Brief overviews about what we discovered when learning about backward propagation on a neural network

1st of all this is where I trained my model: https://www.kaggle.com/code/emilioyared/lufanarprojectmnist

1. Very simple example :

* X will be our input layer
* A will be our hidden layer
* will be our output prediction layer

We get Z = \*X +

Z = \* +

Each row in the weight matrix represent all the weights linked from the previous layer to row-th node in the activation layer, in general if the input layer was N\*1 and the hidden layer was M\*1, the weight will be M\*N always because each row should represent all the weights of the current node of the previous layer, and we have M nodes

Z =

We get A by applying a nonlinear called activation to Z, like sigmoid, softmax, ReLU….

A = ReLU(Z)

We will do the same thing with O

O = f ( \*A + ) f:some non-linear function

1. More Complex Neural Networks:

we can create a more complex neural network by increasing the number of hidden layers and the number of nodes, generally the same rules apply, we have weights and biases between any two layers and an activation function, and to pass from one layer to another we apply the same formula as before.

1. Back Propagation:

Here we will see the prediction of our model and the desired output and through some error/cost function, we will do some math to fine-tune the weights and biases such that our model does less errors

Our whole network is just one big function that has weights and biases as parameters/inputs and outputs also the output layer as output, we add a layer of complexity to that by creating the cost function, we want to change the weights and biases in a way that the error function is minimal,

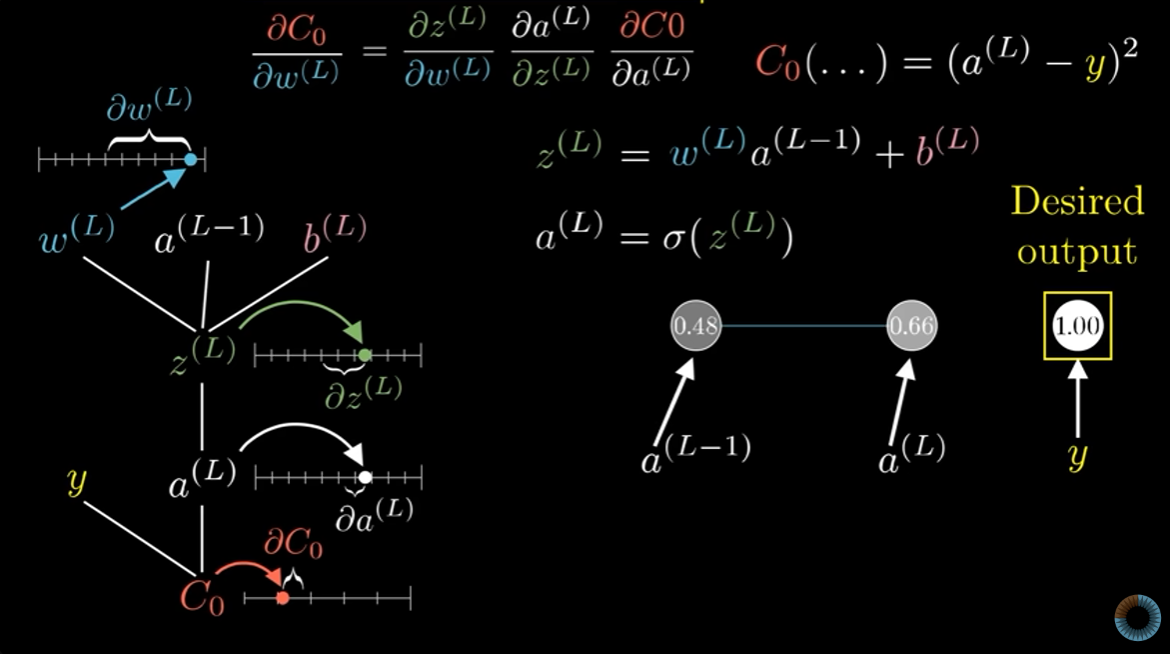
Let’s take the example of 1dimensional function f(x) to find the minimum what we do is we find its derivative, in computational terms, we take a random place in the input space, we check the slope of the function and we take steps to know where the local minimum is (it is not guaranteed to find the absolute minimum), we take our steps proportional to the slope.

now the example of a function with a lot of inputs as parameters, we will be asking the question which n-d vector direction should I take such that I find the local minimum of the function.

The gradient of any higher dimensional function will give us the direction of steepest ascent, so we take the opposite, and we get our local minimum

So this will tell us how we should change our weights and biases such that we minimize the error function (gradient descent).

In other words, the gradient of a function just tells us how sensitive a small nudge in the input is to the change to the output, so what we need to find is the partial derivative with respect to weights and with respect to biases.



Credit: 3Blue1Brown Youtube Channel, Neural Network playlist

same thing applies for the bias, now let’s compute the 1st regression formula for my model which has an input of 784 nodes two hidden layers of 16 nodes and 1 output layer of 10 nodes

dZ3 = A3 - one\_hot\_Y # Gradient of the cost with respect to Z3

dW3 = 1 / m \* dZ3.dot(A2.T) # Gradient of the cost with respect to W3

db3 = 1 / m \* np.sum(dZ3) # Gradient of the cost with respect to b3

the 1/m comes from doing it in batches instead of one by one so we are averaging out the cost function