + b^{[1]}$
  - $A^{[1]} = \sigma(Z^{[1]})$
  - $Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
  - $A^{[2]} = \sigma(Z^{[2]})$



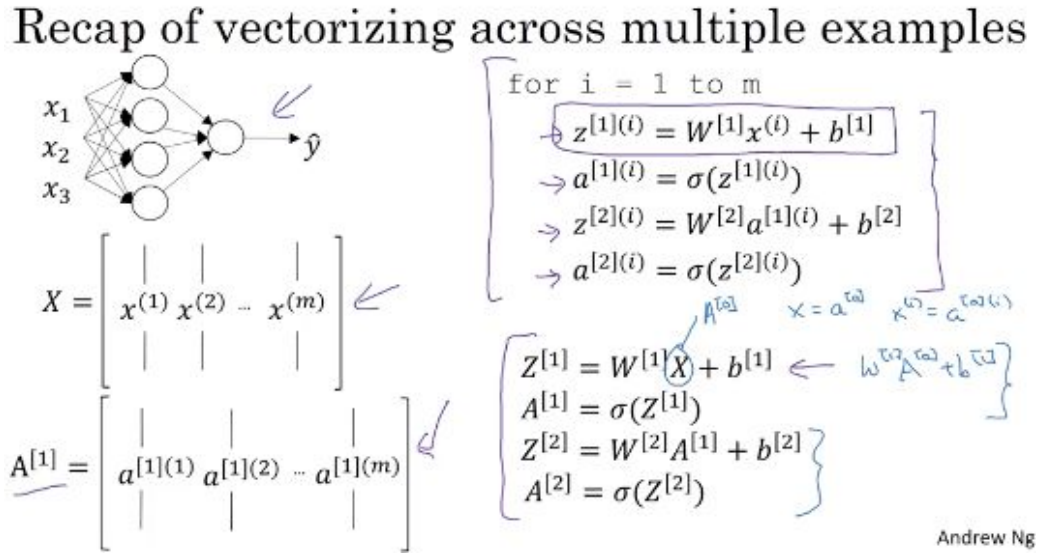
While we horizontally stack all the learning examples and z's and a's we need to remember the resulting matrices denote training examples from left to right and hidden units from top to bottom.

Explanation for Vectorized Implementation

- In this session we go a bit deeper into justifying the vectorized implementation introduced in the last session. What we do is take the z equations for each training example like for example $z^{[1](1)} = W^{[1]} * x + b^{[1]}$. We stack the W parameters horizontally in a matrix $W^{[1]}$, we then stack all the X's horizontally to create X. Our multiplication of $W^{[1]} and\ X$ is then equal to $Z^{[1]}$



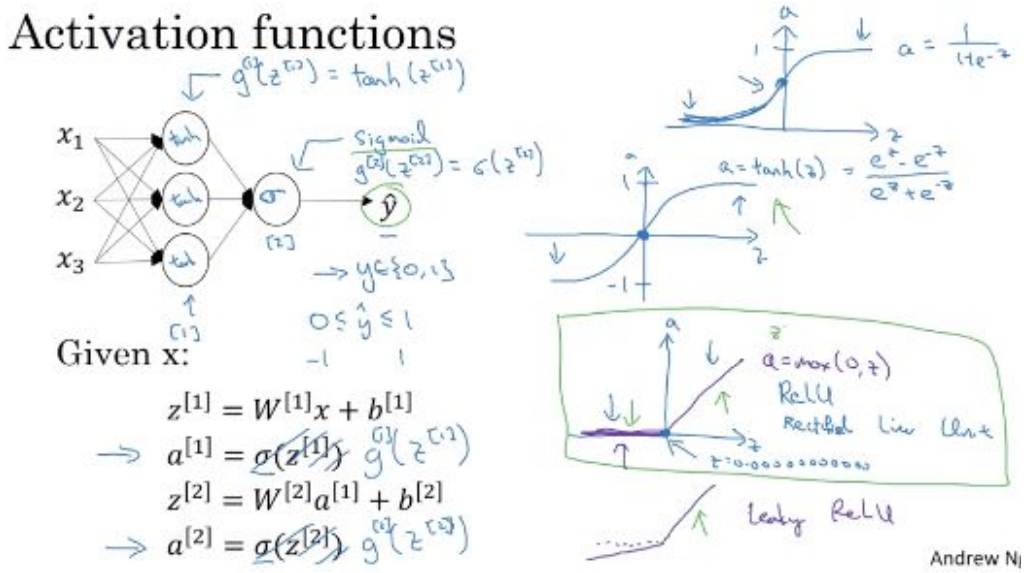
- Recaping vectorizing across multiple examples: on the right we see the unvectorized implementation. Below we have the vectorized implementation: we stack up the training examples to have X and we also remember that X is also known as $A^{[0]}$ so this gives us symmetry between the equations where these equations are similar and we just advance the indices.



Next we are going to talk about other possible activation functions besides the sigmoid function. Turns out the sigmoid function is not be the best choice for neural networks and we are going to look at some alternatives.

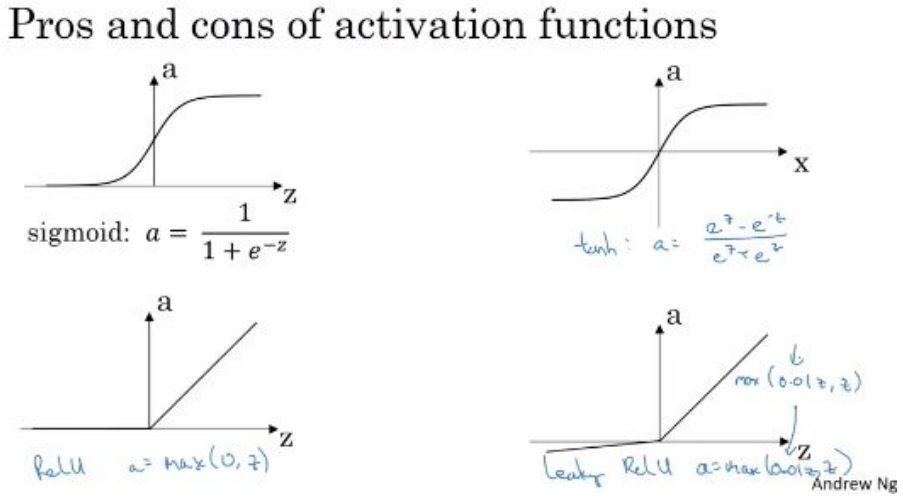
Activation Functions

- There are alternatives to using the sigmoid function. One alternative that most of the time works better than sigmoid is the tangent function (tanh). It is defined as $tanh(z) = e^{z} - e^{-z}/e^{z} + e^{-z}$. In practice we can look at this as a shifted version of sigmoid function that has the same form but centered on (0,0) as shown below. So the takeaway is that we pretty much don't use sigmoid as an activation function anymore. The disadvantage of the tangent function is that the derivative is very small and z is very small or very large and that slows gradient descent.



