NEURAL NETWORK

ARIHARASUDHAN

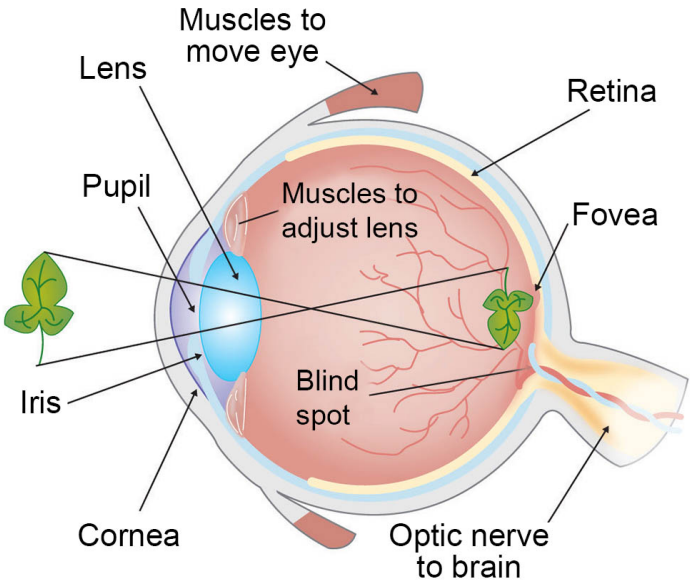
★ THE INTRODUCTION TO ANN

Let’s observe the following image. We are given with two Tamil texts. The first one is written neatly (a bit). The second one is sloppily written. Aren’t they ?

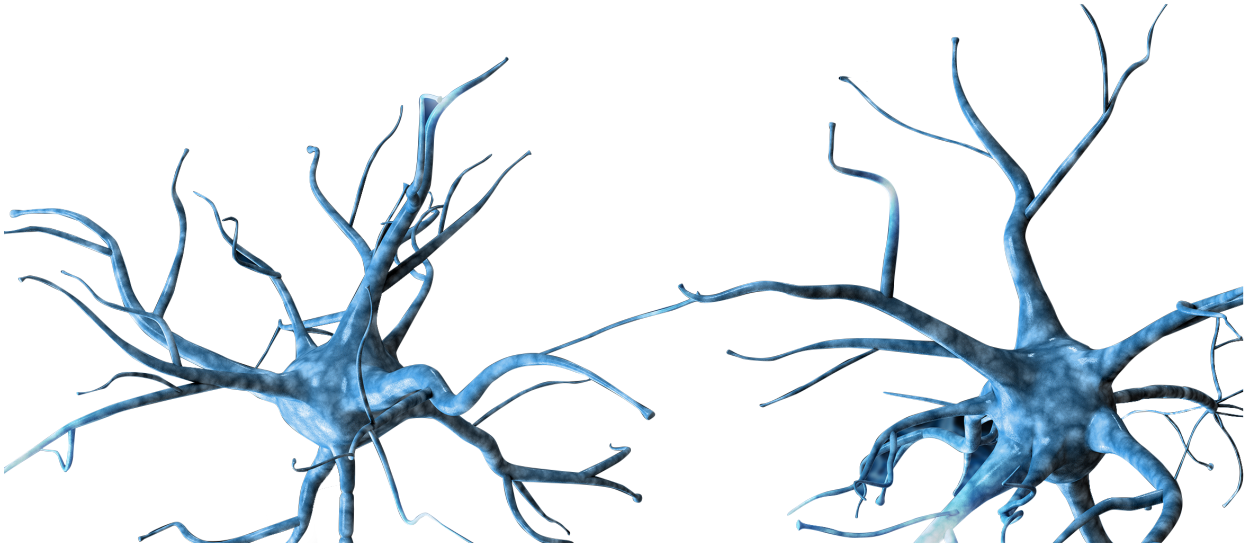
It’s easy to understand the first one. And it is really 99.99% possible to understand the second one too. But, how ? How can we understand it although it is sloppily written ? Because, we are humans. Similarly, does your brain have trouble recognizing the following low-resolution image of letter three ?

Absolutely, there’s no trouble recognizing three here. Isn’t it ?

We can easily identify the three in this image though it is written sloppily. I think there is no need to go premium in LinkedIn to view who viewed your profile. I hope we can identify it by looking closely , comparing with DP and so on. How crazy is this that the brain can do these all so effortlessly ? The particular light sensitive cells in our eyes are fired differently when we look at the first and the second Tamil texts shown above.



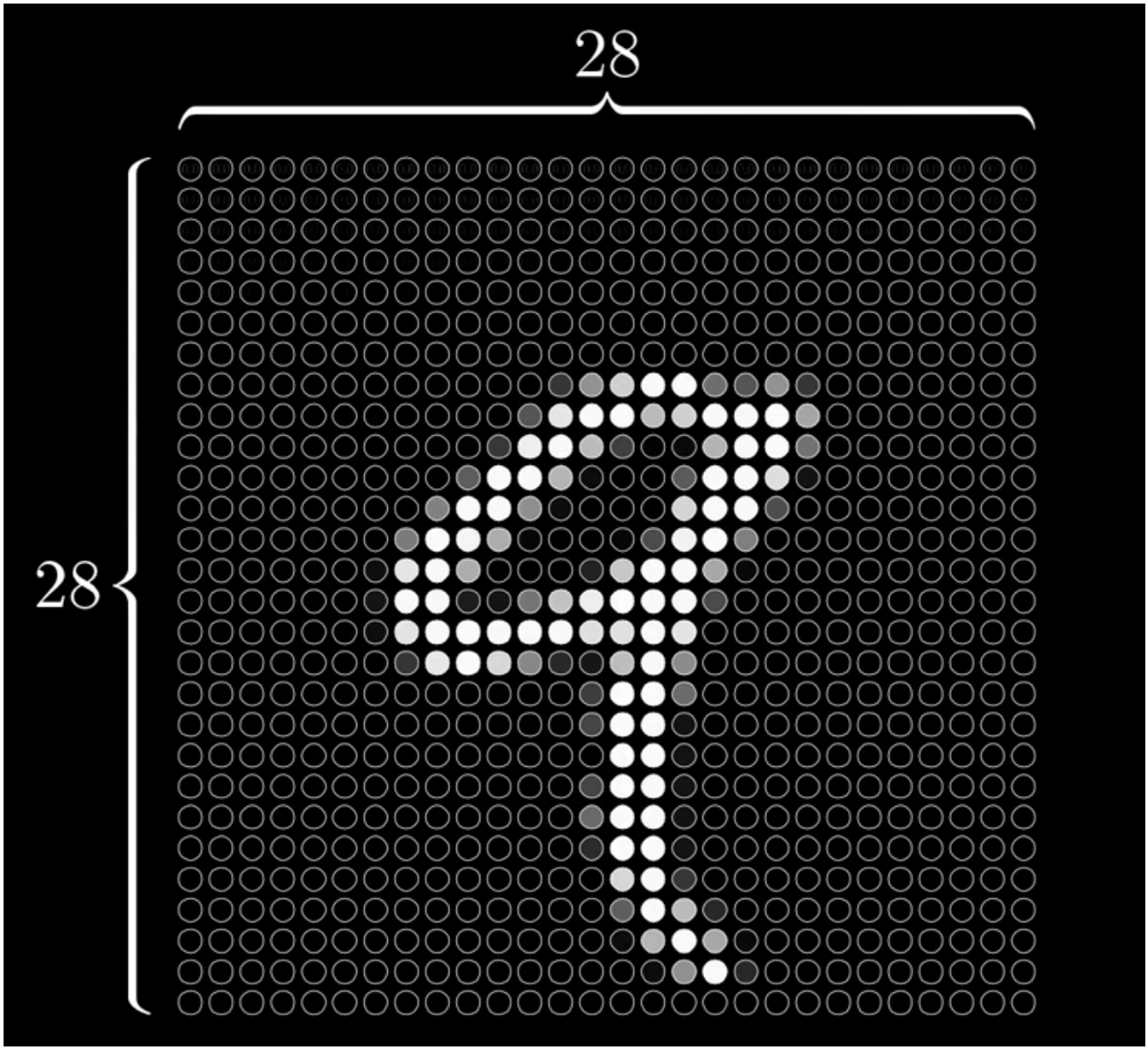
The concept of Neural Network is inspired from The Human Brain. As all of us know, Network is simply a connection / links. To understand it clearly, we need to know of Neurons and in what sense they are linked together!



Ah well! Neurons are information messengers. They use electrical impulses and chemical signals to transmit information between different areas of the brain, and between the brain and the rest of the nervous system. They are the fundamental units of the brain and nervous system, responsible for receiving sensory input from the external world and for sending motor commands to our muscles.

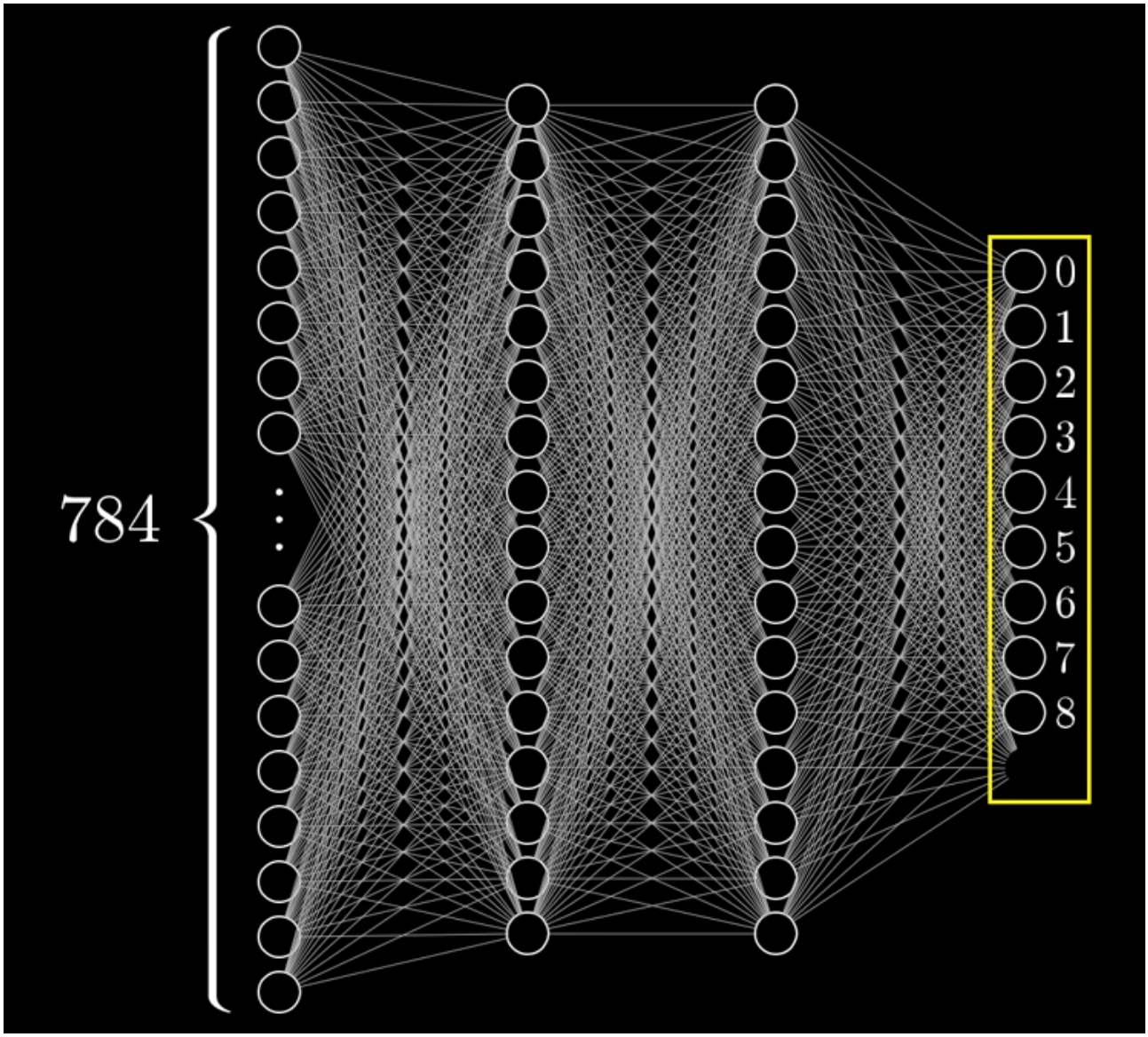
★ TECHNICALLY SPEAKING

Can you guess the number shown below ?

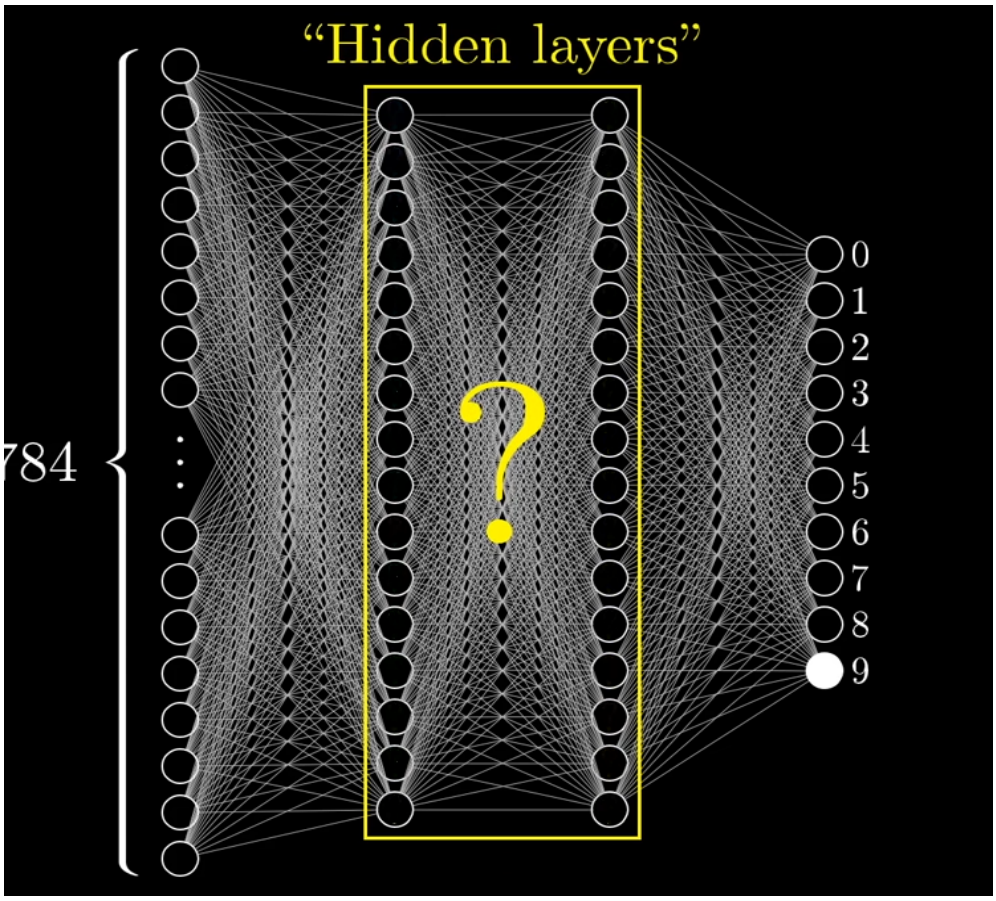


It’s nine indeed. To understand a neuron technically, let’s consider the above. The shown image has the width and height of 28pixels. It means , there are totally 28 times 28 = 784 pixels in this image. We represent each with a circle.

