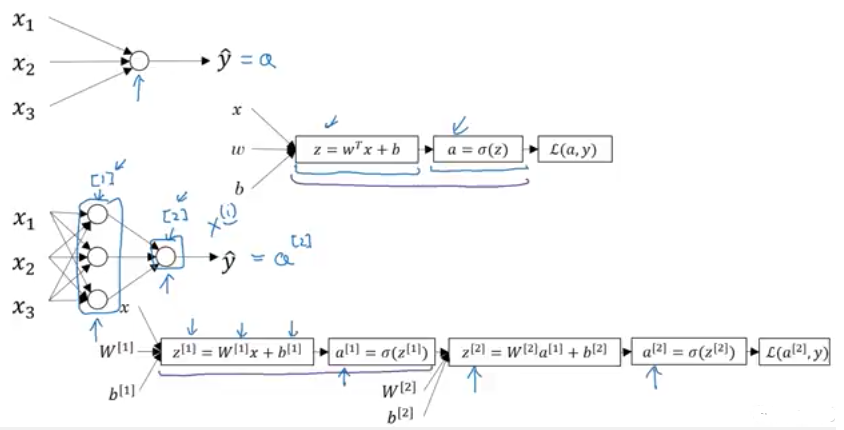
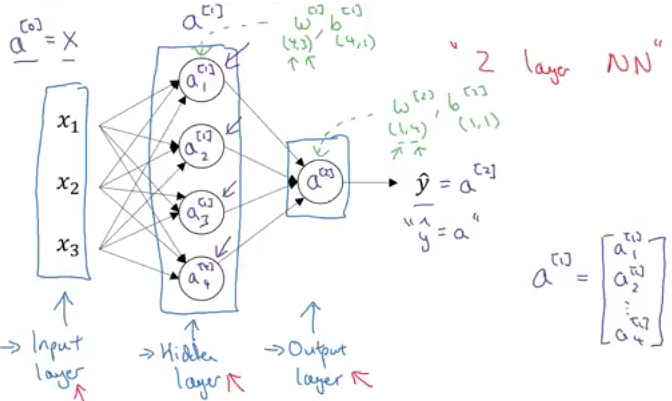
**Lec 23: Neural Network Overview**

Here we are representing logistic regression in each of the two layers:



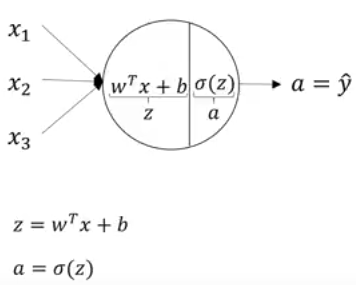
Backpropagation for derivative calculation is also computed in similar way.

**Lec 24: Neural Network Representation**

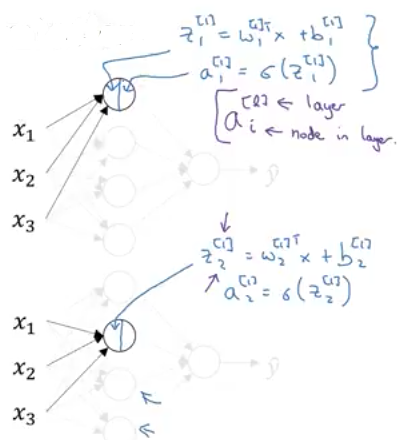


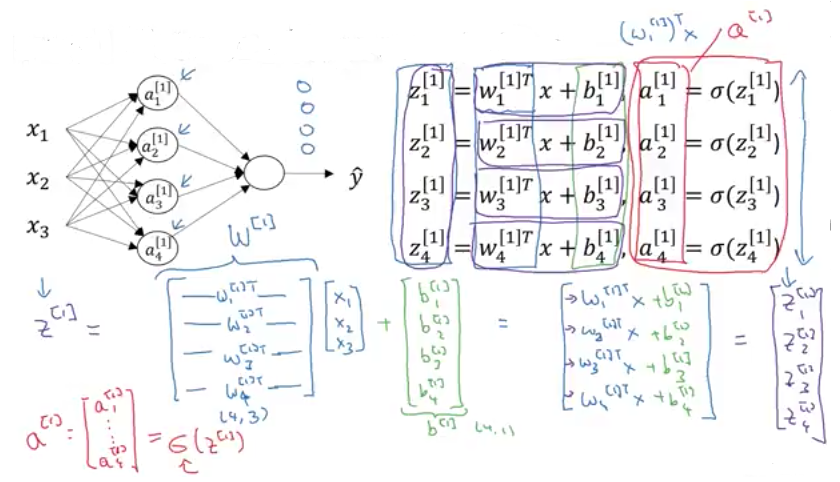
**Lec 25: Computing a Neural Network’s Output**