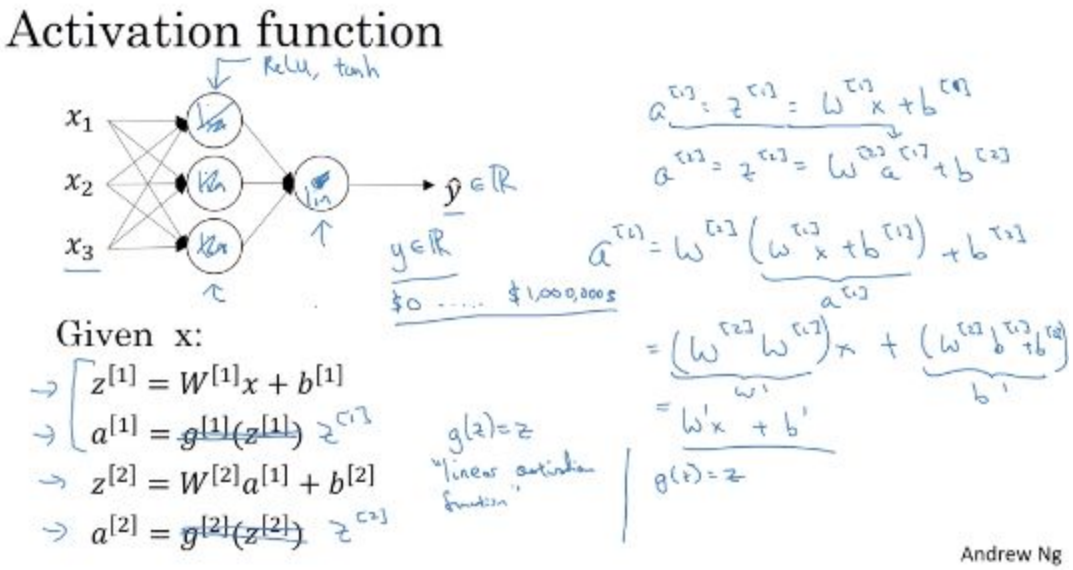
In the neural network shown above we have a hidden layer that uses the tanh function as an activation function and a sigmoid function on the output layer. It is OK to use the sigmoid function is you expect your output to be 1 or 0 as is the case for binary classification. Another activation option is ReLu (Rectify linear unit), whose formula is $max(0, z)$ and a variant called Leaky ReLu. Rules of thumb for activation function choice: Use sigmoid for binary classification and relu for the rest.

- Summarizing advantages and disadvantages of activation functions:
    - Never use the sigmoid function unless it is for the output layer of binary classification. Use tanh instead which is superior
    - Most common activation function is ReLu: if you don't know what to use, use ReLu. Also try Leaky ReLu



Why do we need non-linear Activation Functions?

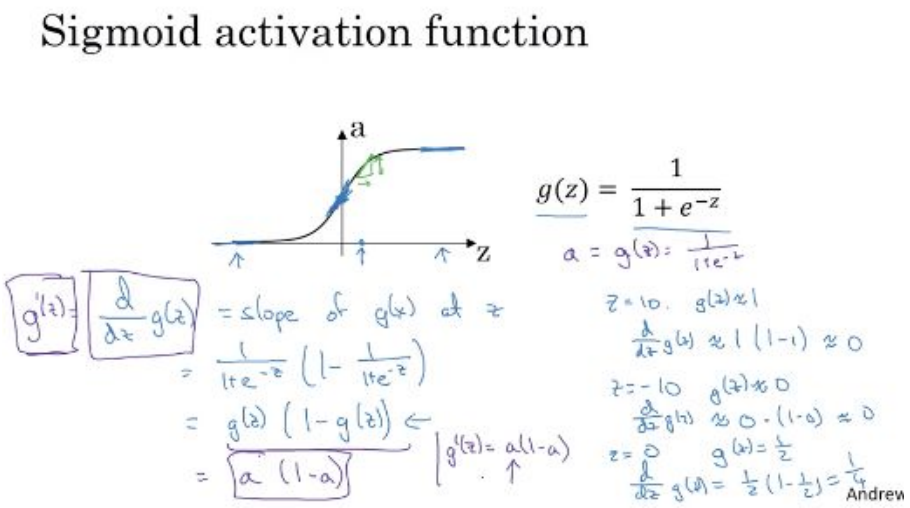
- To understand why we need non linear activation functions let's take our 4 equations for a 1 hidden layer neural network and we are going to set $a^{[1]} = z^{[1]}$, also known as an identity linear function. If we do that we end up with a final function of $W'x + b'$. Basically we are saying that having a linear function means your hidden layers will only compute linear functions of the inputs and this makes these layers useless so you might as well not have them. The summary is that the linear hidden layer is useless because the composition of two linear functions is itself a linear function.



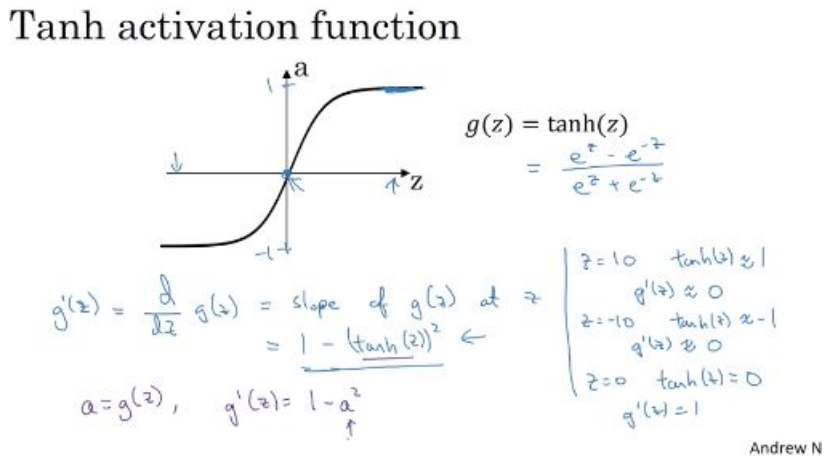
There is just one place where we could use a linear function and that is the output layer of a linear regression problem. In that case the output y is a real number. However, the hidden layers should never use linear functions. Try some of the other choices we know about (ReLu, tanh or Leaky ReLu).

Derivatives of Activation Functions

- Let's look at the derivatives for the activation functions we have previously mentioned. Starting with the sigmoid function we know that the derivative is the slope of g(z) at point z. So $g(z) = dg(z)/dz = g(z)(1 - g(z) = a(1 - a)$. Let's check with a couple of values:
    - If $z = 10,\ g(z) \approx 1$ *then the derivative is* $= 1(1 - 1) \approx 0$ which is OK since at that point the sigmoid function is almost a line so there is no slope
    - If $z = -10,\ g(z) \approx 0$ *then the derivative is* $= 0\ (1 - 0) \approx 0$ which is also OK since at that point the sigmoid function i also almost and line and therefore there is no slope
    - If $z = 0,\ g(z) \approx 1/2$ *and the derivative* $= 1/2(1 - 1/2) \approx 1/4$ which is the correct value

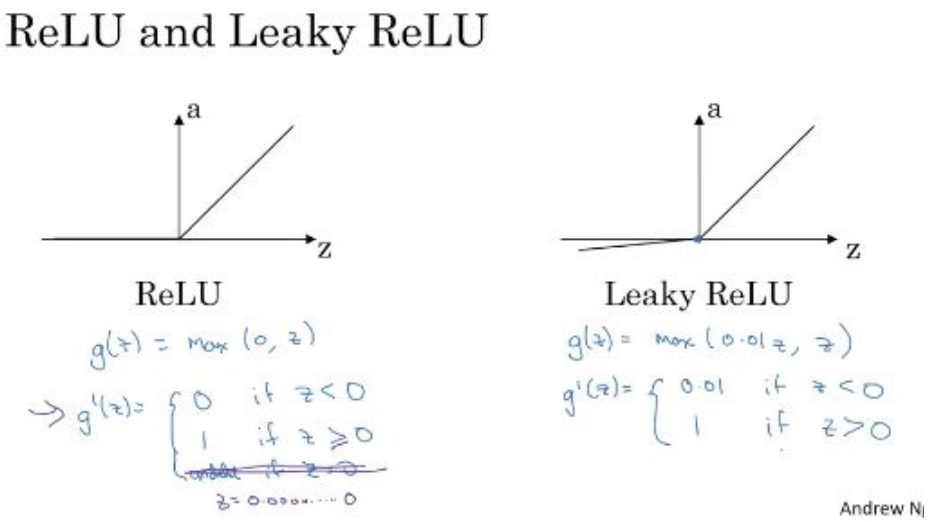


- Now let's take a look at tanh: *the derivative* $g'(z) = 1 - (tanh(x))^2 = 1 - a^2$, Checking some values:
    - If $z = 10,\ tanh \approx 1$ *then the derivative* $g'(z) = 1 - 1^2 \approx 0$
    - If $z = -10,\ tanh \approx -1$ *then the derivative* $g'(z) = 1 - (-1)^2 \approx 0$
    - If $z = 0,\ tanh \approx 0$ *then the derivative* $g'(z) = 1 - 0^2 \approx 1$



- Finally, let's look at the derivatives for ReLu and Leaky ReLu.