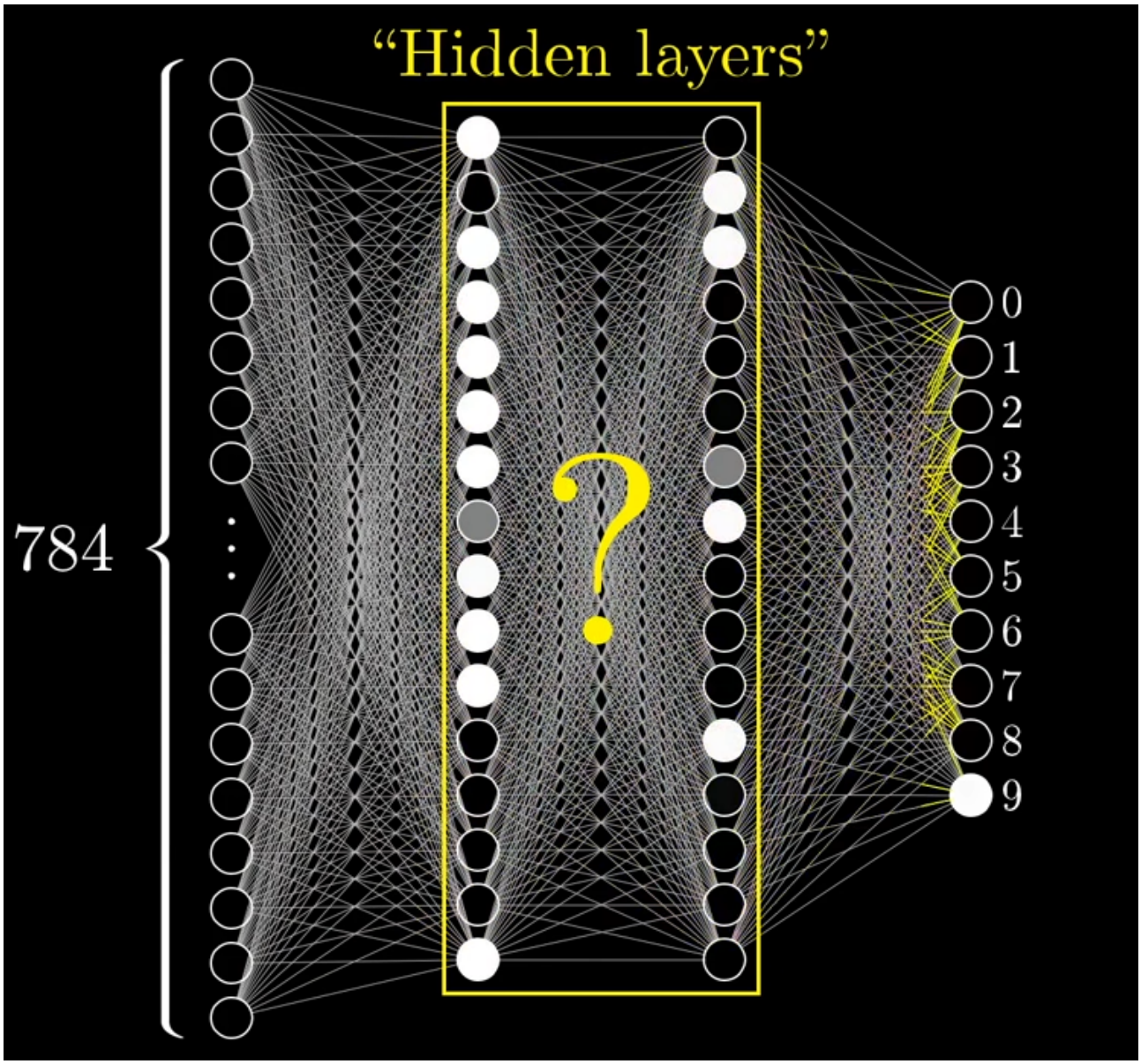
Now, we are about to call one circle, a Neuron. So, there are 784 neurons in total. These neurons represent the gray-scale value for each pixel. [Ranging from 0 for pure Black and 1 for pure White] The number inside the neuron is called it's ACTIVATION. The neurons with high activation number are lit up.



All these 784 neurons makes the first layer of the neural network. The count in the first layer is reduced while heading towards the last layer. So, we can observe a Multilayered Complex Structure here. The final layer is our output indeed. The last neuron of the last layer talks of the confidence level of the output. We have tasted the input layer and the output layer. What about the layers in between ? What are they supposed to do ? As we guess, they are the core part! They does what to be done! They are the so-called Hidden Layers!

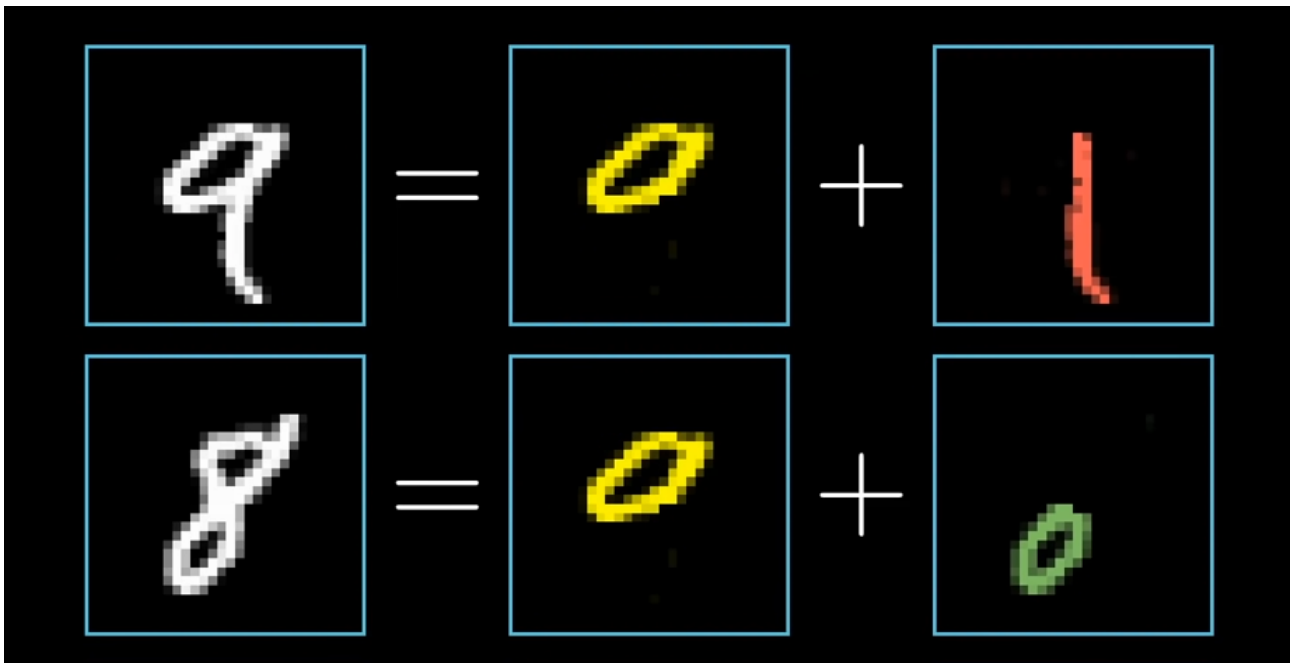


As we can see, there are two hidden layers in our neural network and 16 neurons each. We clearly know that there are activations in each layer. Important to remember is that the activation in one layer determines the activation in the next layer and so on.

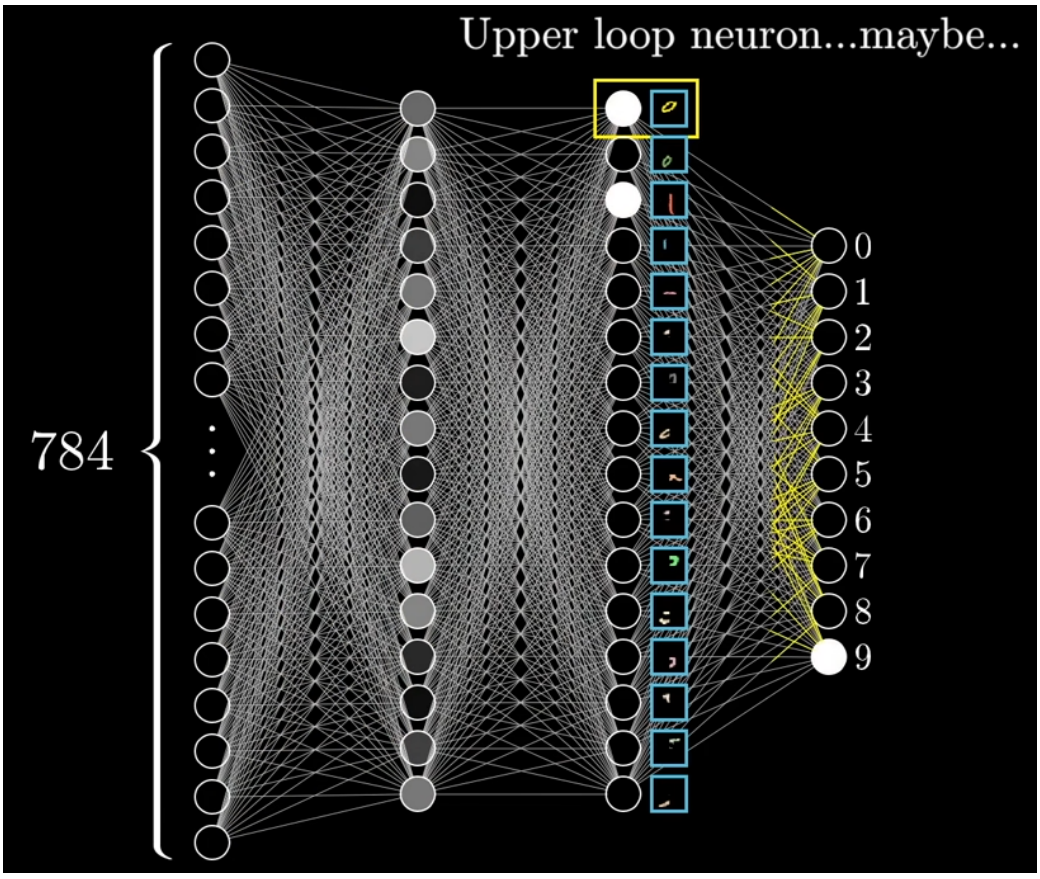


THINK : Does the activation in each layer seem like a filtering process ?

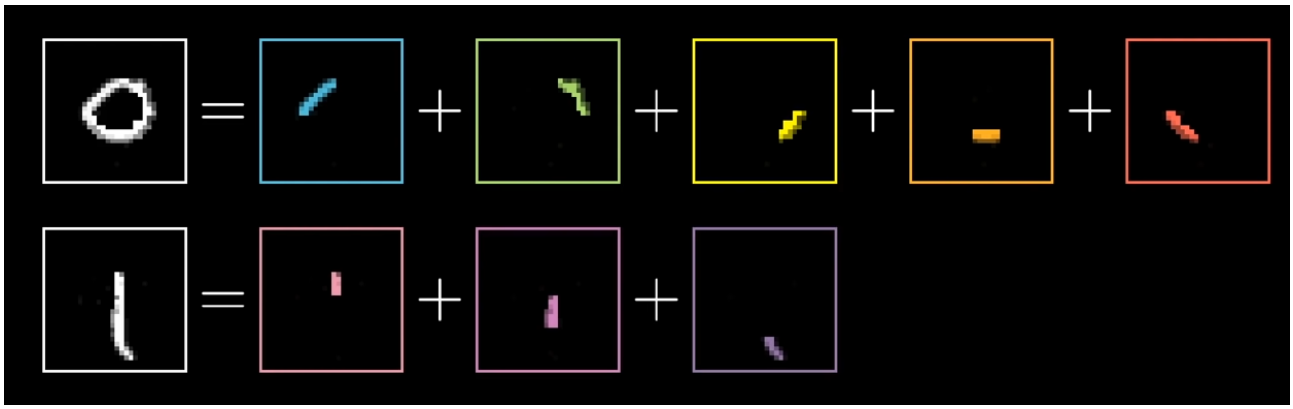
Why layers ? When we look on a number, we piece together the various components a number has. NINE has a loop and a line. EIGHT has two loops.



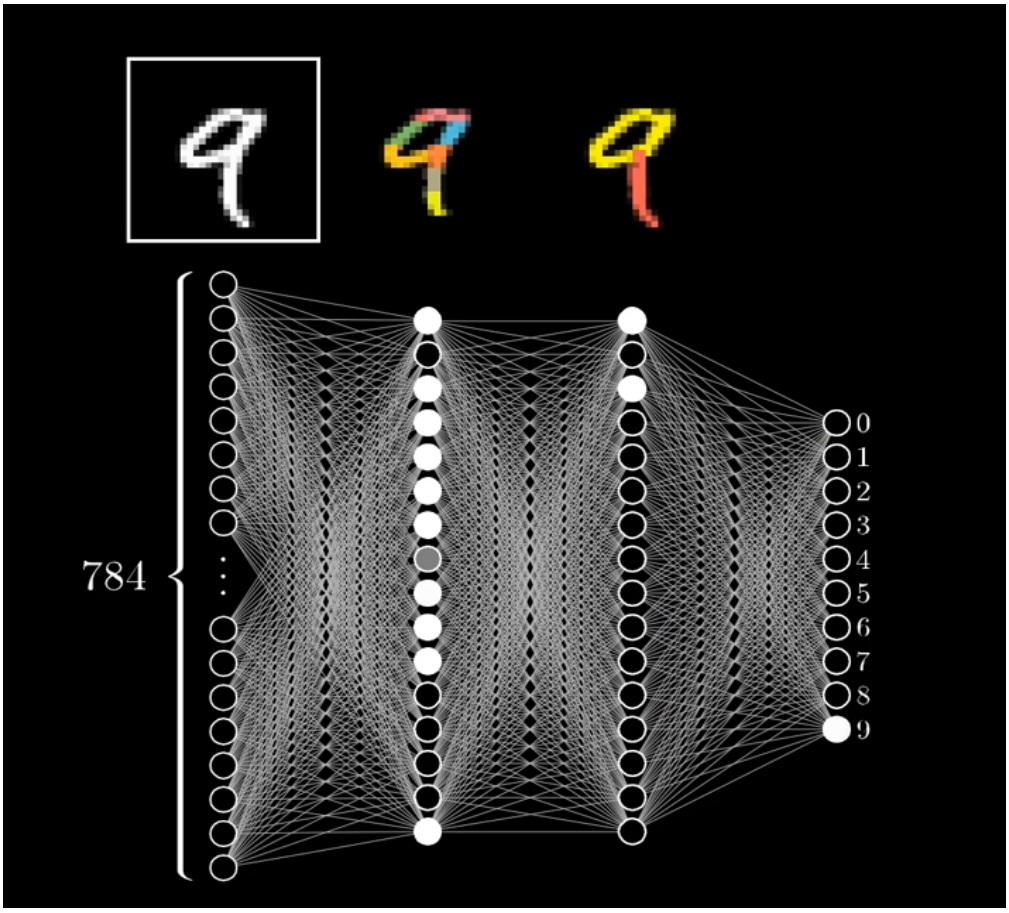
How can we recognize 9 that has two sub-components a loop and a line ?