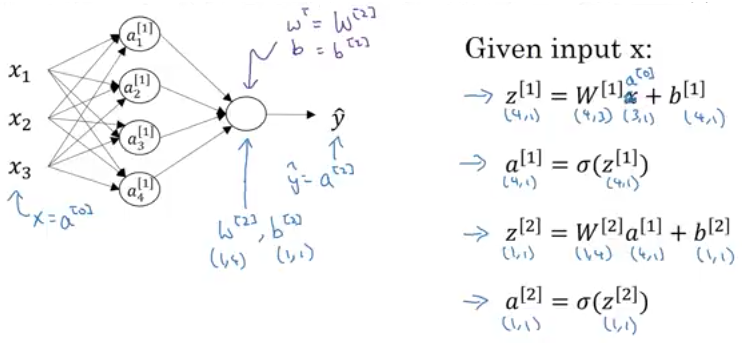
Each neuron in the hidden layer computes logistic regression, first it find (z) then (a):



Below is the formulation for logistic regression of the first two nodes of the first layer (hidden layer):

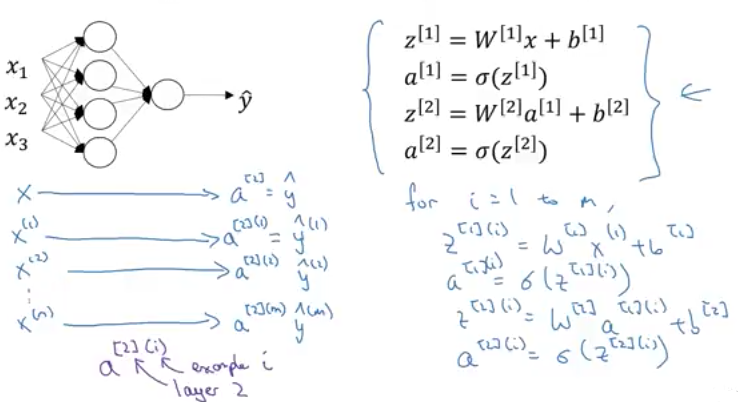




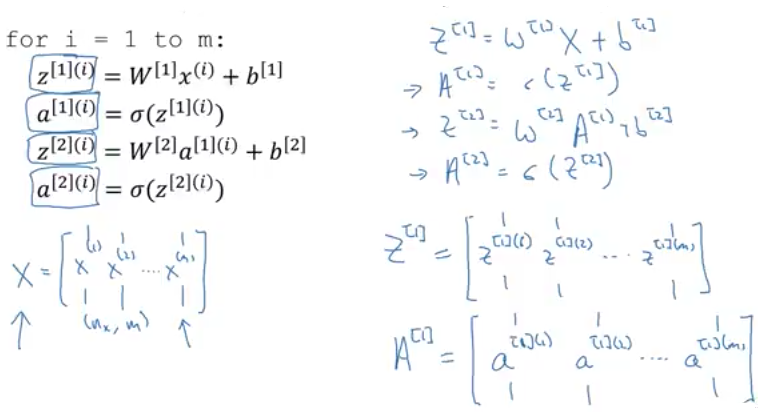


**Lec 26: Vectorizing Across Multiple Training Examples**

The ‘for’ loop implementation across all ‘m’ training examples:



Replacing the ‘for’ loop implementation by vectorizing:



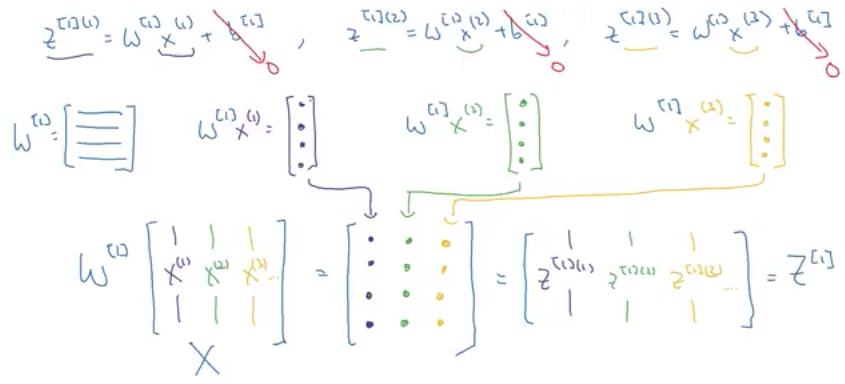
*Point to note:*

In the matrix ‘A’ and ‘Z’, the top left element corresponds to the activation result of the first training example of the first hidden layer.