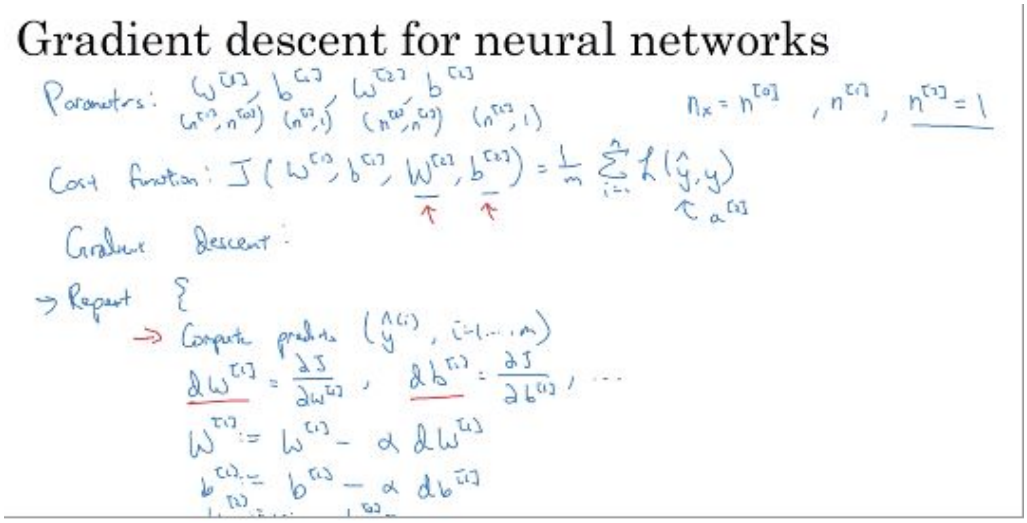
- For ReLu: $g(z) = max(0, z)$ *then* $g'(z) = \{0 \text{ if } z < 0,\ 1 \text{ if } z \geq 0\}$
- For Leaky ReLu: $g(z) = max(0.01, z)$ *then* $g'(z) = \{0.01 \text{ if } z < 0,\ 1 \text{ if } z \geq 0\}$



Knowing these formulas enable us to implement gradient descent for neural networks which we will take a look next.

Gradient Descent of Neural Networks

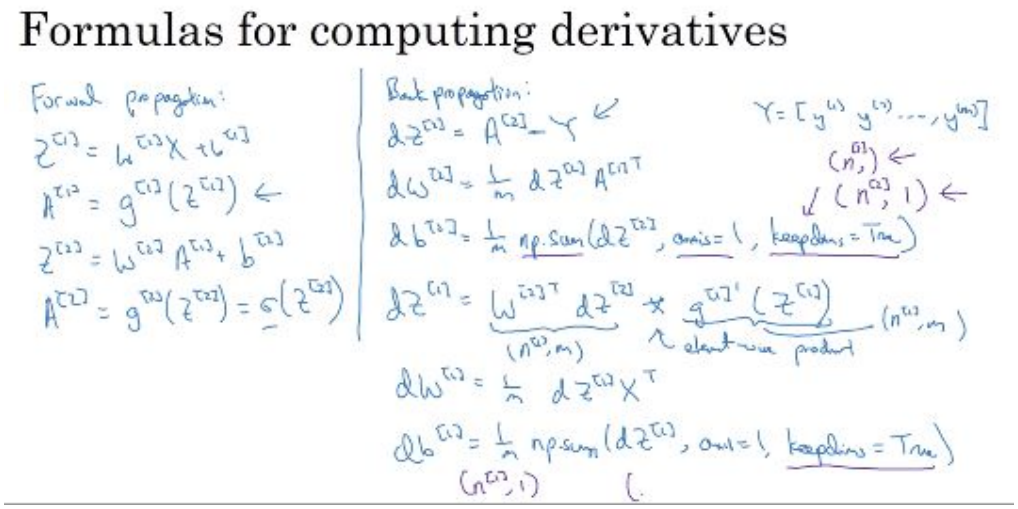
- Now that we know the derivatives of the different activation functions we can look at gradient descent. Let's look at gradient descent for a 1 hidden layer network. Our parameters are $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$ and our cost function is a function of those parameters: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = (1/m) \sum_{i=1}^{n} L(y^*, y)$. For now let's assume we are doing binary classification so our loss function would be the logistic regression loss function.



Now we can train the parameters using gradient descent. First we initialize them to some values (not zeros). Then what we do is we repeat the following steps until we converge:

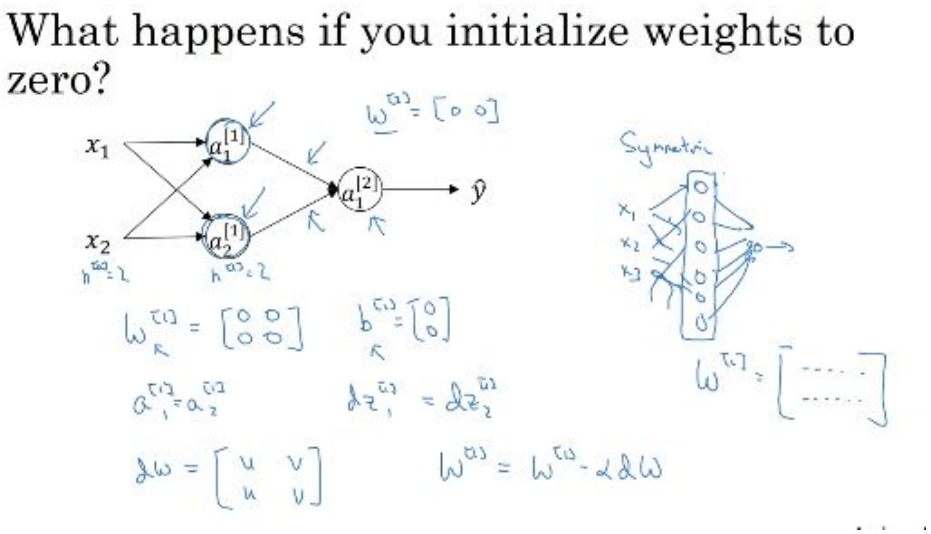
1. Compute predictions $(y^* for\ 1 = 1....m)$
2. Compute the derivatives: $dw^{[1]} = dJ/dw^{[1]},\ db^{[1]} = dJ/db^{[1]}\ and\ so\ on$
3. Update the parameters:
    a. $w^{[1]} = w^{[1]} - \alpha * dw^{[1]}$
    b. $b^{[1]} = b^{[1]} - \alpha * db^{[1]}$
    c. $w^{[2]} = w^{[2]} - \alpha * dw^{[2]}$
    d. $b^{[2]} = b^{[2]} - \alpha * db^{[2]}$