Here, the way the neurons are activated is the one and only important stuff by means of which the appropriate number is determined. We have to recognize the components. But, how ? A component itself has a number of sub-components.



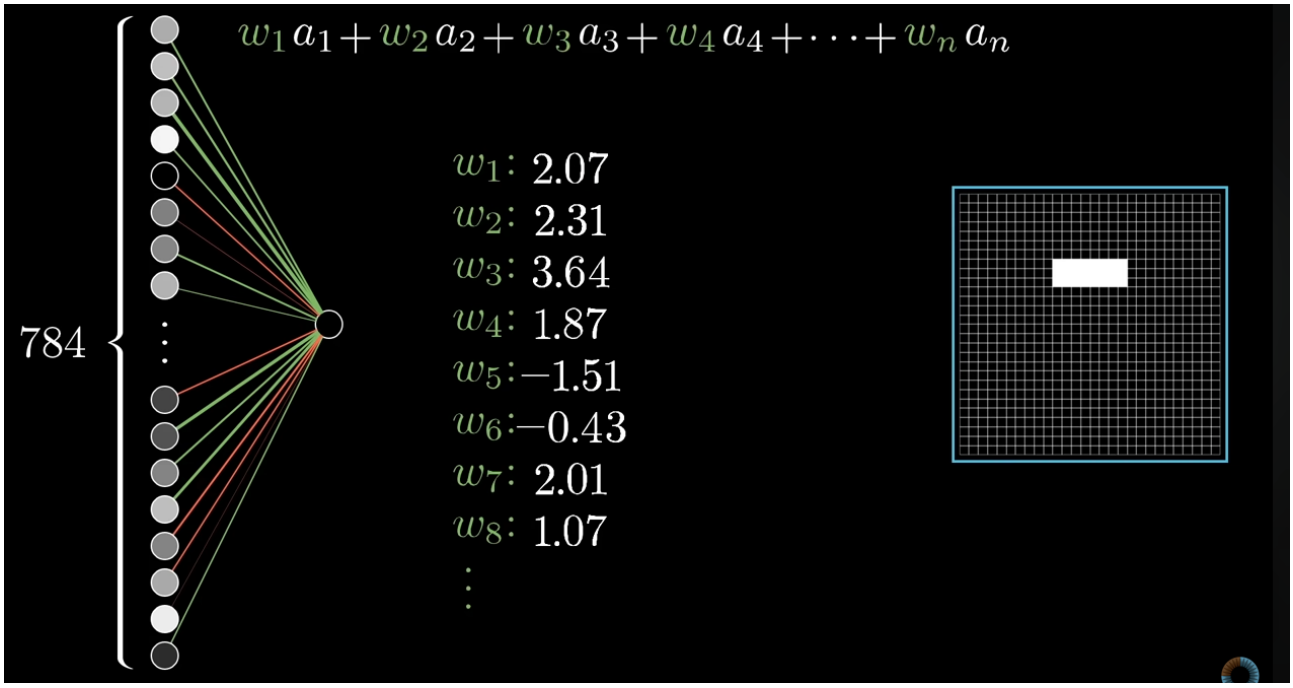
So again, The Activation in the one layer defines that in the next layer.



★ NO NEED TO BEAT ABOUT THE BUSH

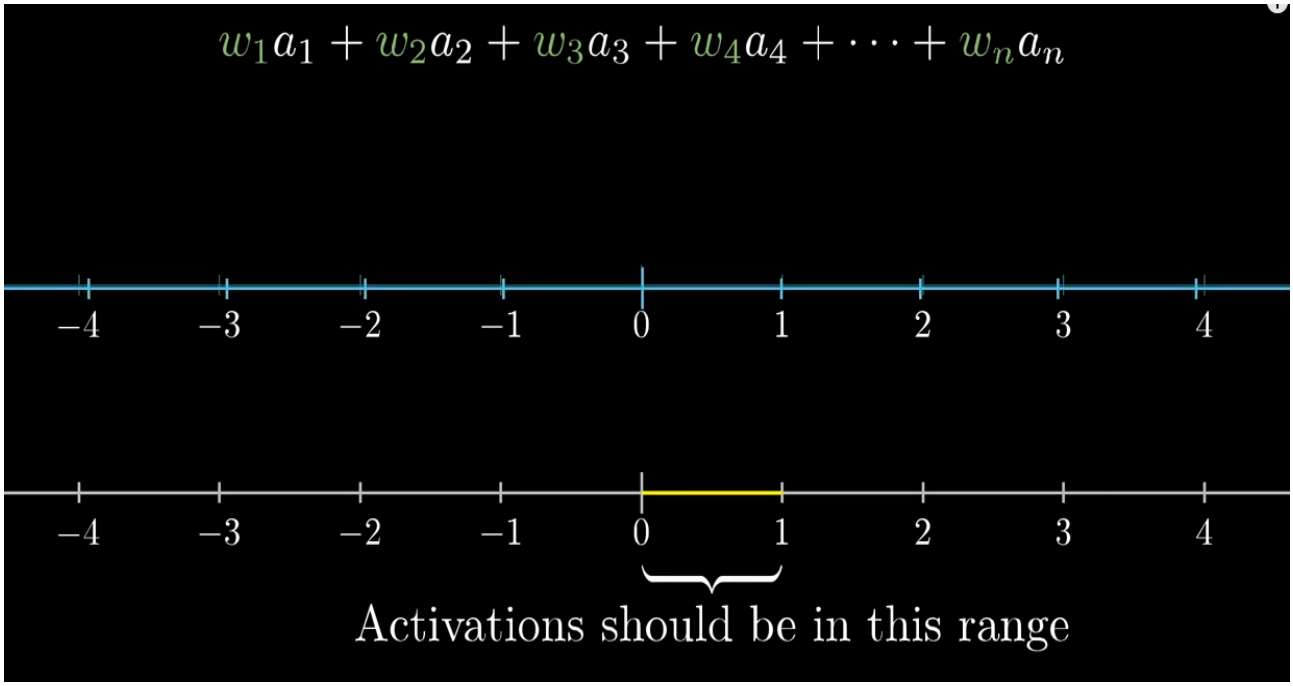
How to activate the appropriate neurons ? The answer for question let us know how neural network is going to work. Let's take one neuron in the second layer which is yet to be activated. What value should it hold ? How to determine that ? Yes! I am talking of the activation number. WELL! For that, just do the following blindly for now! We'll explore them in detail later in this book.

★ Find the WEIGHTED SUM

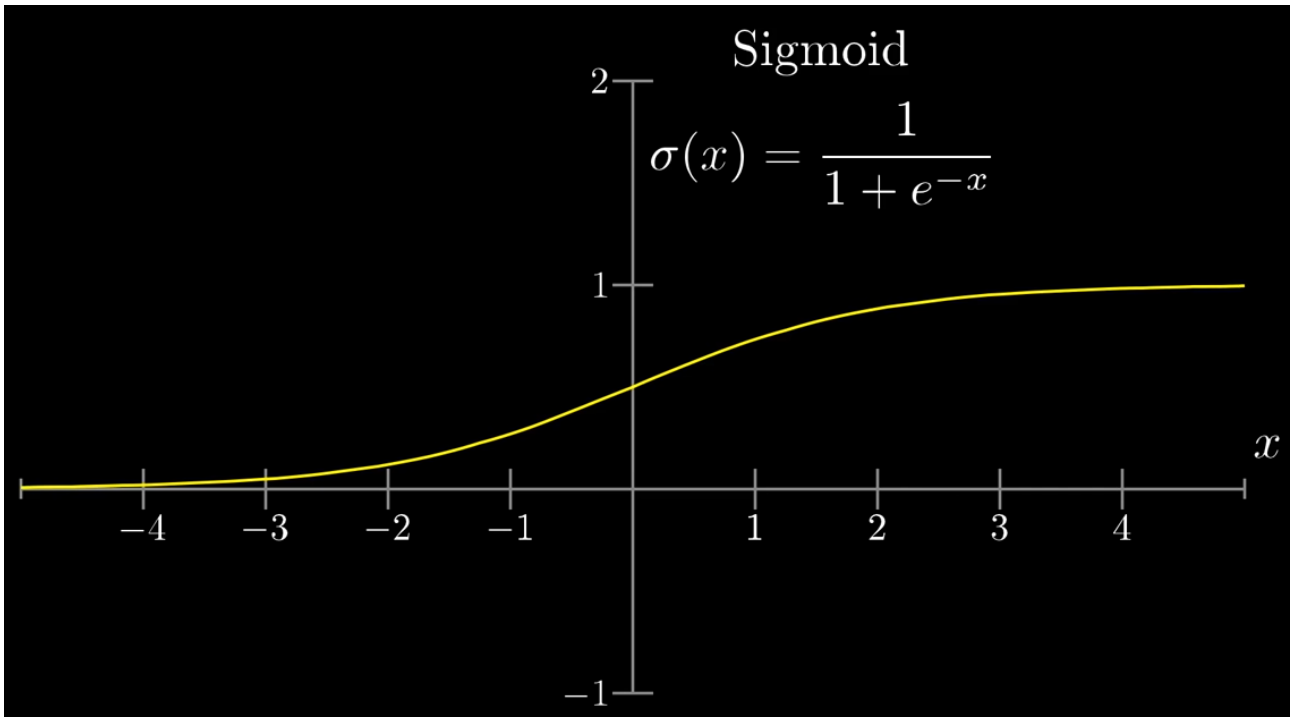


Find the sum of the activation numbers each multiplied by a specific weight (we'll talk of it later).

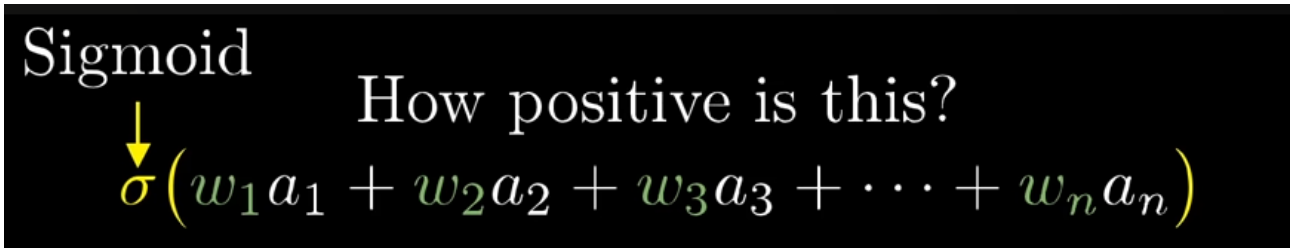
★ Using AN ACTIVATION FUNCTION



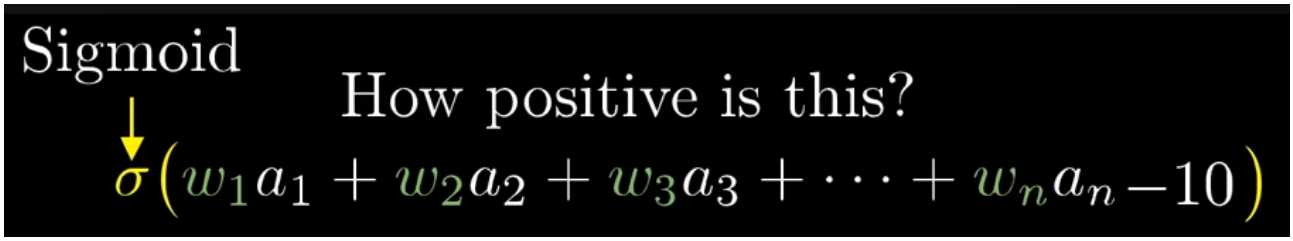
The Activation number should be in a particular range. For gray-scale, the range is 0 to 1. For this, Sigmoid Activation function is quite appropriate. This function takes any real value as input and outputs values in the range of 0 to 1.

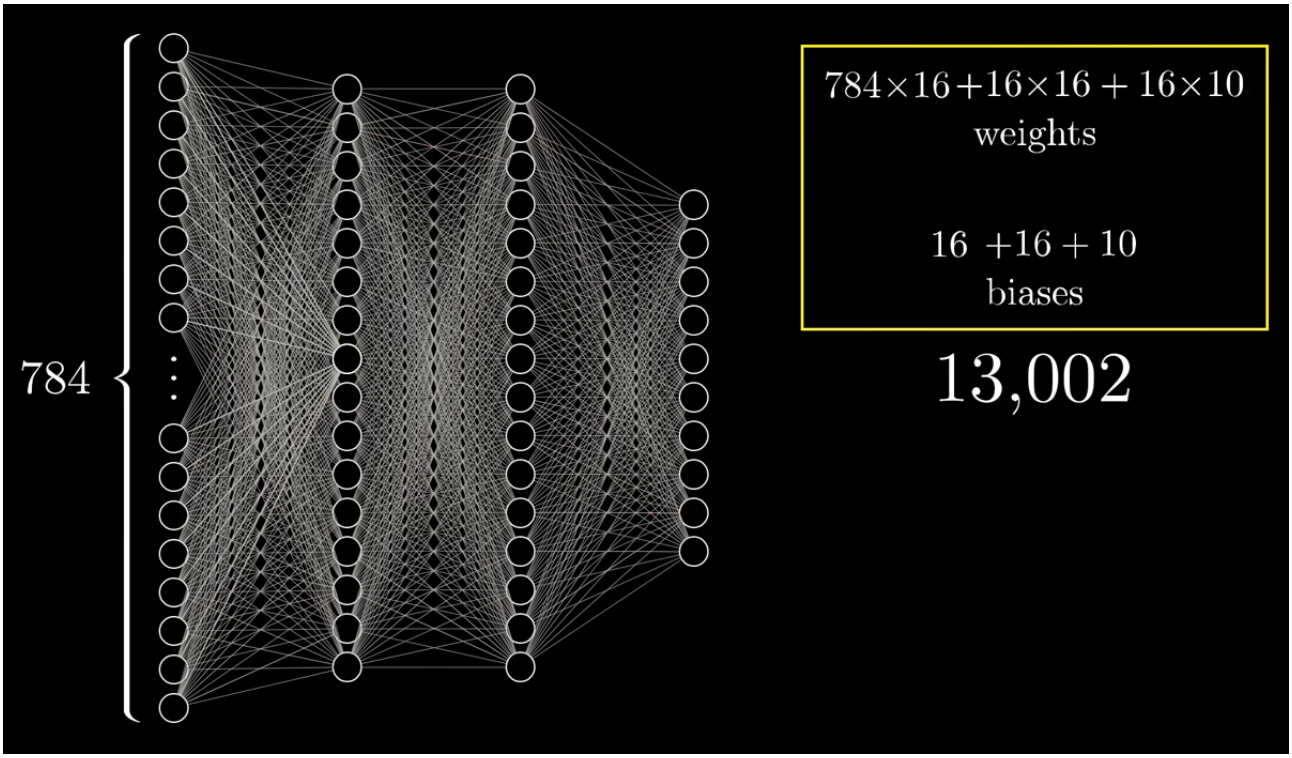


Basically, in sigmoid function, very negative numbers end up with 0 and very positive numbers end up with 1.