Likewise, horizontally across we move from one to the next training example. Vertically downwards we move from one activation neuron of one layer to next neuron of the next layer.

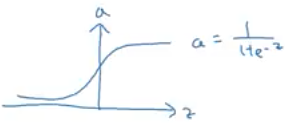
**Lec 27: Vectorized Implementation Explanation**

Justification of vectorization:

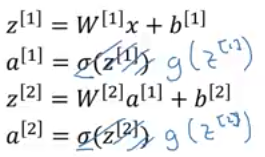


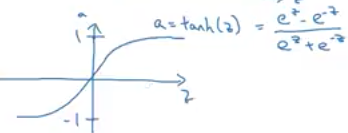
**Lec 28: Activation Functions**

There are many activation functions available for hidden layers and output layer.

Till now we saw the **sigmoid** activation function: 

So a general activation function can be represented by g(z) in the following:

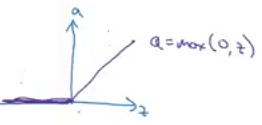


A good activation is the **tanh** function whose values are between [-1, 1]: 

This function is better because the mean of the activations that out of the hidden layer are closer to having 0 mean (centering the data). This makes learning for next layer easier.

But for output layer it makes sense to have [0, 1], in which we would use the sigmoid function.

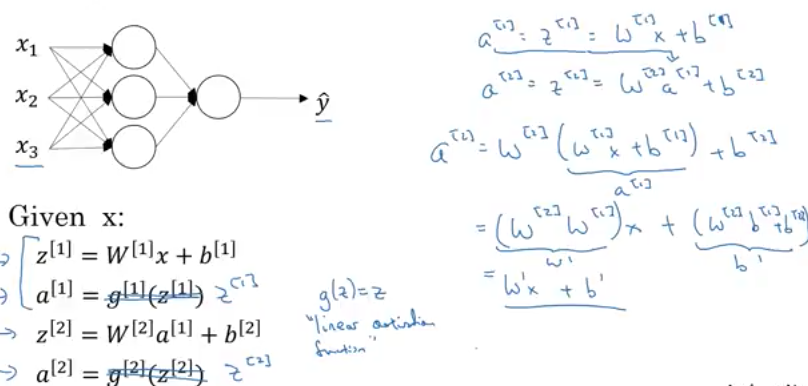
The downside of the both **tanh** and **sigmoid** is that if ‘z’ is extremely large or extremely small, then the gradient (derivative) of those functions would be very small, close to zero. This can greatly slow down gradient descent.

The other choice is **ReLu:** 

One disadvantage here is that for negative values of ‘z’, the gradient is 0.

**Lec 29: Why Non-Linear Activation Function?**

Let us first see what happens when you substitute a linear function for a non-linear function?



So performing a sequence of linear functions even for a deep network only results in a linear function, which is in itself useless. Throwing a non-linear function only brings out the expressiveness of the network.

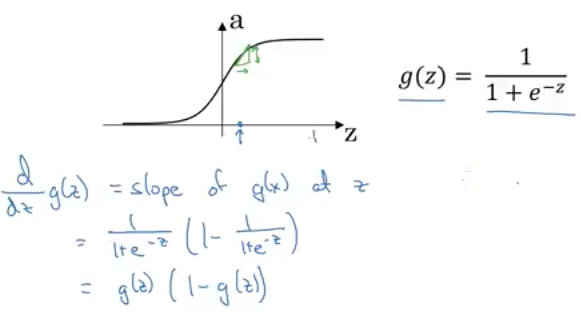
One place where linear activation functions work is in the output layer for a regression problem where the output is a real number.

**Lec 30: Derivatives of Activation Functions**

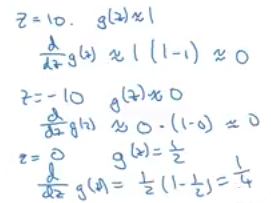
In order to perform backpropagation in neural networks, the derivatives of different activation layers is required.