- Now we can take a look at the equations for computing the derivatives. They are:
    - $dz^{[1]} = A^{[2]} - Y$
    - $dw^{[2]} = (1/m)\ dz^{[2]}A^{[1]T}$
    - $db^{[2]} = 1/m\ np.sum(dz^{[2]}, axis = 1,\ keepdims = True)$ with dimensions $(n^{[2]}x\ 1)$
    - $dz^{[1]} = w^{[2]T}dz^{[2]} * g^{[1]'}(Z^{[1]})$ where * is the element-wise product so both dimensions on this product have to be $(n^{[1]}, m)$
    - $dw^{[1]} = 1/m * dz^{[1]} * X^{[T]}$
    - $db^{]1]} = 1/m * np.sum(dz^{[1]}, axis = 1\ keepdims = True)$ with dimensions $(n^{[1]}x\ 1)$

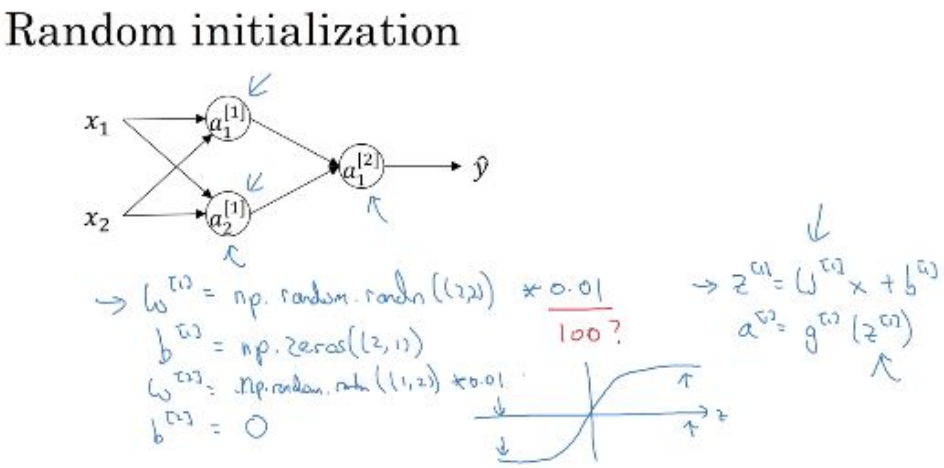


Random Initialization

- What happens if we initialized our parameters to zero? In a sentence: it means all the nodes in the hidden layer are computing the same function and that means every single iteration of training will continue to compute the same function. It also means it's useless to have more than one hidden unit. This is not helpful because we want each unit to compute different functions. The solution is to initialize the parameters randomly



- Here's how we initialize them randomly: first it is OK to initialize b to zero. We can then initialize
  - $w^{[1]} = np.random.randn(2,2) * 0.01$
  - $b^{[1]} = np.zeros(2,1)$
  - $w^{[2]} = np.random.randn(1,2) * 0.01$
  - $b^{[2]} = np.zeros$

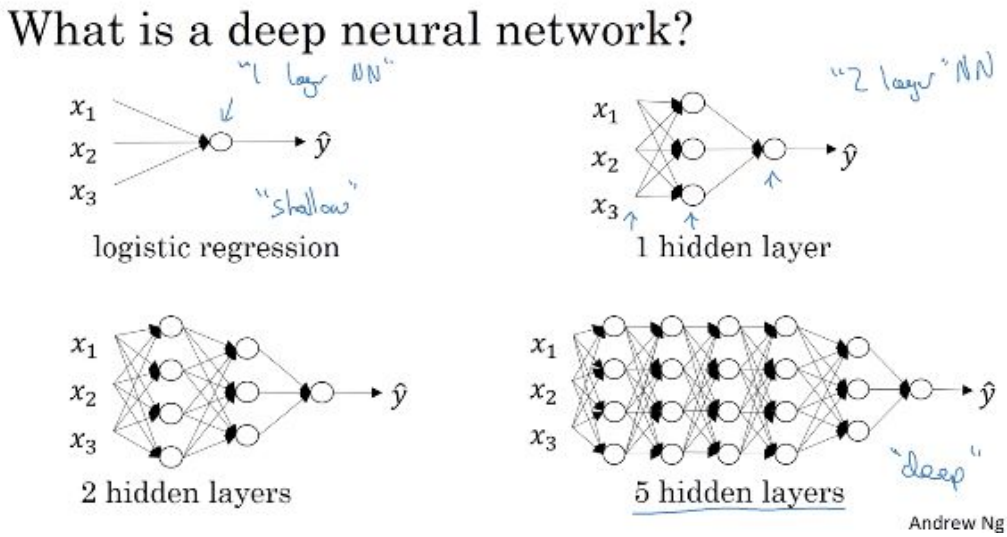


Why do we want to multiply by small numbers? Because that will give us a small w and therefore a small z. If we go with big value and we are doing binary classification (which means sigmoid) we end up in the 'flat' parts of the sigmoid function and learning will be very slow. A neural network with 1 hidden layer is relatively shallow and it will work fine with 0.01. Deeper neural networks might need a different constant but it will still be a small value.

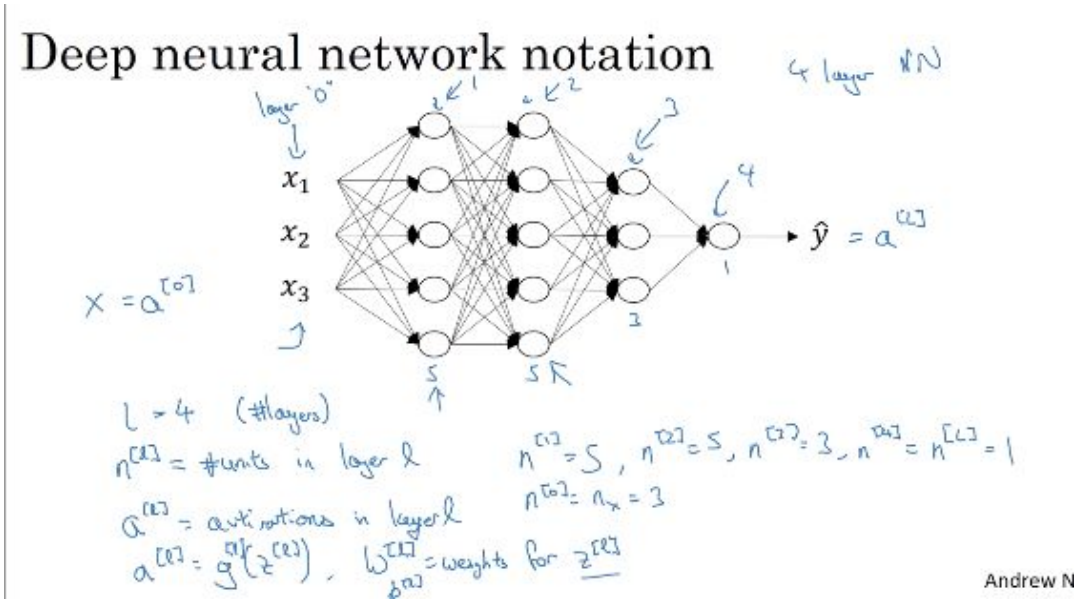
Week 4: Shallow Neural Networks

Deep L-Layer Neural Network

- What is a deep neural network? Below we have various neural network examples. Logistic regression is technically a 1 layer neural network and is considered a shallow NN. A 1 hidden layer neural network is a 2 layer NN since we don't count the input layer. A 5 hidden layer NN is considered deep