In simple words, the sigmoid function tells us how positive the weighted sum is. If we want the activation to happen only when the weighted sum is greater than 10, add a value of -10 to the weighted sum and sigmoid it. This value is called BIAS.

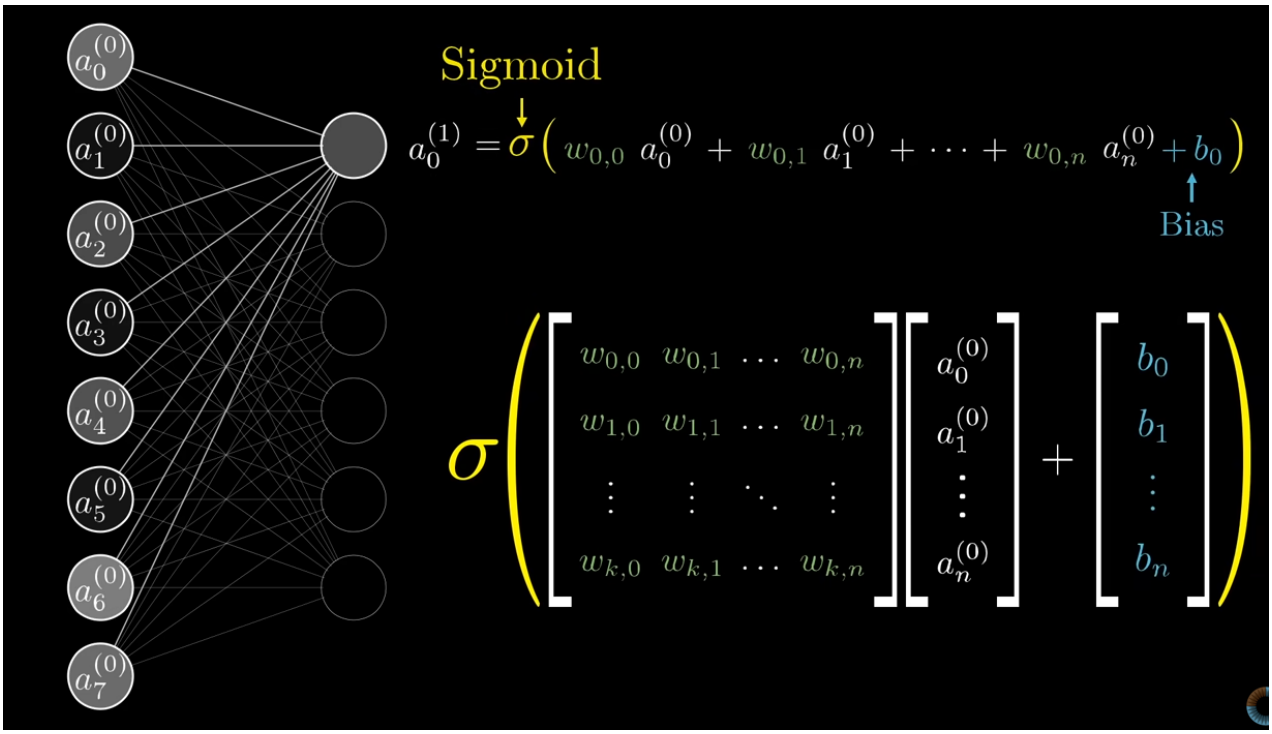




We've been talking of one single neuron. It is the case for almost all the neurons. As we can observe above, we just... just.... just..... need a 13,002 weights and bias to set! Wow! Can you manually go ahead ? Better, eat some fruits and nuts that time! So, how can we ? It's to be learned in a moment.

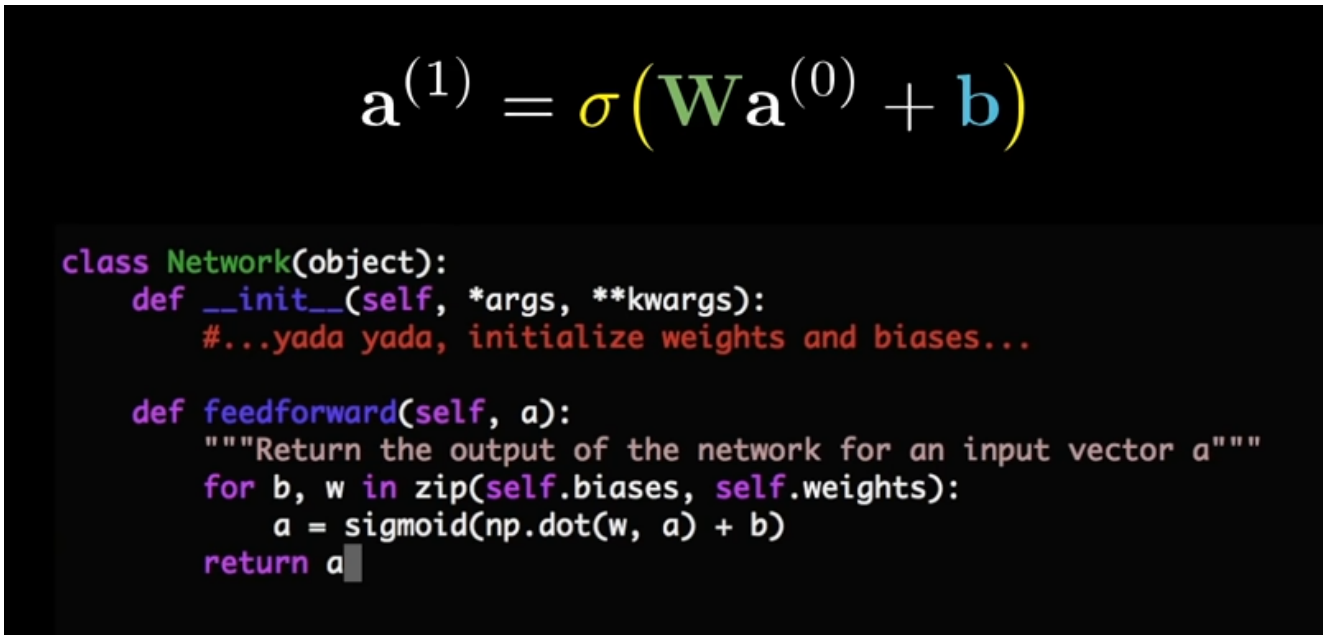
★ ALGEBRAIC REPRESENTATION

It's a bit lengthy to represent the sigmoid expression. We can do it with the linear algebraic representation.



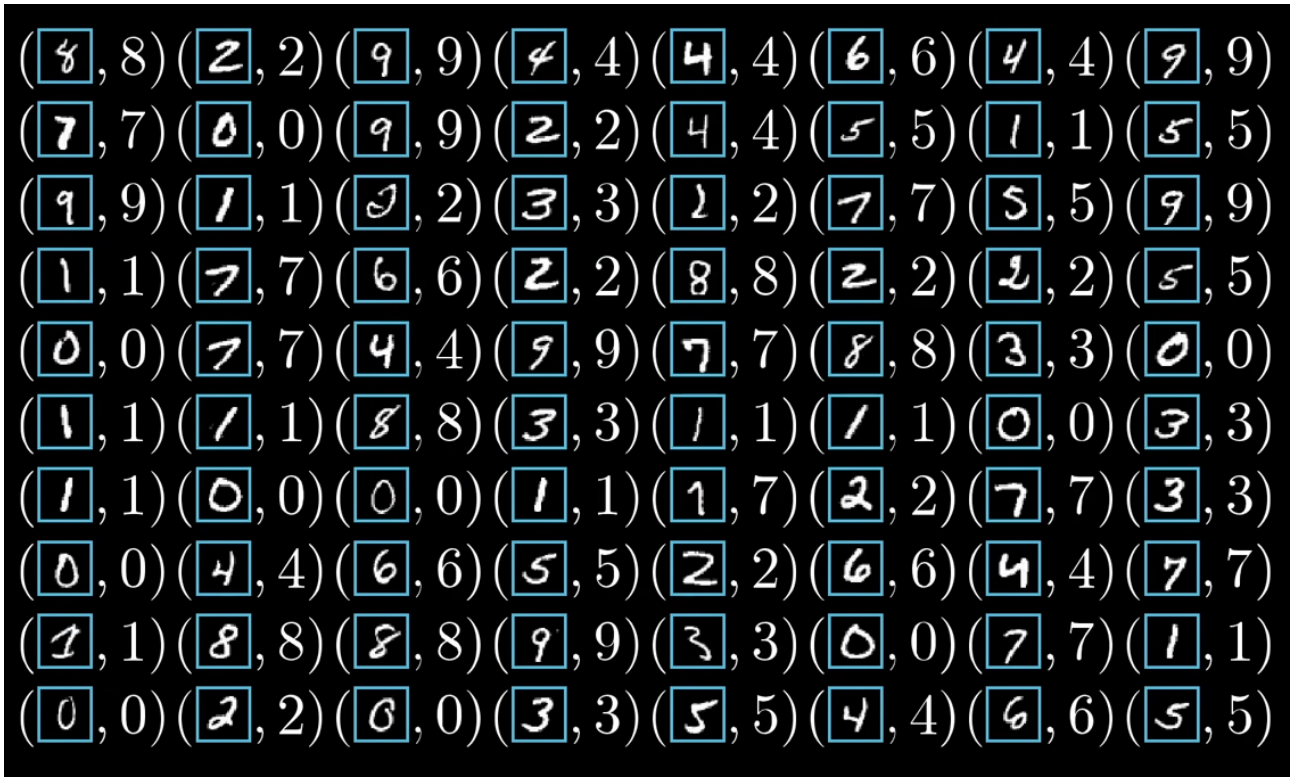
It is a bit essential to use Linear Algebraic notations to represent the expression.

Because, for most of the languages, it is a much-much efficient way.



Because, for most of the languages, it is a much-much efficient way. (WOW) Definitely, the numbers held by each neuron depend on the input image. It is quite meaningful for now to think of the each neuron as a number holder whose number would be 0 or 1 based on the previous layer input which is passed into a activation function like sigmoid. If we think beyond, really the entire network is just A FUNCTION, one that takes 786 numbers as input and gives out 10 numbers as output. But, it's a complicated function. That's it! YET, how does this network learn the appropriate weights and biases ?

We are about to build a model that does hand-written digit recognition. The MNIST dataset is much cool. There are many and many of hand written digit images and the label.

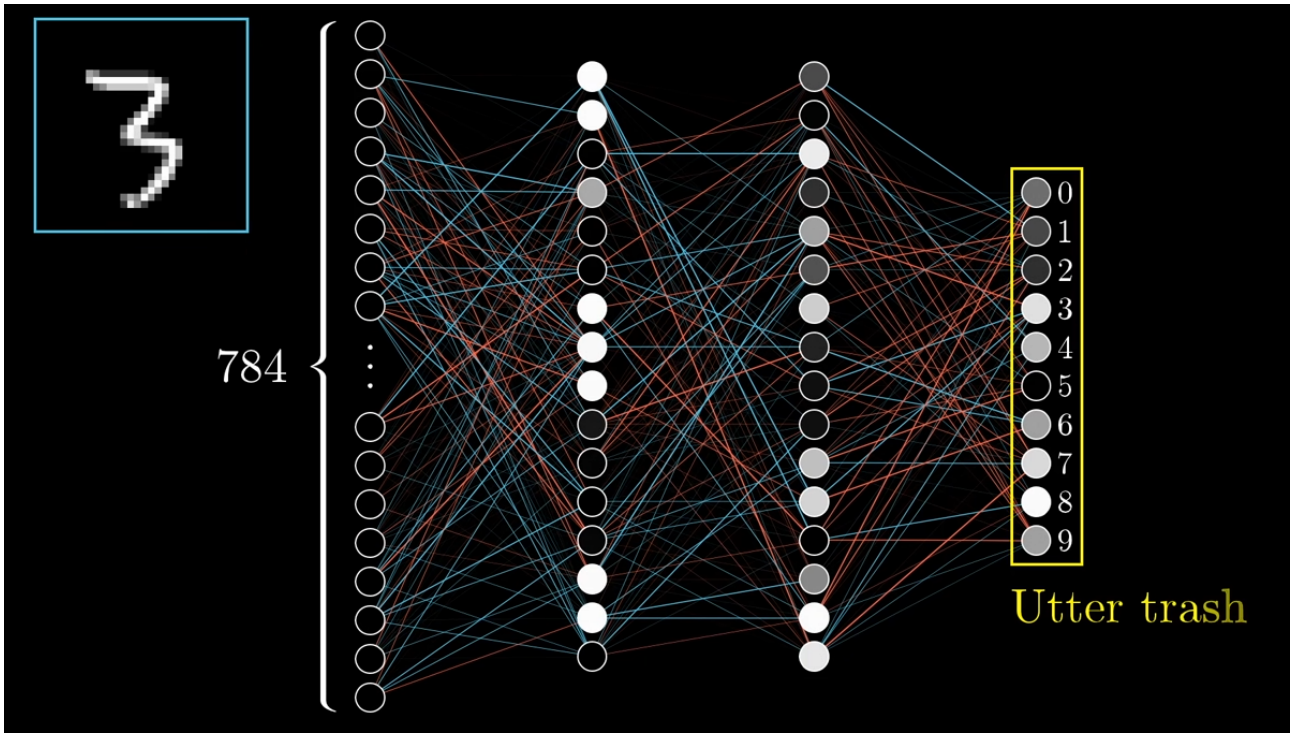


We'll come to it later in a moment. Before that, we have a bit to learn.

★ LEARN FROM MISTAKES

To start things off, we are just going to initialize all of these weights and biases totally randomly! (Needless to say this network is going to perform pretty much horribly on a given training) If we give 3, the output will be a MESS!

As we see below, what we get is a TRASH indeed. But, why should we do this ? That's TRAINING.



★ THE COST FUNCTION

"Hello Computer! You have provided me a very wrong output! It's a utter trash! I need THIS! Not that!" - **We have to tell the computer that this is not what was expected when we get the wrong output with random weights and bias values. Then, the computer will take note of it and try to improve the output next time. It is similar to teaching something to a baby and even orangutan monkeys learning to fear and beware of snakes.**

They find a snake and fear.