**Sigmoid activation function:**

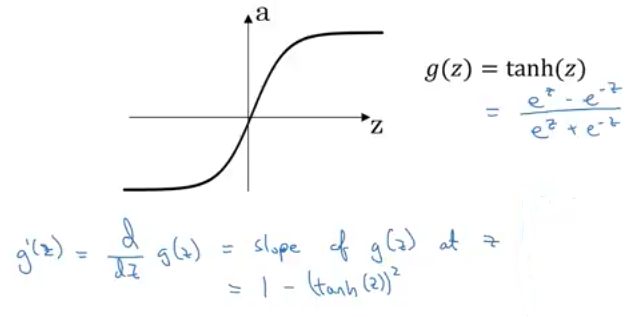
The derivative is a s follows:



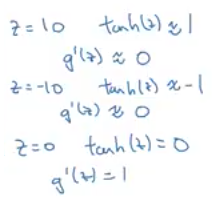
Let us see some cases:



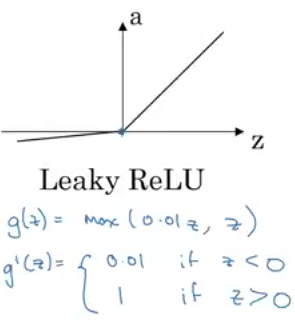
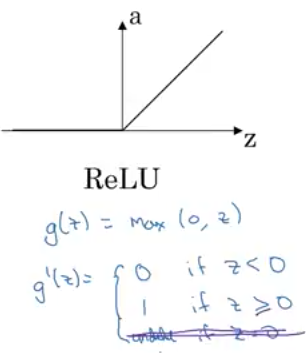
**tanh function:**



Let us see some cases:

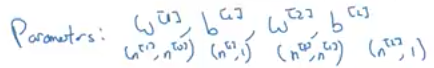


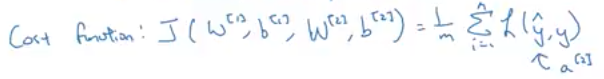
**ReLU and Leaky ReLU:**



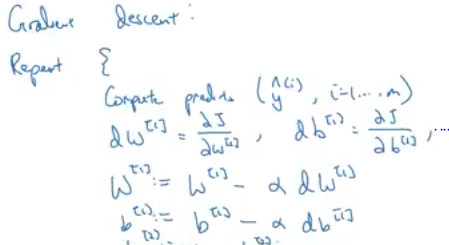
**Lec 31: Gradient Descent for Neural Networks**

We will work out gradient descent for a neural network with one hidden layer.

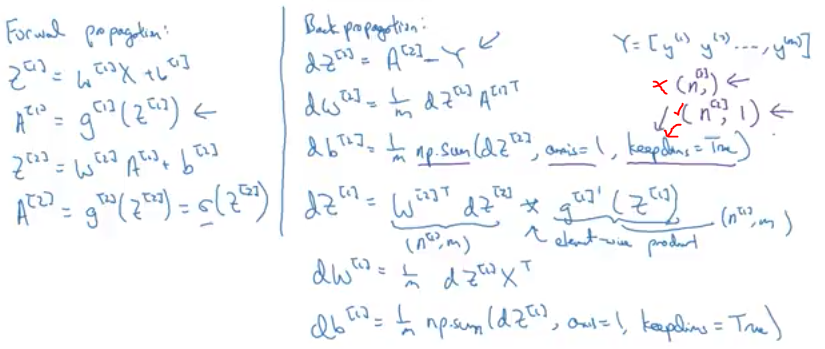
The following are the parameters for the same: 

The cost function: 

Before training initialize the parameters to some random values rather than 0. Then we perform gradient descent:

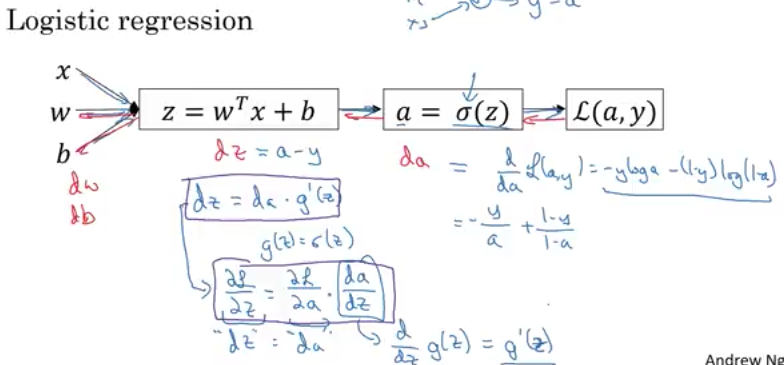


Formulas used for computing derivatives:

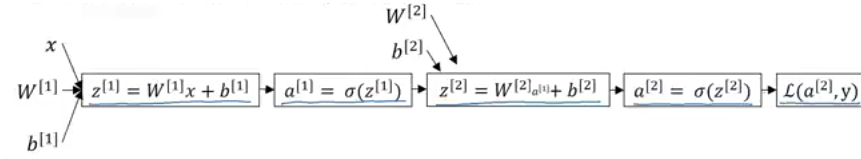


**Lec 32: Backpropagation Intuition**

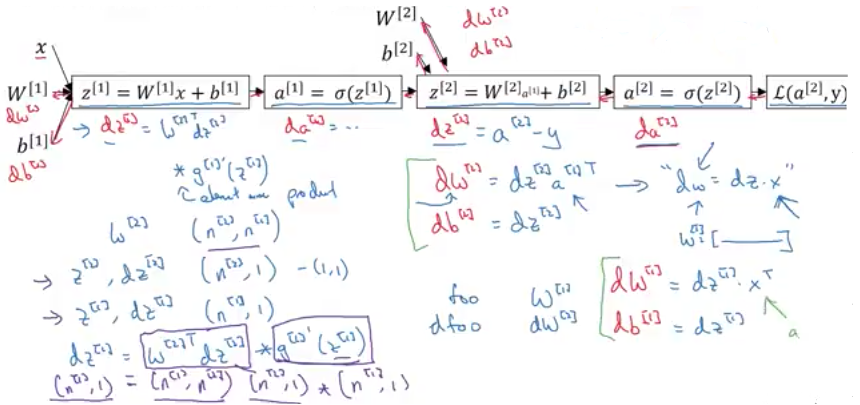
The following is the scenario of backpropagation in logistic regression:



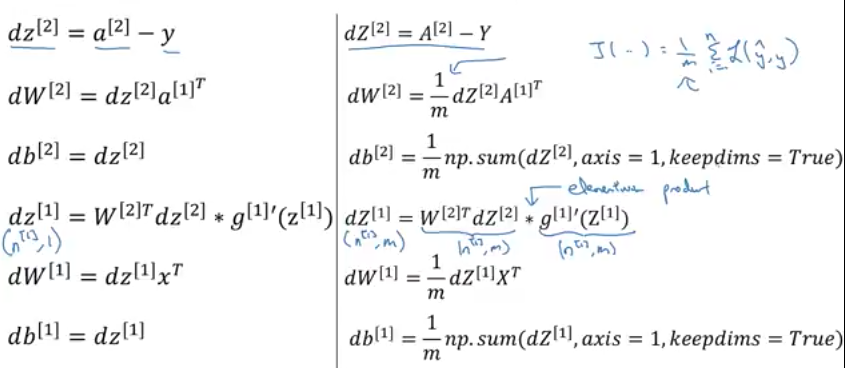
Now let us look at backpropagation in a neural network with two layers (one hidden and one output layer):



The following is backpropagation when there is only one training example:

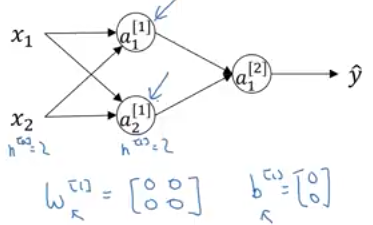


The following is backpropagation for ‘m’ training examples (right-hand column):



**Lec 33: Random Initialization of Weights**

Initialization of weights to zero in a logistic regression to zero is alright, but for a neural network it won’t work.

Consider the following NN where there are two input features and two hidden units. The W1 matrix is a 2x2 matrix. Let us assume that weight and bias are initialized to zero.

When the weights are assigned to zero the activations are the same, performing the same function

Moreover, during backpropagation the terms  are also the same.

Hence when weights are all initialized to 0, all the activations are symmetric computing the same function.

The key to this is selecting random weights. For the bias term it is not a problem to have 0 since it does not have the ‘symmetry breaking’ problem.

Now why should we choose a very small random weight rather than a large number? It turns out that when using ***sigmoid*** or ***tanh*** activation functions, for a large weight you end up on the flat portion of the curve:



Hence the gradient descent would be very slow and the learning would also be slow.