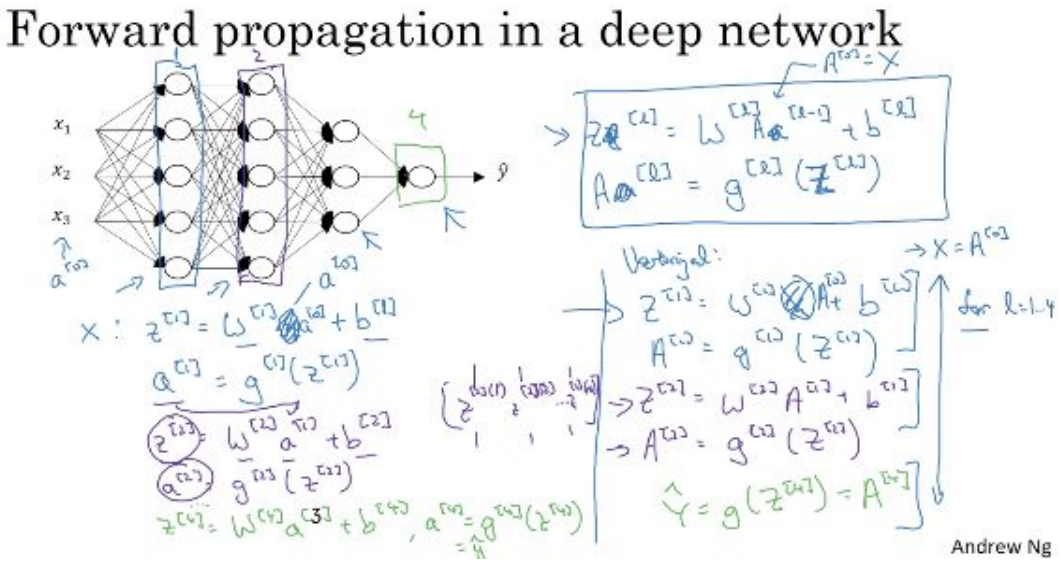


- Now let's take a look at the deep neural network notation. In the example below we have a 4 layer neural network and here's where we introduce some new notation:
  - L is the number of layers which in this case = 4.
  - $n^{[l]}$ is the number of units in layer l. For the example below they are: $n^{[1]} = 5,\ n^{[2]} = 5,\ n^{[3]} = 3,\ n^{[4]} = 1,\ n^{[0]} = n_x = 3$
  - $a^{[l]}$ is the activations on layer l
  - $a^{[l]} = g(z^{[l]})$
  - $w^{[l]}$ is the weights for layer $z^{[l]}$



Forward Propagation in a Deep Neural Network

- A lot to unpack on one slide. Using the neural network example below (where L=4). The un-vectorized implementation of forward propagation is
  - $z^{[1]} = w^{[1]}x + b^{[1]}$
  - $a^{[1]} = g^{[1]}(z^{[1]})$
  - $z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$ and so on until we get to
  - $z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$, $a^{[4]} = g^{[4]}(z^{[4]})$



The vectorized implementation of forward propagation is

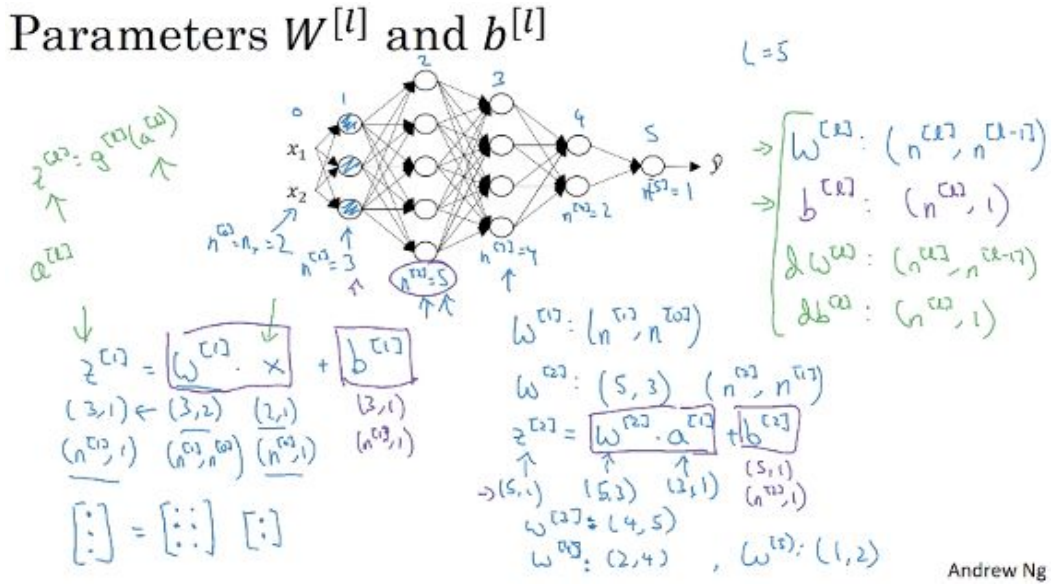
- $Z^{[1]} = W^{[1]}X + b^{[1]}$

- $A^{[1]} = g^{[1]}(Z^{[1]})$
- $Z^{[2]} = W^{[1]}A^{[1]} + b^{[2]}$
- $A^{[2]} = g^{[2]}(Z^{[2]})$
- $Y^* = g^{[4]}(Z^{[4]}) = A^{[4]}$