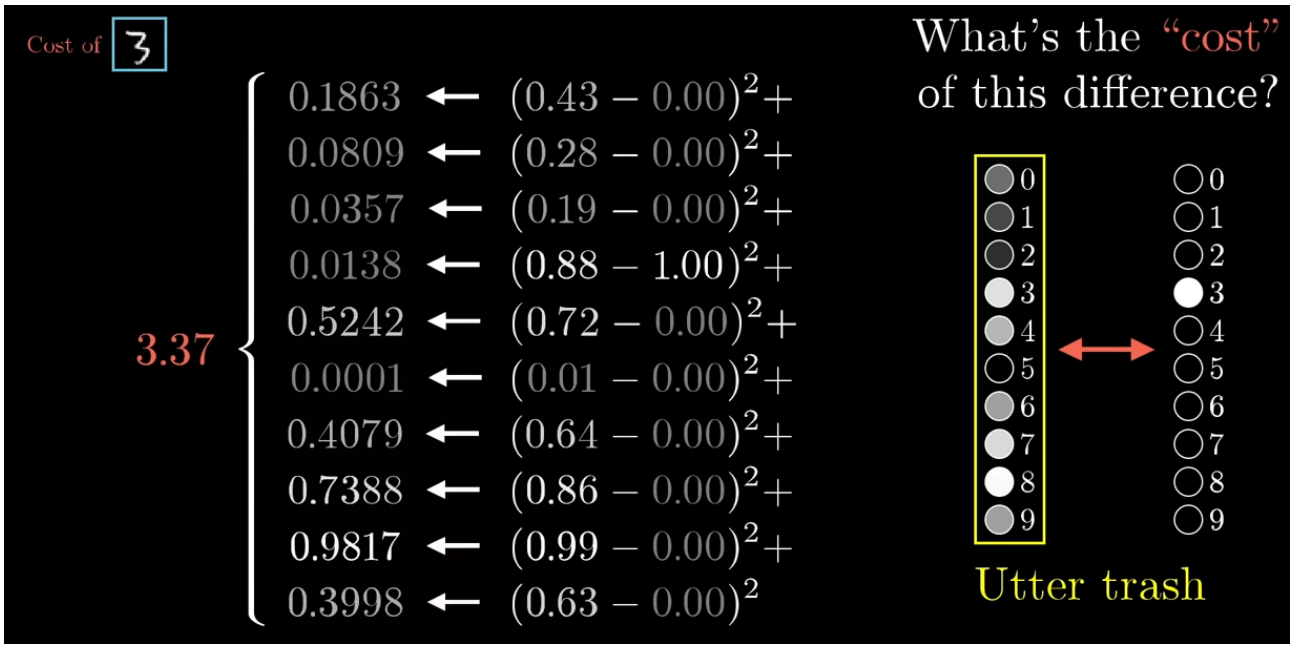
They expose their fear by hugging.

Lesson：To fear; Beaten in front of them

But, how to say these all to the computer ? Here, we need the so-called COST Function. For a definition sake, we can read something like,

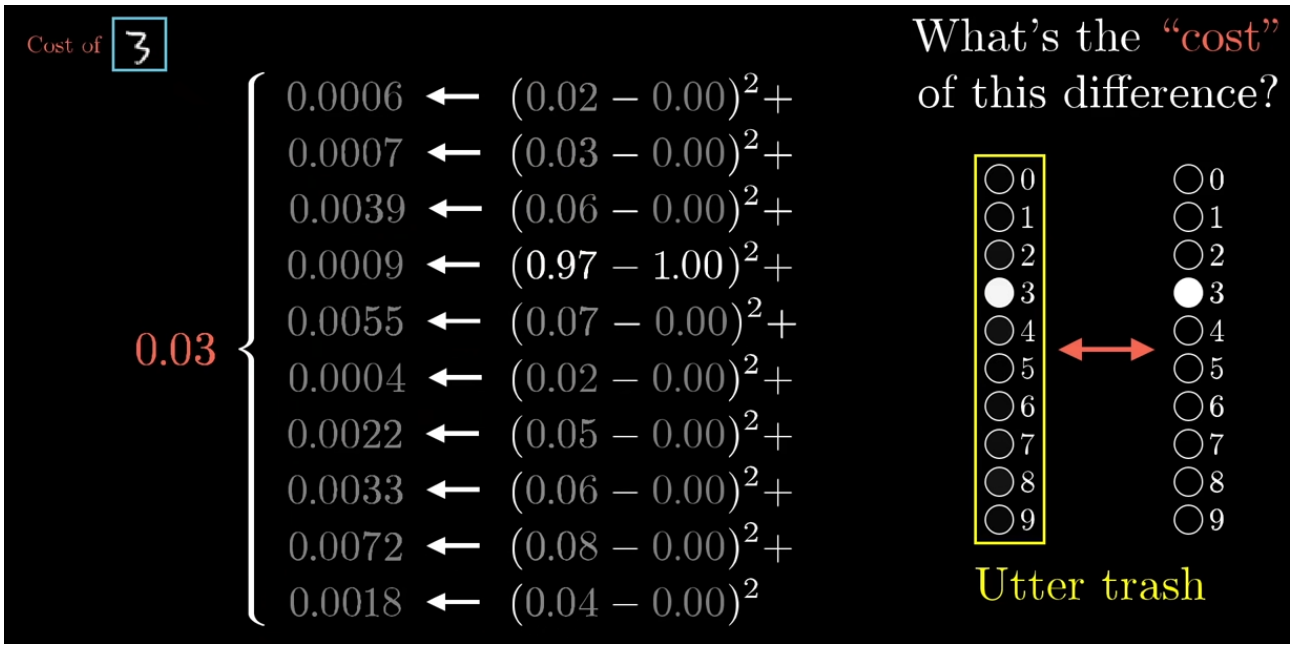
"A cost function is an important parameter that determines how well a machine learning model performs for a given dataset."

Mathematically speaking, we have to add up the squares of the differences of the trash value activations and the value we want them to have. This is what exactly the cost function does. This is the cost value for the single training example.

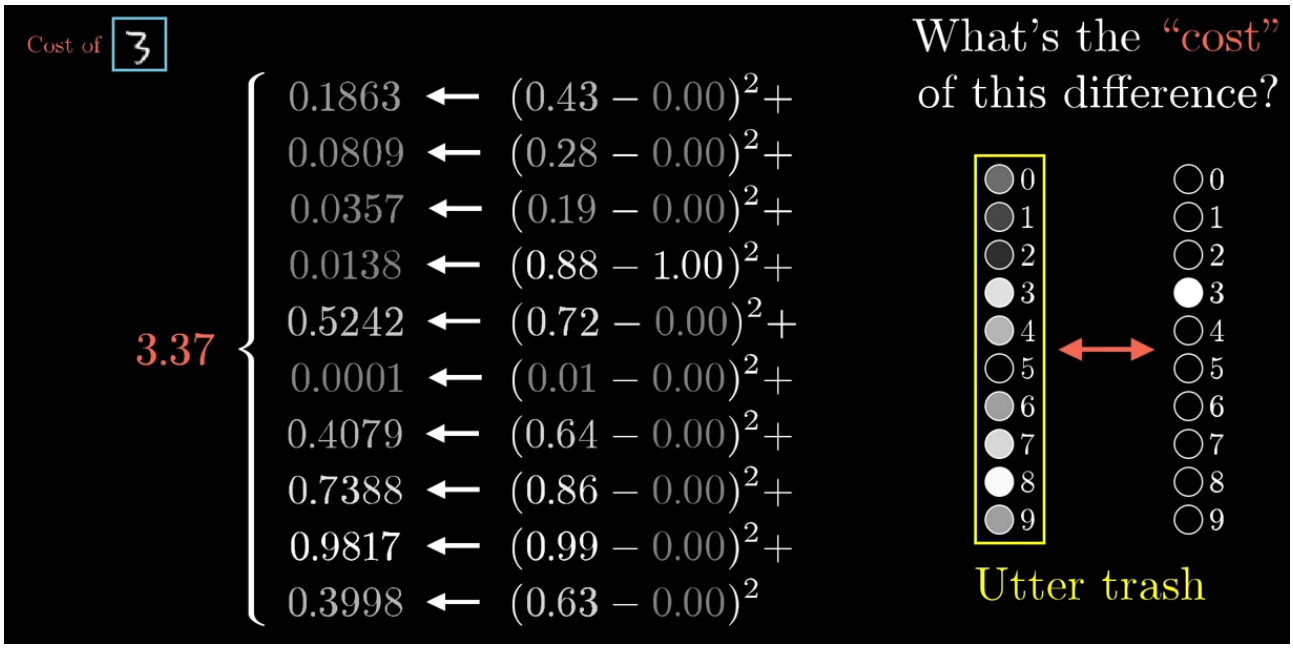


If the network classifies the image correctly, this cost value will be less. If it is not, then the cost value is increased.

When the network does correctly,

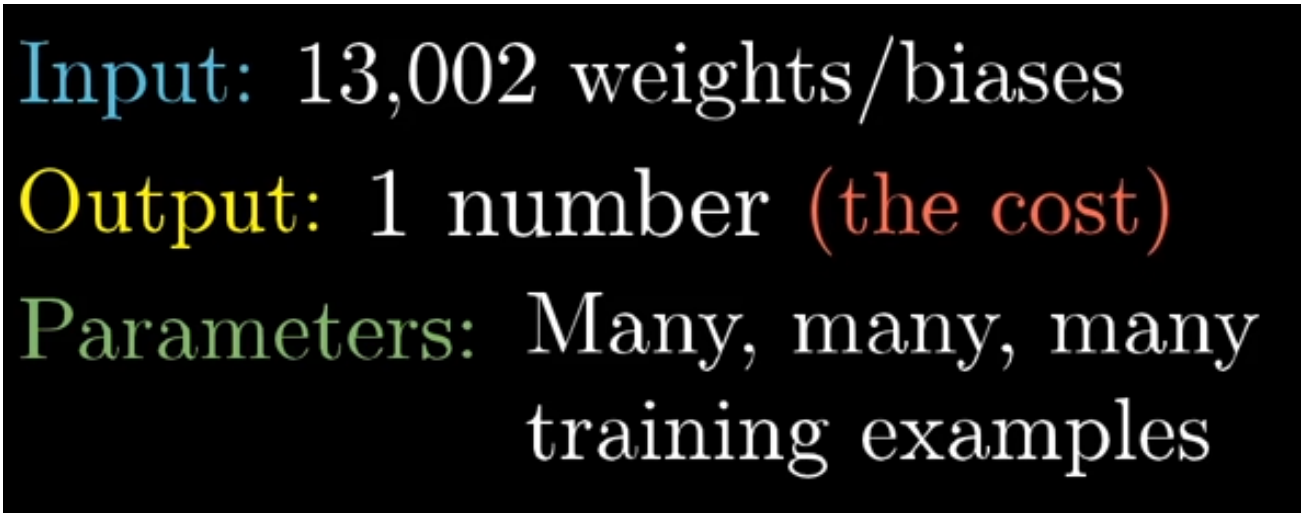


When the network does wrongly,



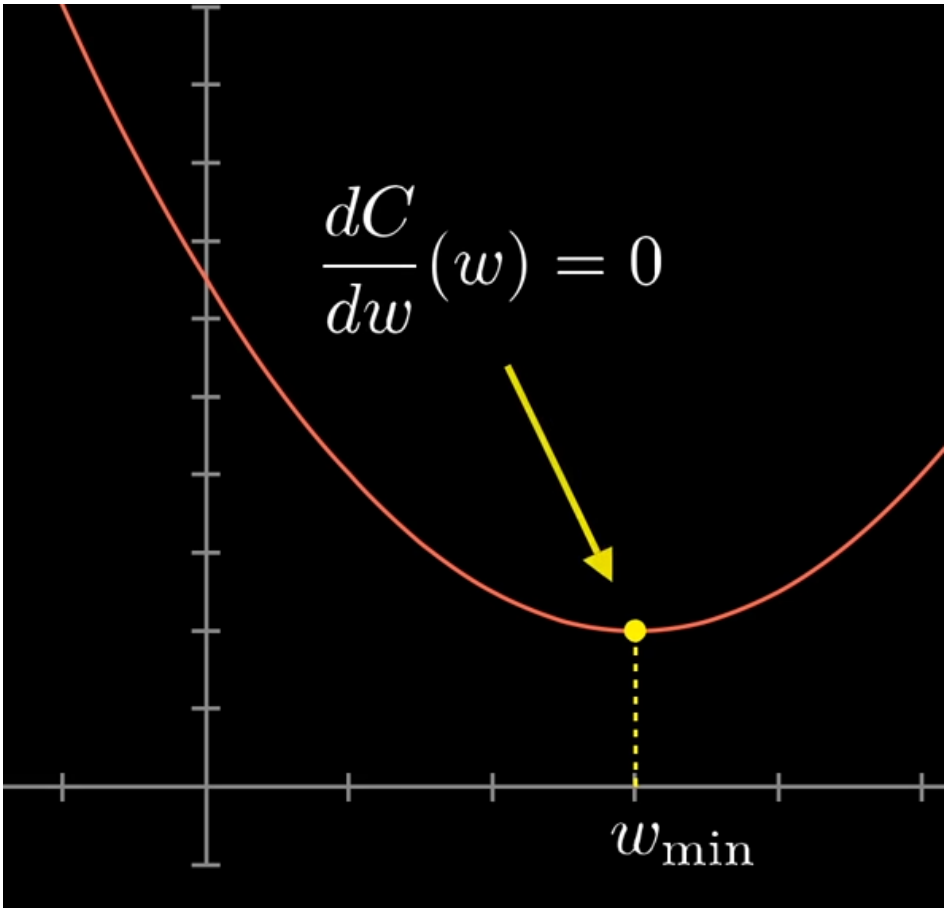
Ah Well! We need to perform many trainings and tell the computer the cost value. The average of the cost states how lousy the network is. Because, A Cost function returns the error between predicted and the actual.

The Cost function can be viewed as the following. A Function that takes many input weights and biases and gives out one value the cost. Many training examples must be given as parameters for improving the future output.



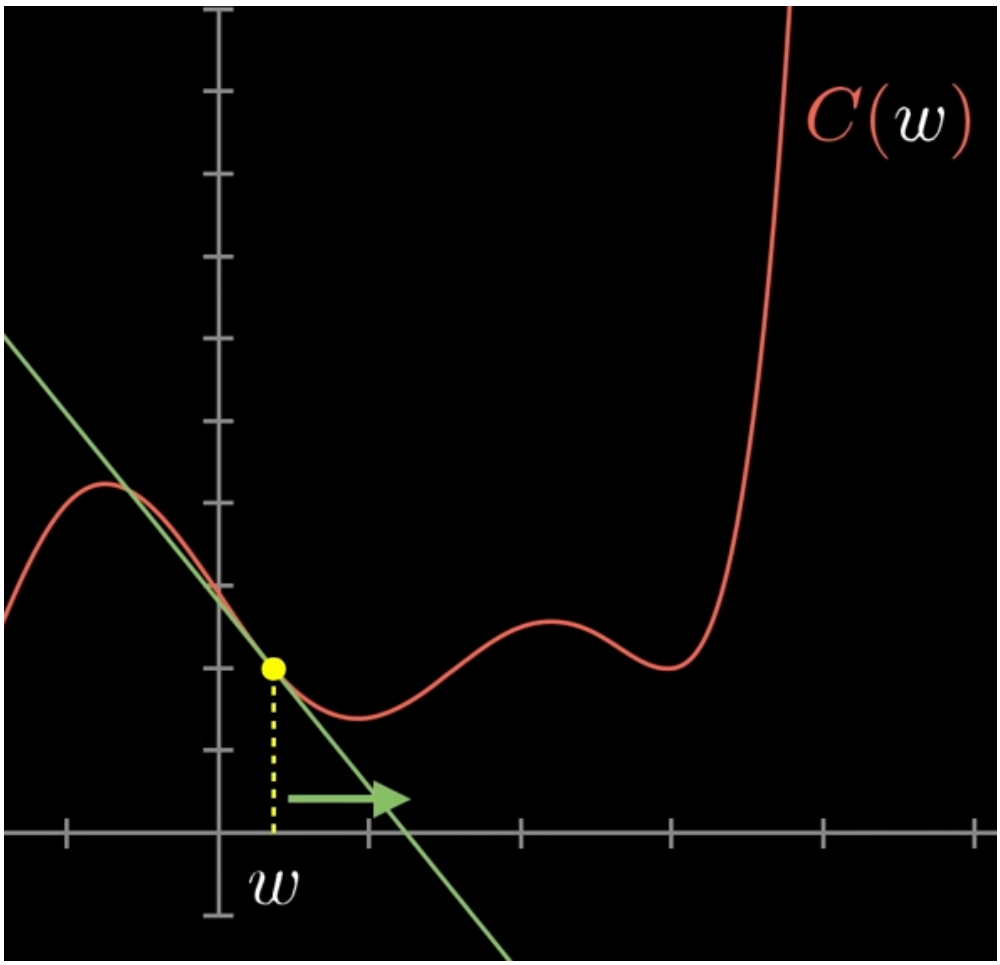
It is quite needed to say how to change the output to the anticipated one is much needed rather than just scolding the computer on observing which the future outputs will be so provocative. So, how can we tell the network to CHANGE to yield the appropriate output ? What we have to do here is, "FINDING THE INPUT THAT MINIMIZES THE COST". Rather than considering the cost function with lot and lot of inputs , just consider a cost function with only one input say w.

How to find for which value of w, the cost function will yield a minimum value ? If we are familiar with calculus, we can use the derivative. But, that can’t be used for complex functions.

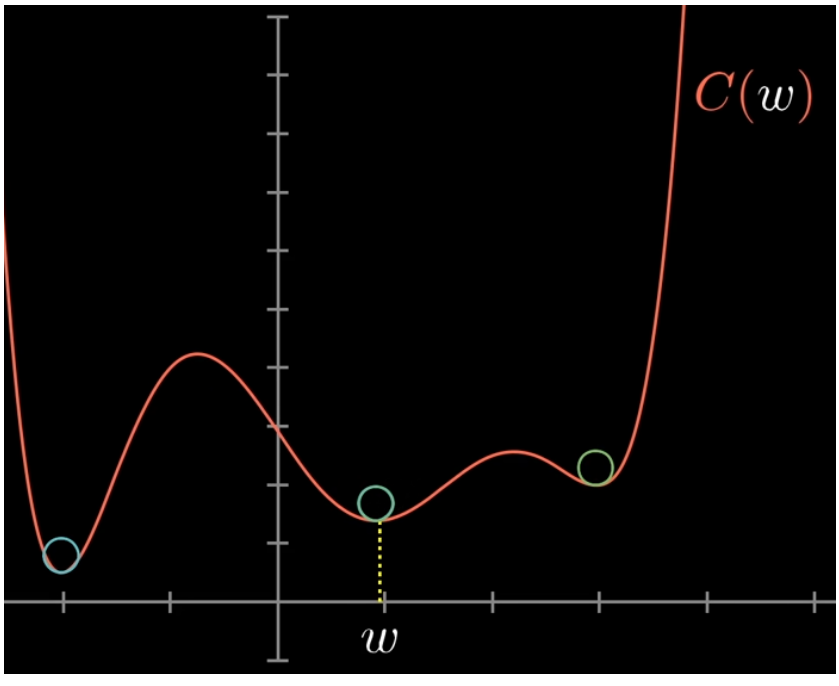


So, what can be a perfect solution to achieve this ? What we have to do is, find the slope of the function where you are. If the slope is positive, shift to the LEFT and if it is negative, just shift to the RIGHT.

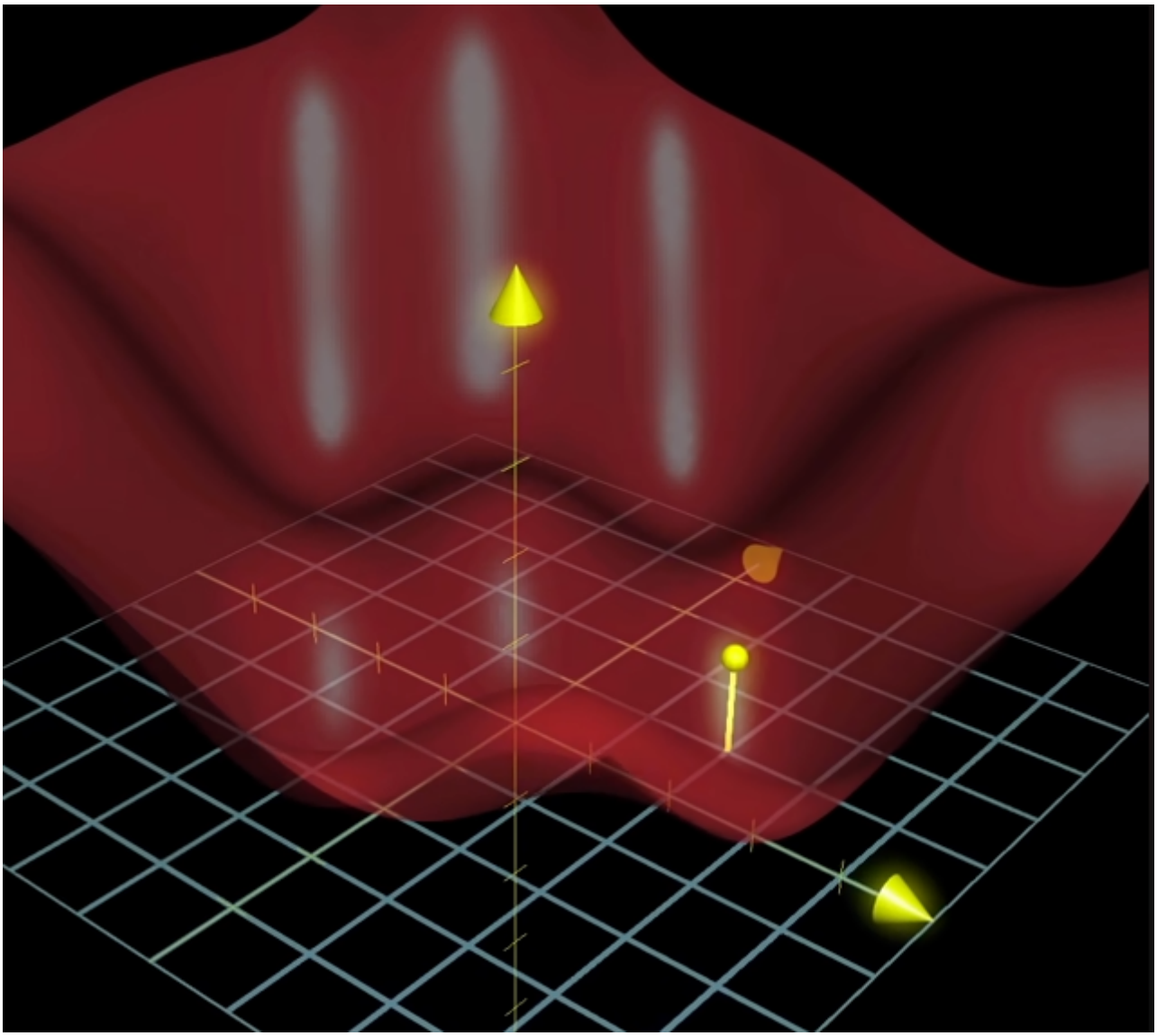
Finding the minimum value can be done like the following.



Surely, we can find a local minimum here. We can imagine a ball rolling from a mountain. Possibly, we can have many minimum values.

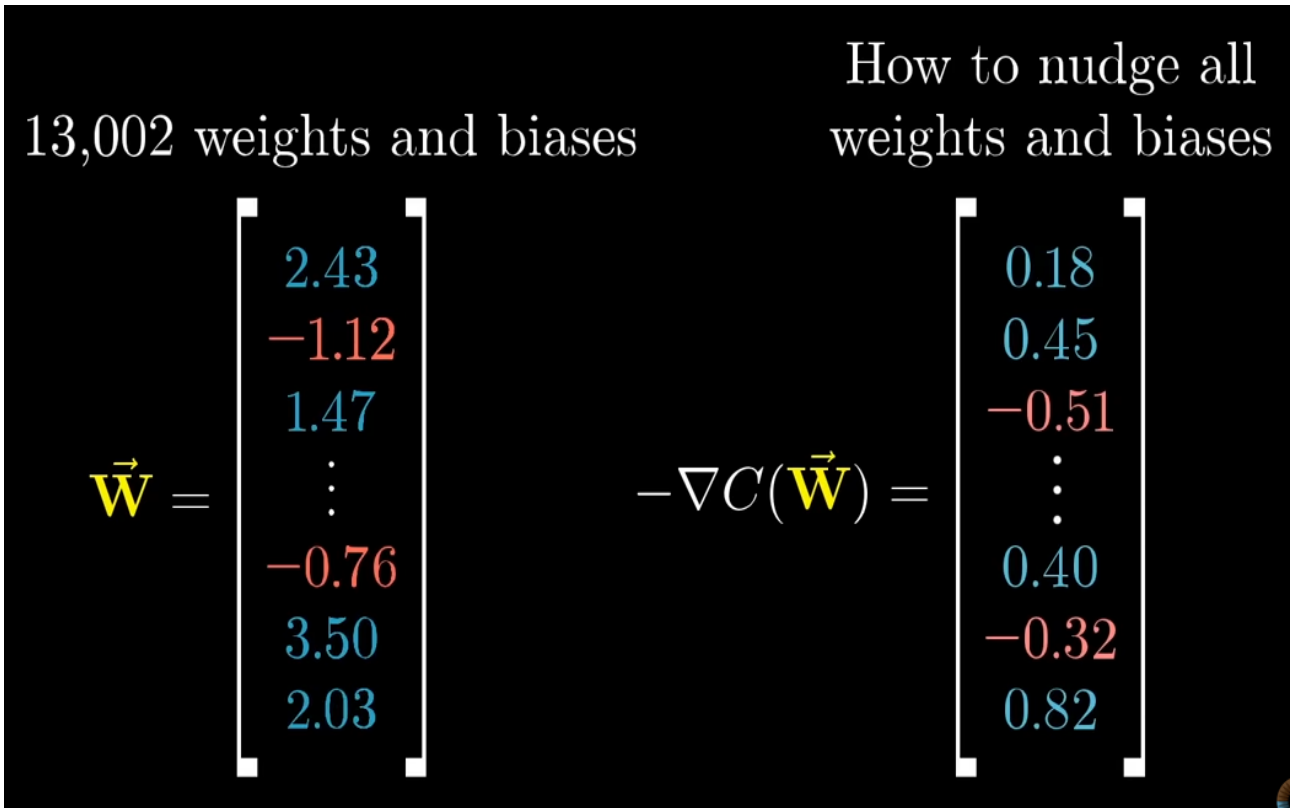


And there is no guarantee that the local minima we reach may/may not be the minimum of all the global minimum. Now, let's imagine a function with two inputs and an output. Let the input space be the XY plane and the cost function , C(x,y) be a surface over the plane.



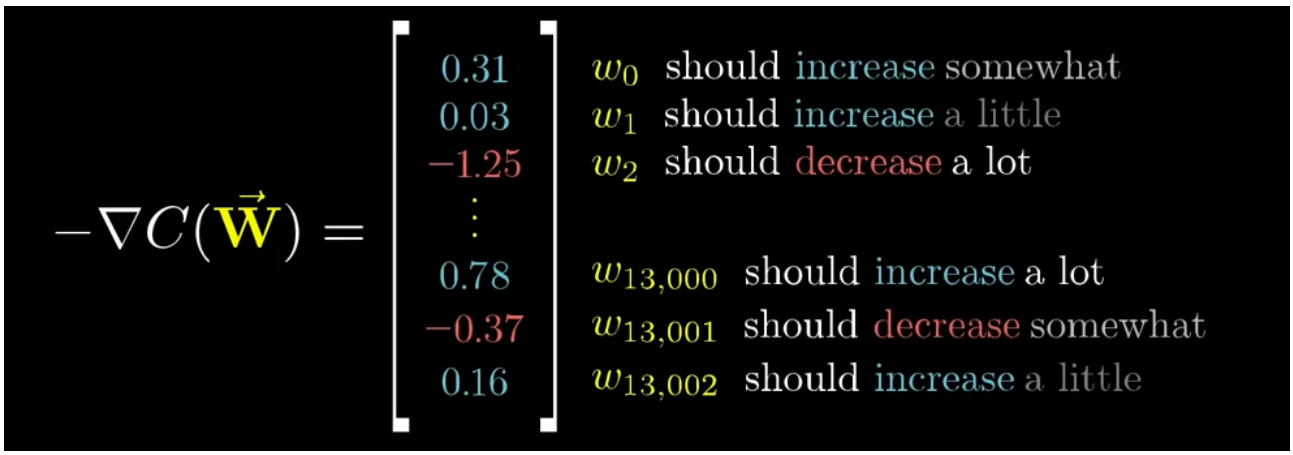
We have to find out which direction decreases the cost value most quickly.

Gradient of a function gives us the direction of the steepest ascent (The direction which increases the value of the function most quickly). It is enough to just take the negative of that gradient value which decreases the value of that function most quickly. We have to calculate it for every randomly selected weights and biases.

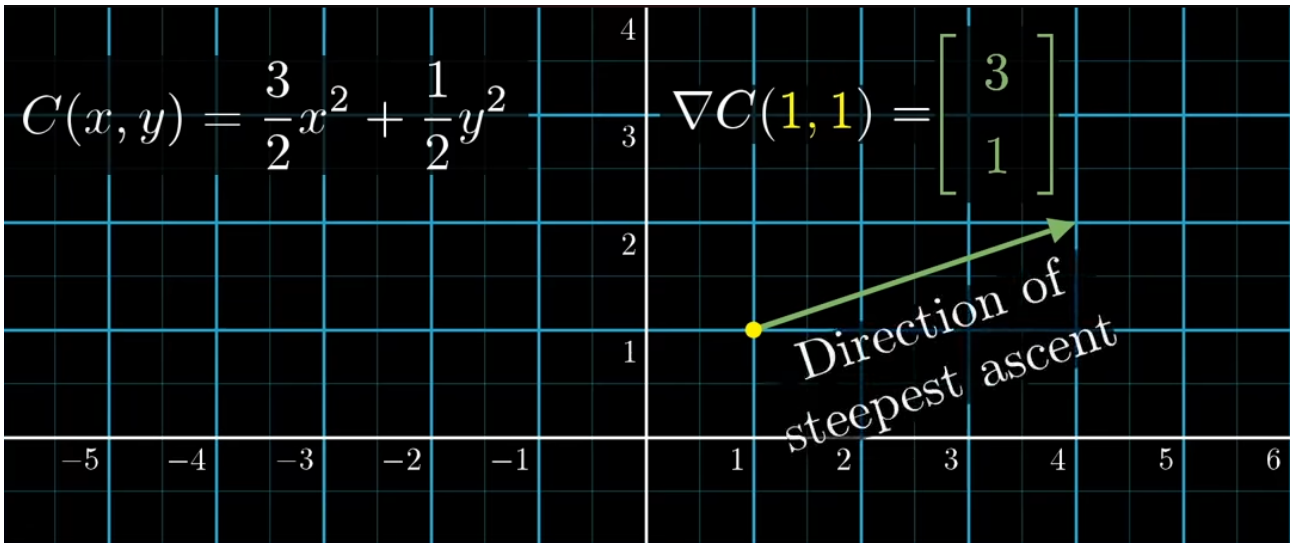


The algorithm that effectively calculates this gradient value is termed as THE BACK-PROPAGATION.
So, here in Neural Networks, the term learning is simply REDUCING THE COST FUNCTION.