And we do this in a for-loop that goes from layer 1 to layer L

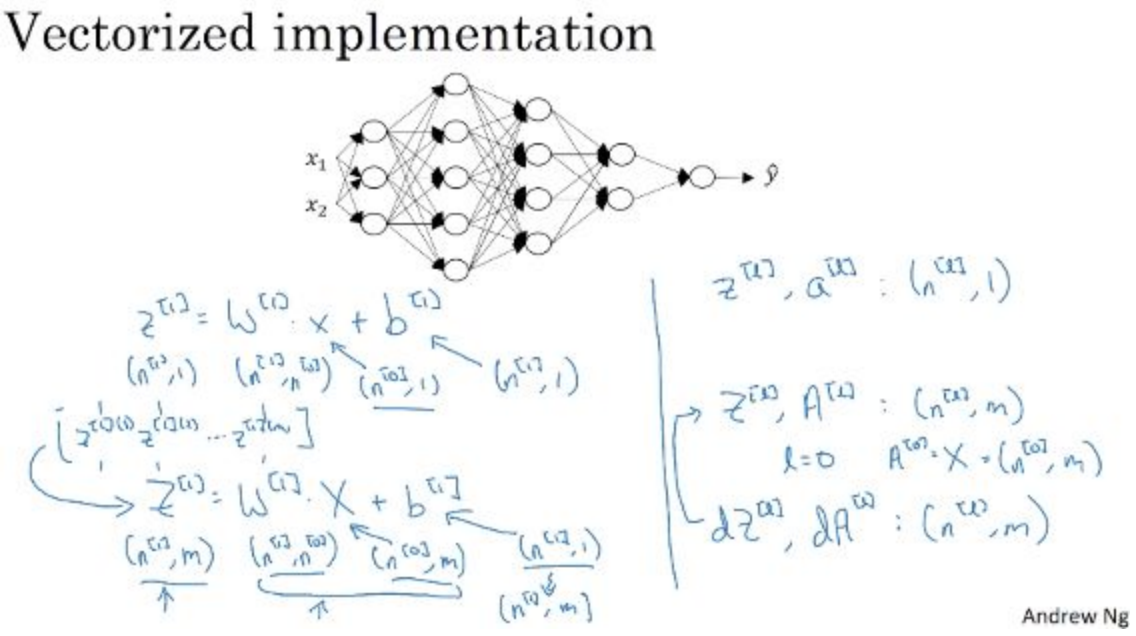
Getting Matrix Dimensions right

- Knowing matrix dimensions is a common debugging tool to make sure our code is correct. Let's take a look at what are the matrix dimensions for our parameters. In the example below we have a neural network with 5 layers. We remember $z^{[l]} = w^{[l]}X + b^{[l]}$ so we apply that equation to the first layer so we have $z^{[1]} = w^{[1]}x + b^{[1]}$. We also know $n^{[1]} = 3,\ n^{[2]} = 5,\ n^{[3]} = 4,\ n^{[4]} = 2,\ n^{[5]} = 1\ \&\ n^{[0]} = 2$. So for the first layer we know $z^{[1]} = w^{[1]}x + b^{[1]}$ so we know z is the vector of activations so for the first hidden layer $z^{[1]}$ has dimensions $(3,1)$ or more generally $(n^{[1]},1)$. We also know that X has dimensions $(2,1)$ or, more generally $(n^{[0]},1)$. Now we just need to know what $w^{[1]}$ will have and that would be dimensions $(3,2)$ because those are the only dimensions that allow our matrix multiplication to work as shown below for $z^{[1]}(3,1) = w^{[1]} * x(2,1)$ so $w^{[1]}$ has to have dimensions $(3,2)$ or more generally, $(n^{[l]}, n^{[l-1]})$. The rest of the relevant dimensions are:
  - $b^{[l]}:\ (n^{[l]},1)$
  - $dw^{[l]}:(n^{[l]},\ n^{[l-1]})$
  - $db^{[l]}:(n^{[l]},1)$



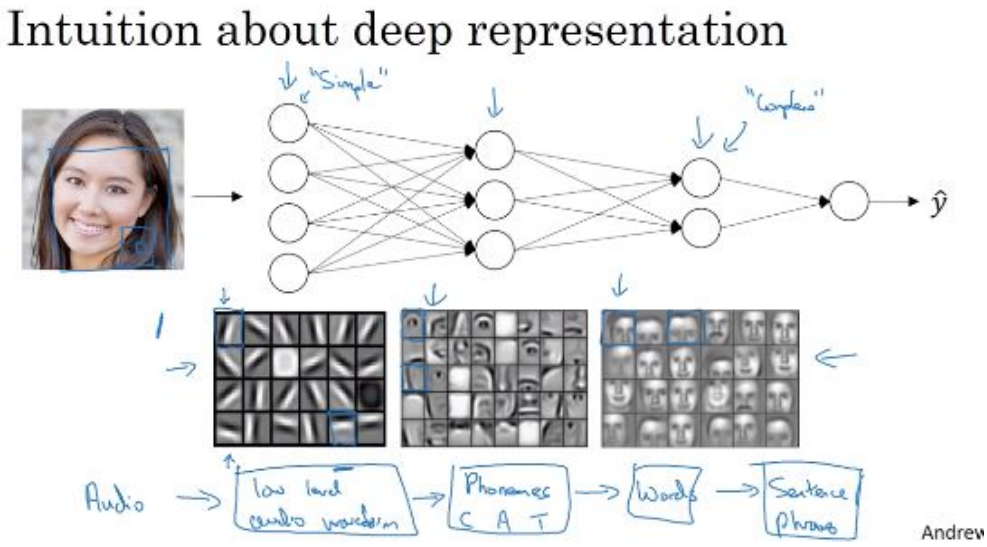
- Now let's take a look at the vectorized implementation for multiple training examples. In this case, the dimensions of $z, a\ \&\ x$ will change a little bit. Before we had: $z^{[1]}(n^{[1]},1) = w^{[1]}(n^{[1]}, n^{[0]}) * x(n^{[0]},1) + b^{[1]}(n^{[1]},1)$. The vectorized implementation would

be $Z(n^{[1]}, m) = W^{[1]}(n^{[1]}, n^{[0]}) * X(n^{[0]}, m) + b^{[1]}(n^{[1]}, 1)$. More generally, $Z^{[l]}, A^{[l]}$ have dimensions $(n^{[l]}, m)$ and correspondingly $dZ^{[l]}$, $dA^{[l]} : (n^{[l]}, m)$. One special case for X is when $X = 0$. In this case $X = A^{[0]} : (n^{[0]}, m)$



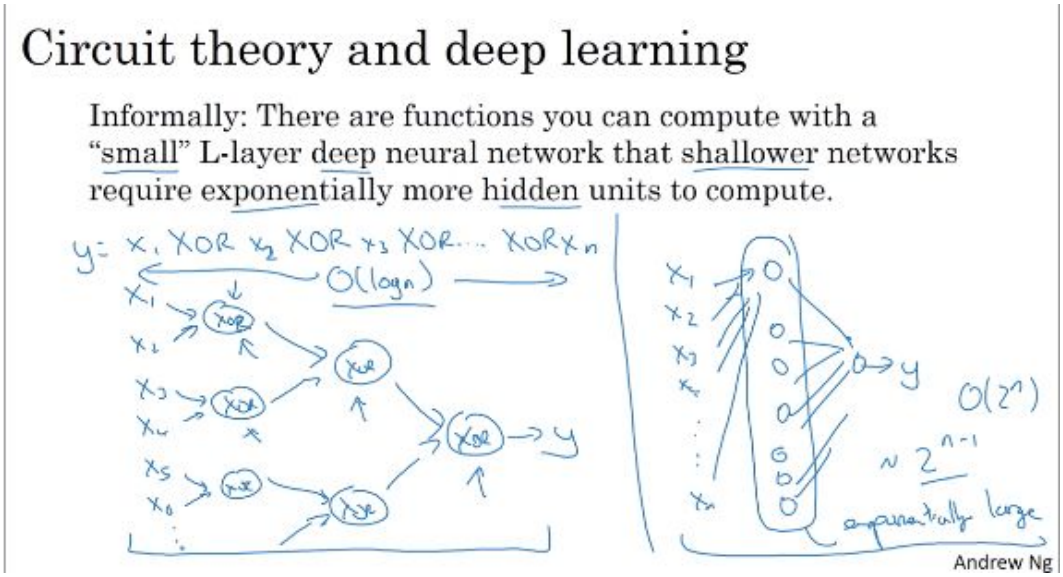
Why Deep Representations?

- Why do deep neural networks work well for a lot of problems? Let's take a look at why that is the case. Take the example of face recognition: a deep neural network starts by doing 'simple' computations like recognizing the edges on an image. The next layer takes these edges and groups them to form parts of faces. The last layer then takes the parts of a face to try to recognize different face types.



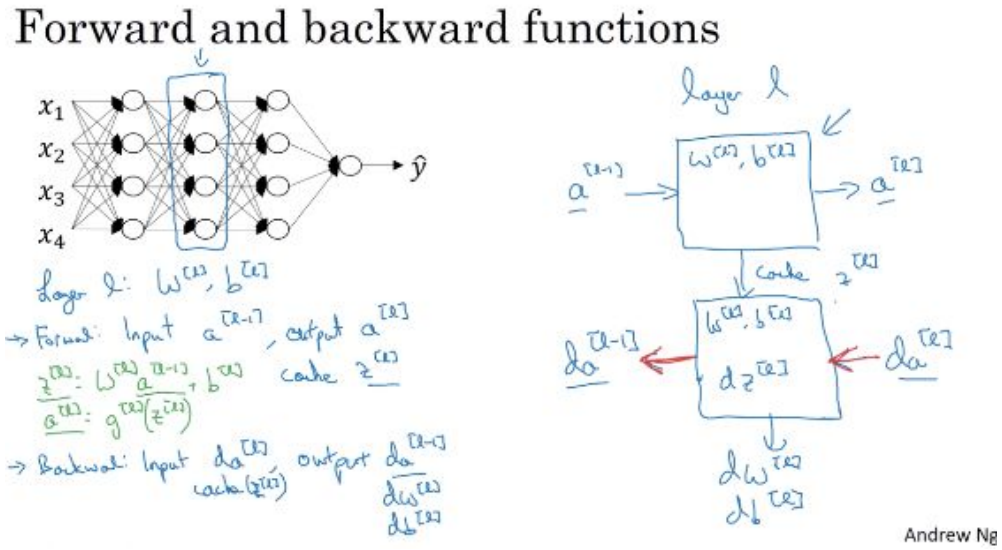
And that's how we end up with 'complex' functions in the later layers of neural network. The main intuition is that this simple-to-complex hierarchical applies to other types of data.

- The other intuition is you can compute functions with a 'small' L-layer deep neural network and shallower networks will require exponentially more hidden units to compute. The XOR example below shows how we have a small number of hidden units in the 'deep' neural network on the left while on the right we have a smaller neural network that has many, many more hidden units.



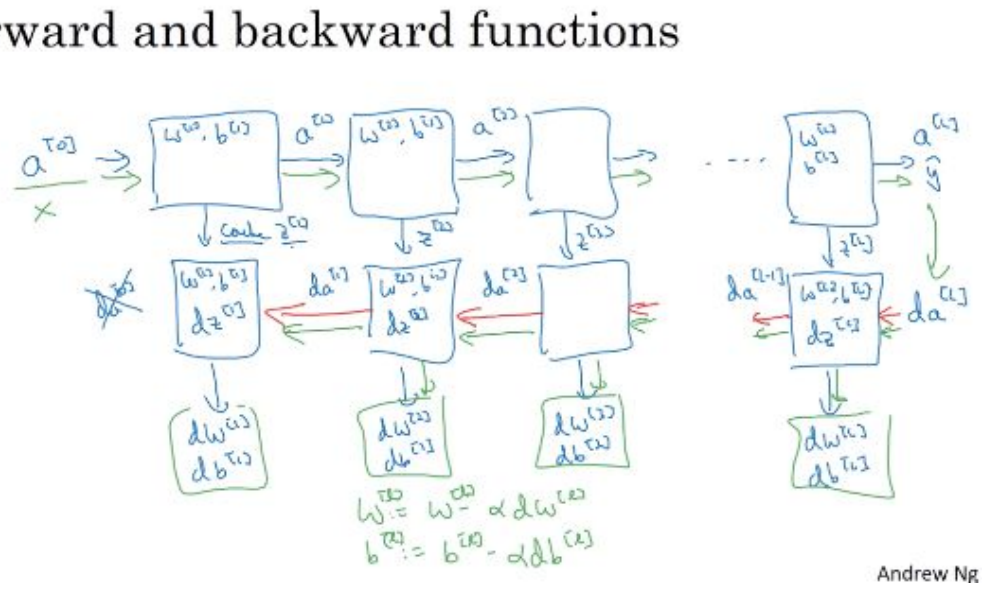
Building blocks of Deep Neural Networks

- In this session we are going to see how we can put in practice everything we have learned this week so we can build a deep neural network.



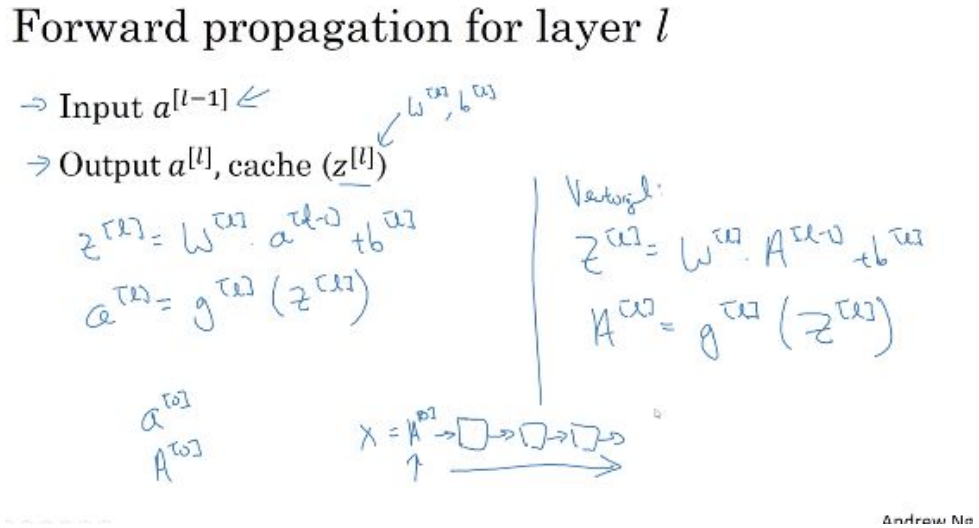
For forward propagation we have as input $a^{[l-1]}$ and as output $a^{[l]}$ and our equations are $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$, $a^{[l]} = g^{[l]}(z^{[l]})$. For backward propagation we have as input $da^{[l]}$ and as output $da^{[l-1]}$, $dw^{[l]}$, $db^{[l]}$. So we end up with two functions: forward propagation and back propagation

- We can use those 2 functions to then have one iteration of each one as shown below. For forward propagation, we start with input $a^{[0]}$, $use\ w^{[l]}\&\ b^{[l]}$, $cache\ z^{[l]}$ and output $a^{[l]}$ and so on until we get $a^{[L]}$, aka y^*. For backprop, we compute $da^{[L]}$ and use it as an input to the function. In the function we use $w^{[L]}\&\ b^{[L]}$ and compute $dz^{[L]}$, $cache\ dw^{[L]} and\ db^{[L]}$ and output $da^{[l-1]}$ and so on until we get $da^{[1]}, dw^{[1]} and\ db^{[1]}$



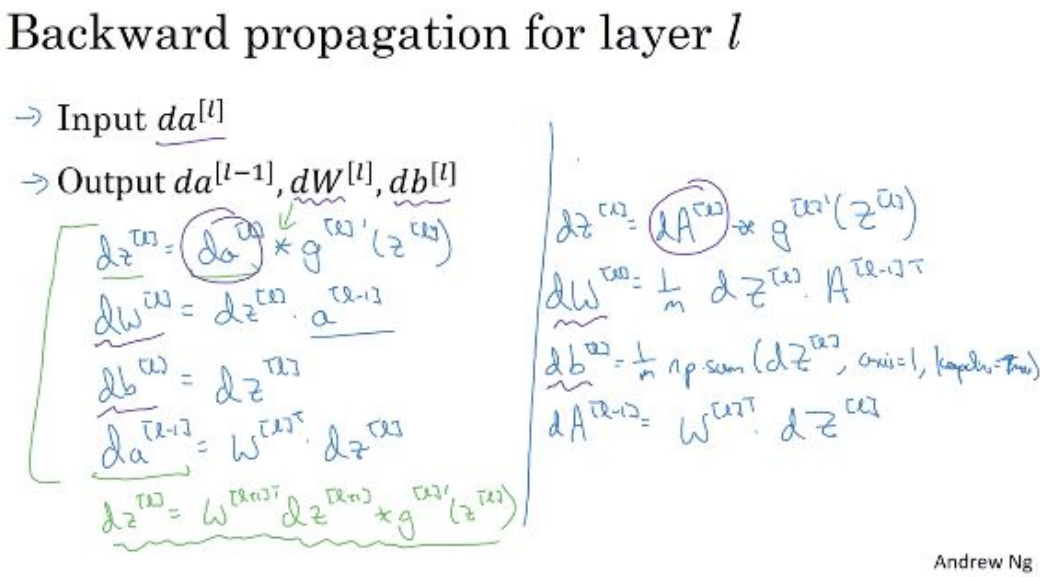
Forward and Backward Propagation

- Now let's see how we implement the previous steps.