These nudge values really matter. And certainly, their sign and the extent are also so essential for what to do either increase or decrease.



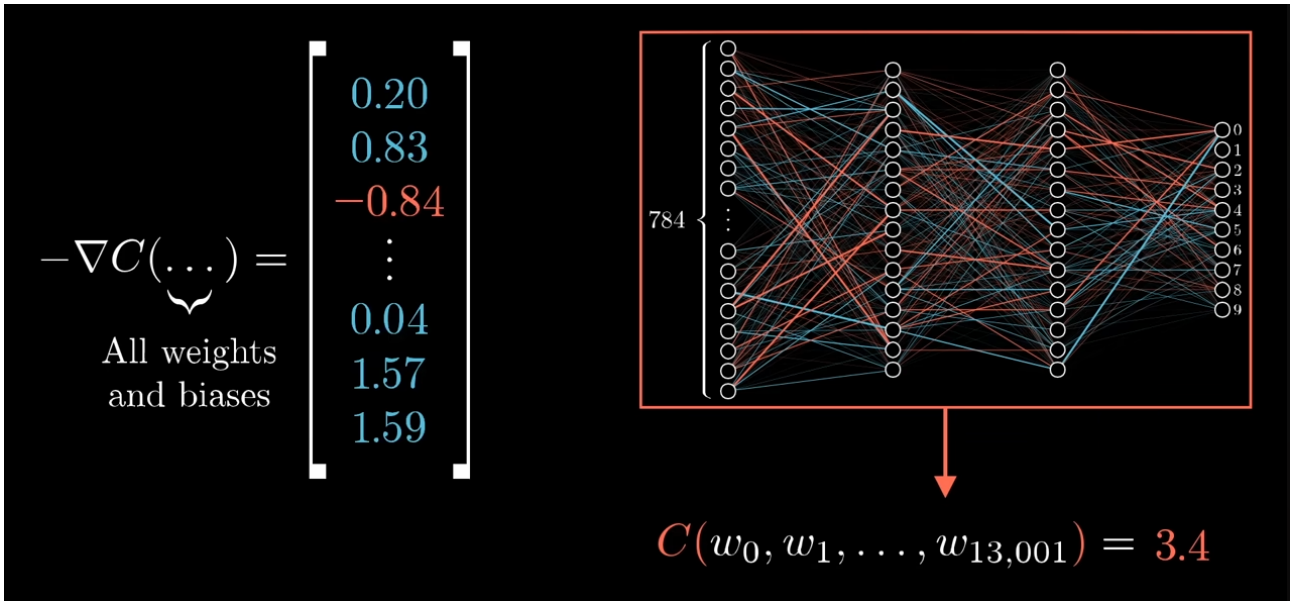
Let's talk of the steepest ascent that we have learned earlier.

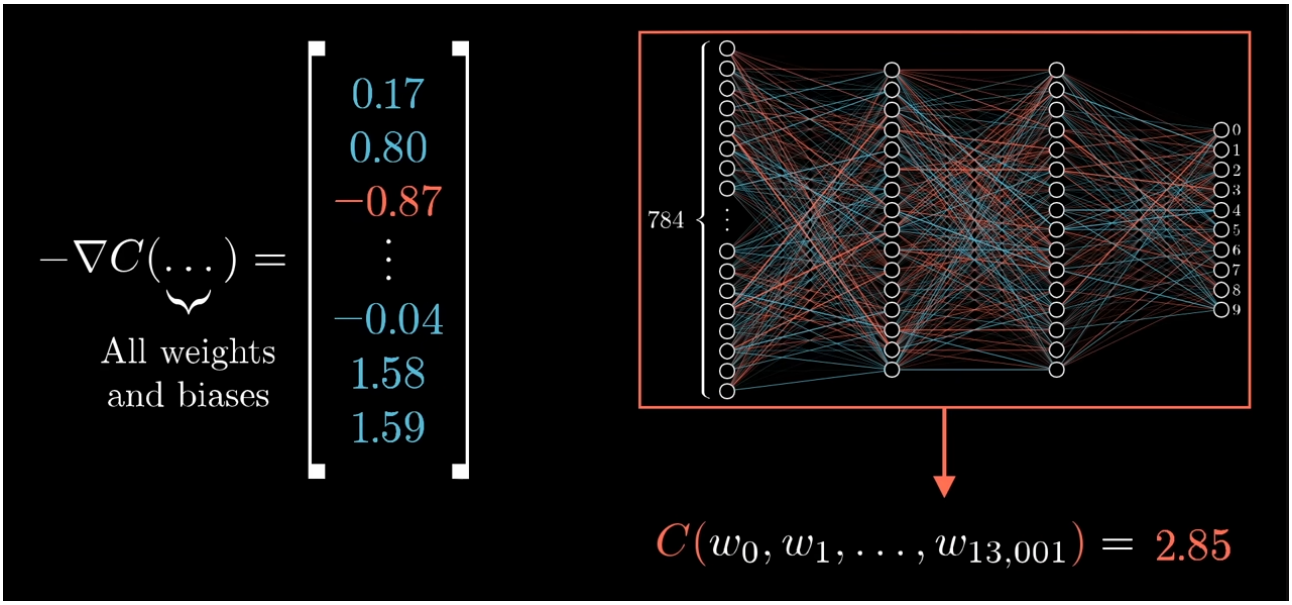


Finding the Multi-Variable Calculus of the cost function for a particular point, it can determine the direction of the steepest ascent. We have to take the negative of this for the direction of the steepest descent.

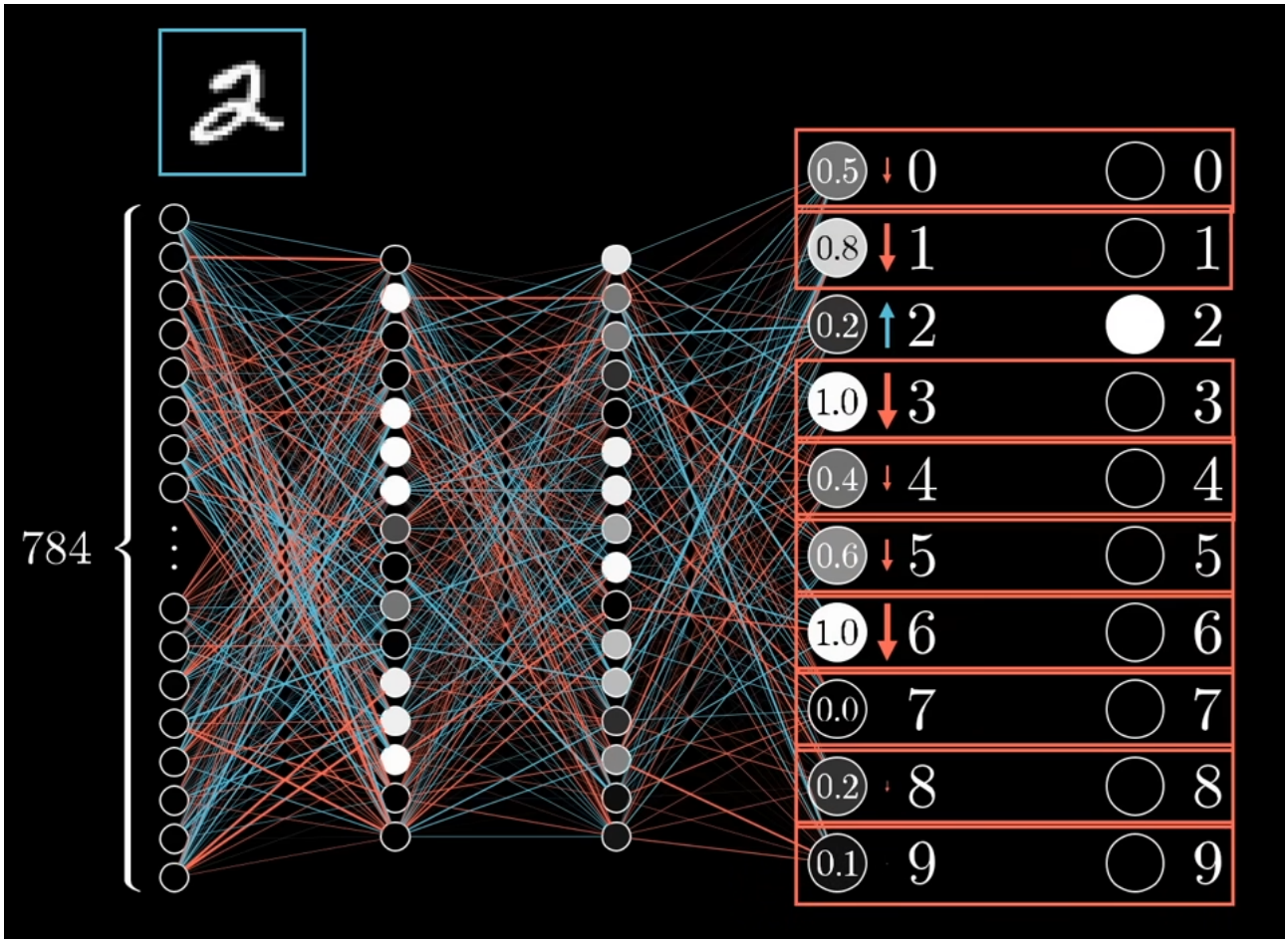
★ INTRODUCING BACK-PROPAGATION

We have been using the term, “Learning” for a long while. But, There is no idea of how that is achieved. Here we go. It’s simply because of the step called Back-Propagation. We wonder of the Gradient Descent values on applying which to our weights can retard the cost the most quickly.

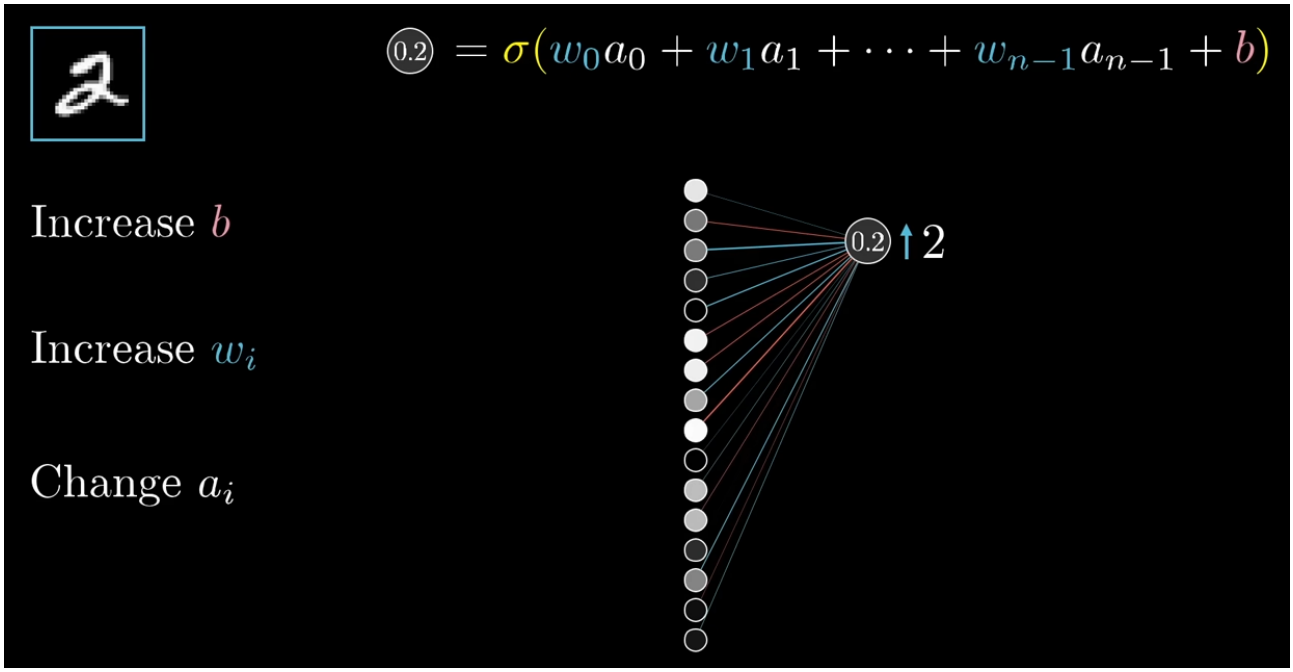




Okay.. Here , we can observe that the gradient values are the most needed. Finding them is a challenging task. And indeed, Back-Propagation is an algorithm for finding those gradient values. Now, let’s talk a bit of the following scenario. If we want to recognize 2, what we have to do? We need the output activation for 2 to be nudged up (LIT UP) meanwhile that of others should be nudged down.



We know that the output activation depends on the previous activations, weights and the bias.



So, there are three different avenues that can team up to increase the activation. We can increase the bias (b). We can increase the weight (w_i). We can change the activation value (a_i).