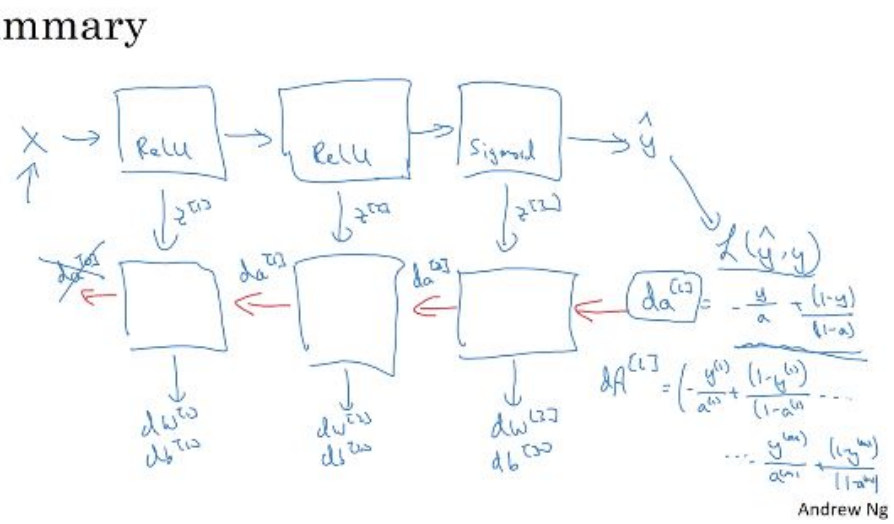


For forward propagation for one training example we looked at the following equations: $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]},\ a^{[l]} = g^{[l]}(z^{[l]})$. The vectorized implementation is $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]},\ A^{[l]} = g^{[l]}(Z^{[l]})$

- For backprop we have the equations below. We are just going to write down the vectorized implementation
  - $dZ^{[l]} = d\,A^{[l]} * (element - wise\ multiplication)\ g^{[l]}(Z^{[l]})$
  - $dW^{[l]} = (1/m)\ dZ^{[l]}A^{l-1]T}$
  - $db^{[l]} = 1/m\ np, sum(dZ^{[l]},\ axis = 1, keepdims = True)$
  - $dA^{[l-1]} = W^{[l]T}dZ^{[l]}$

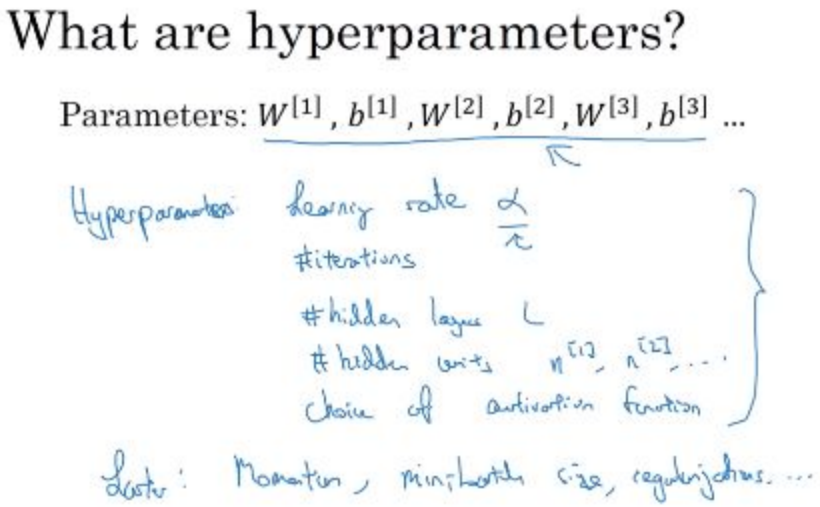


- Summarizing with a 3 hidden layer net , we start with X and we output y^* and we compute the loss function $L(y^*, y)$. We then compute $da^{[l]} = -(y/a) + (1-y)/(1-a)$. We use $da^{[l]}$ to start backprop until we end up with $da^{[1]}$. The vectorized version of $dA^{[l]}$ is equal to $(y^{[l]}/a^{[l]} + (1-y^{[l]})/(1-a^{[l]}) \ldots . y^{[m]}/a^{[m[} + (1-y^{[m]})/(1-a^{[m]})$

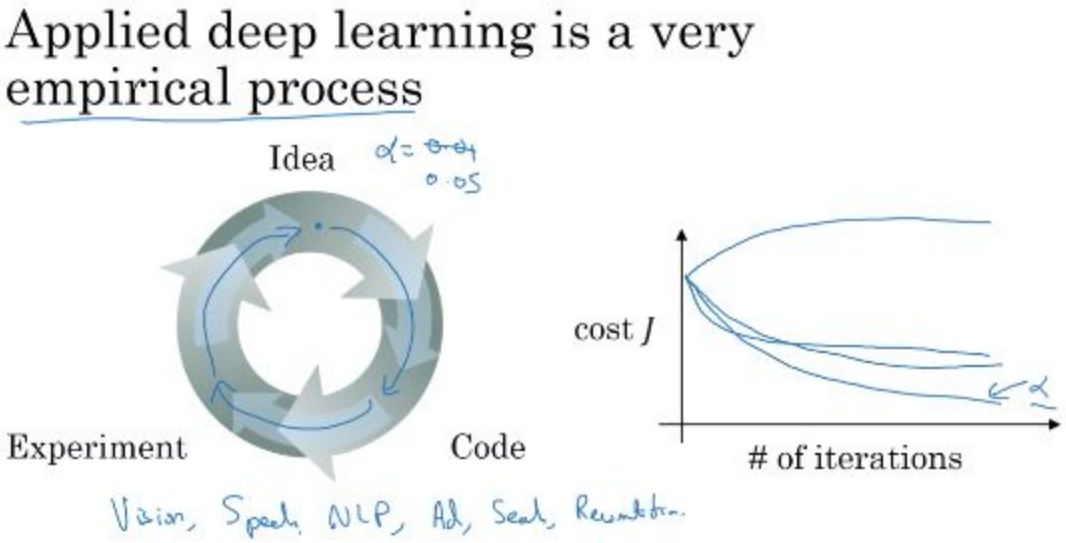


Parameters and Hyperparameters

- Being effective in developing neural nets means we organize the parameters and the hyperparameters. Hyperparameters are:
  $learning\ rate\ \alpha,\ \#\ of\ iterations,\ \#\ of\ hidden\ layers\ L,\ \#\ of\ hidden\ units\ ,\ choice\ of\ activation\ function.$
  These parameters control our parameters, which are $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, ..., W^{[m]}, b^{[m]}$.
  There are other hyperparameters that we will learn in a future course.



- Applied deep learning involves trying out new ideas and see how they pan out. That means we try out different hyperparameters values and see how they perform.



What does this have to do with the brain?

- Below we have the forward and backward propagations equations. The analogy with the brain is that brain neurons receive inputs from other neurons, do some processing and send electrical signals (the output) to other neurons, similar to the neural net process.