HEBBIAN THEORY IN BIOLOGY

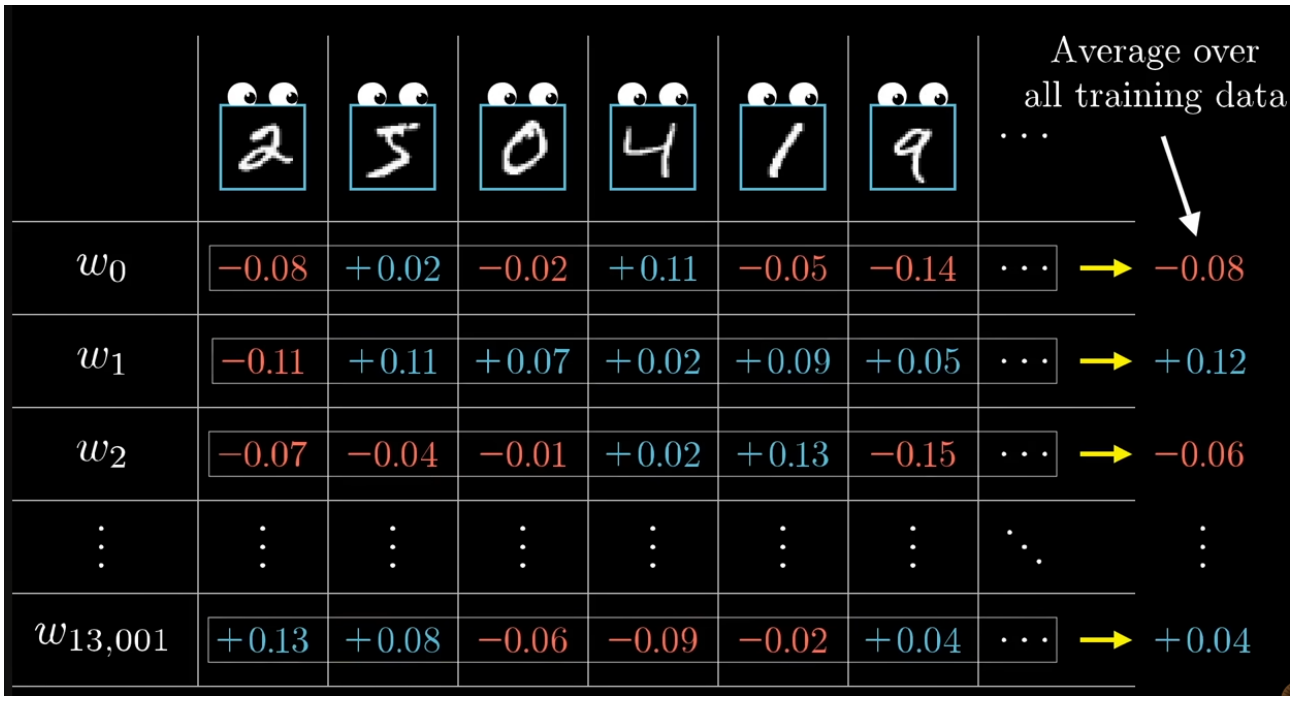
“Neurons that fire together wire together”

According to The Hebbian’s Theory, “The biggest increase to weights, the biggest strengthening of connections happens between the neurons which are the most active and the one which we wish to become more active”. But, we can’t assure that the Artificial Neural Network exhibits all the properties of the Natural Neural Network.

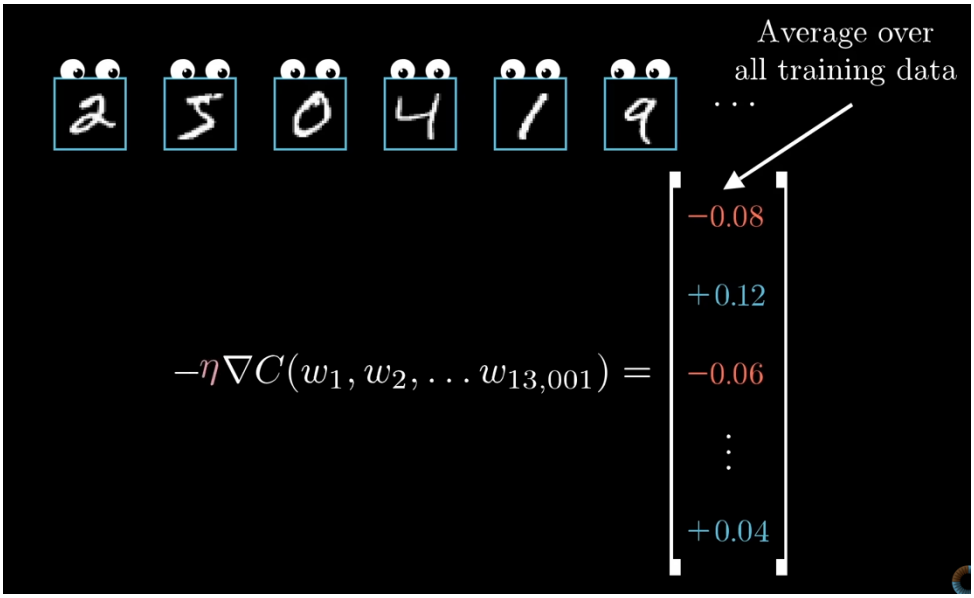
SO WHAT TO DO ?

Back-Propagation indeed! We know that there requires a tuning in the activation values of the neurons in the output layer.

To achieve such changes, what we have to give to the previous layer ? Adding together all these desired effects, we will get the list of nudges to be applied to the second last layer. Finding out the average nudges for all the input data collectively gives us the Gradient Descent approximately.



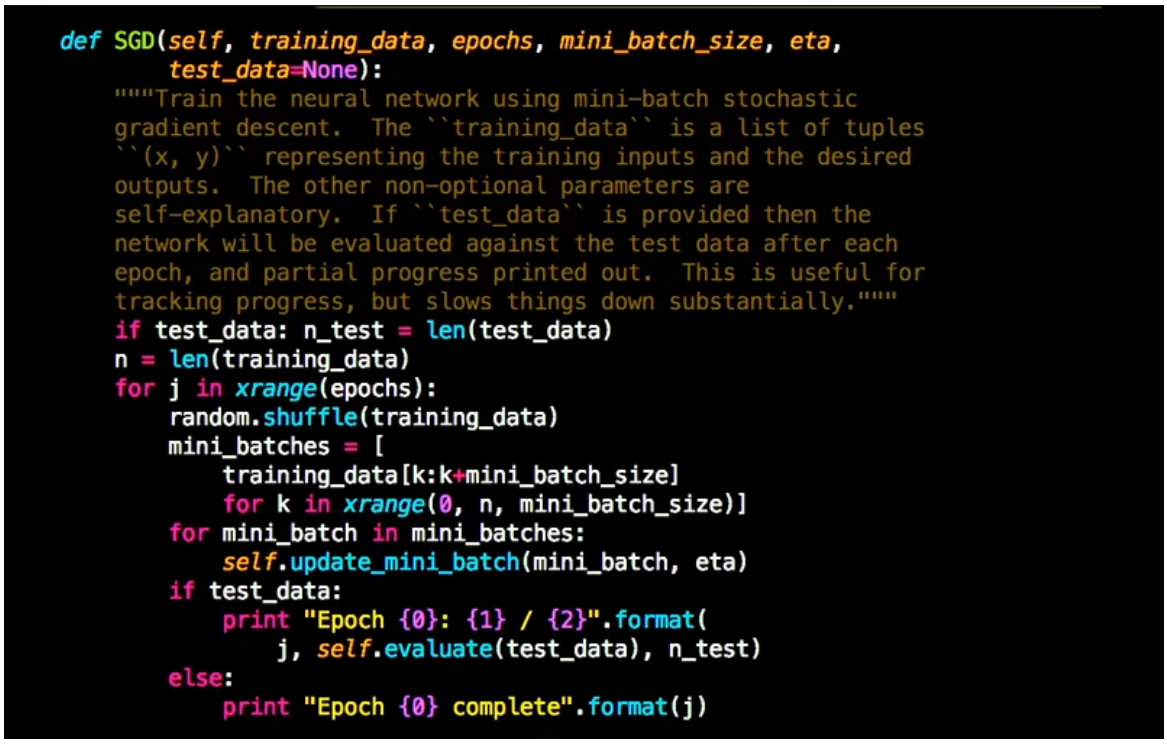
The gradient descent values aren't so accurate. This is DEAD-SLOW.



★ STOCHASTIC GRADIENT DESCENT

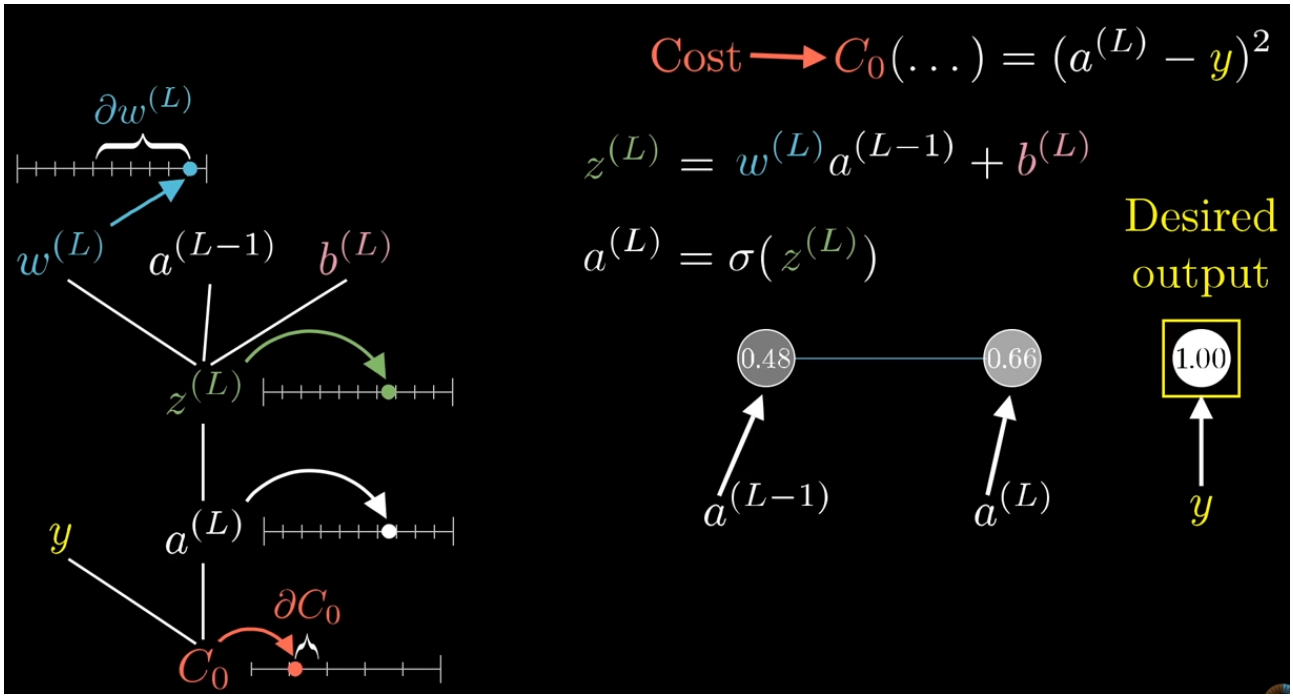
The word '*stochastic*' means a system or process linked with a random probability. Hence, in Stochastic Gradient Descent, a few samples are selected randomly instead of the whole data set for each iteration. In Gradient Descent, there is a term called "batch" which denotes the total number of samples from a dataset that is used for calculating the gradient for each iteration. In typical Gradient Descent optimization, like Batch Gradient Descent, the batch is taken to be the whole dataset. Although using the whole dataset is really useful for getting to the minima in a less noisy and less random manner, the problem arises when our dataset gets big.

Suppose, you have a million samples in your dataset, so if you use a typical Gradient Descent optimization technique, you will have to use all of the one million samples for completing one iteration while performing the Gradient Descent, and it has to be done for every iteration until the minima are reached. Hence, it becomes computationally very expensive to perform. This problem is solved by Stochastic Gradient Descent. In SGD, it uses only a single sample, i.e., a batch size of one, to perform each iteration. The sample is randomly shuffled and selected for performing the iteration.



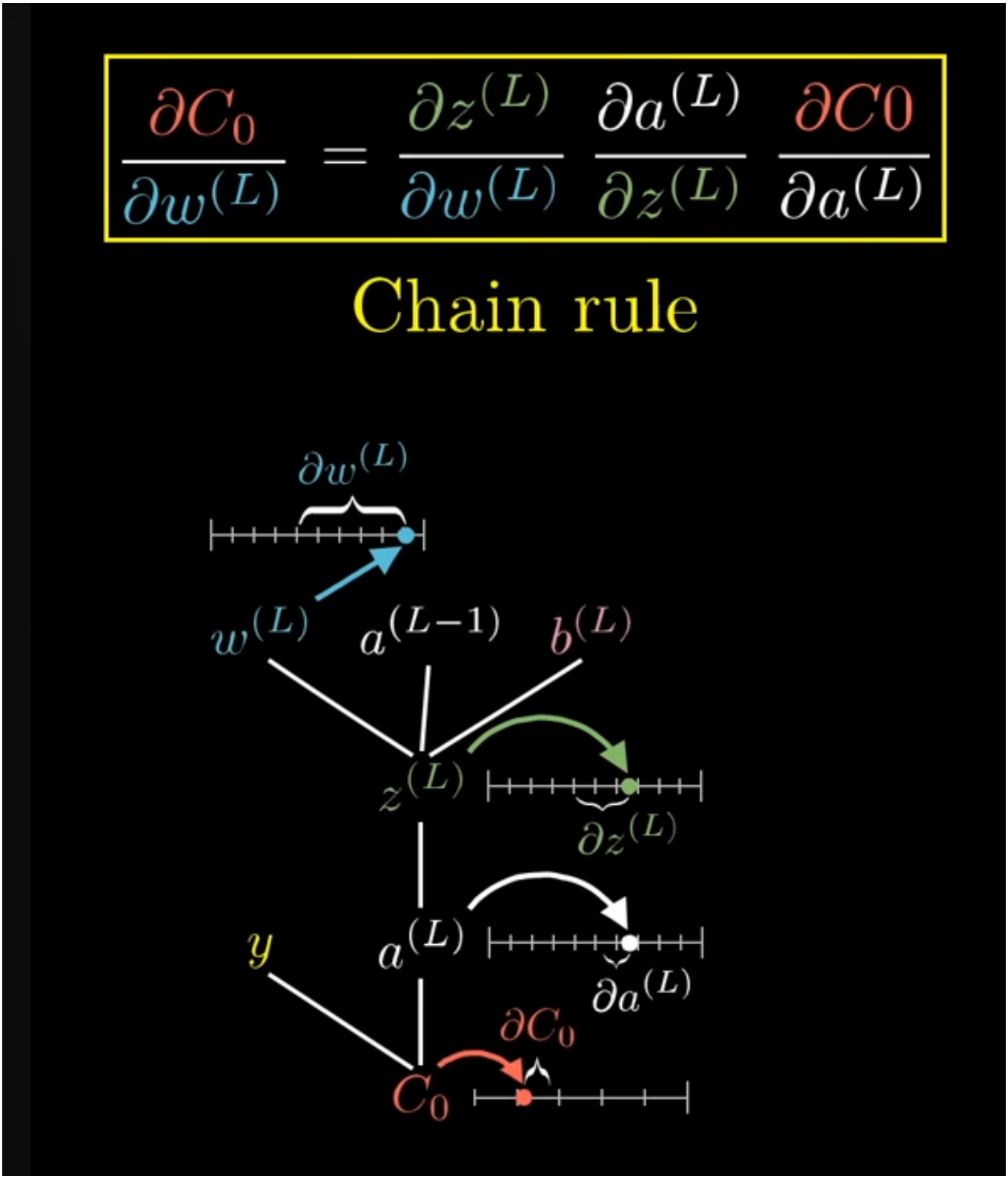
```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):
    """Train the neural network using mini-batch stochastic
    gradient descent.  The ``training_data`` is a list of tuples
    ``(x, y)`` representing the training inputs and the desired
    outputs.  The other non-optional parameters are
    self-explanatory.  If ``test_data`` is provided then the
    network will be evaluated against the test data after each
    epoch, and partial progress printed out.  This is useful for
    tracking progress, but slows things down substantially."""
    if test_data: n_test = len(test_data)
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test)
        else:
            print "Epoch {0} complete".format(j)
```

★ BACK-PROPAGATION CALCULUS

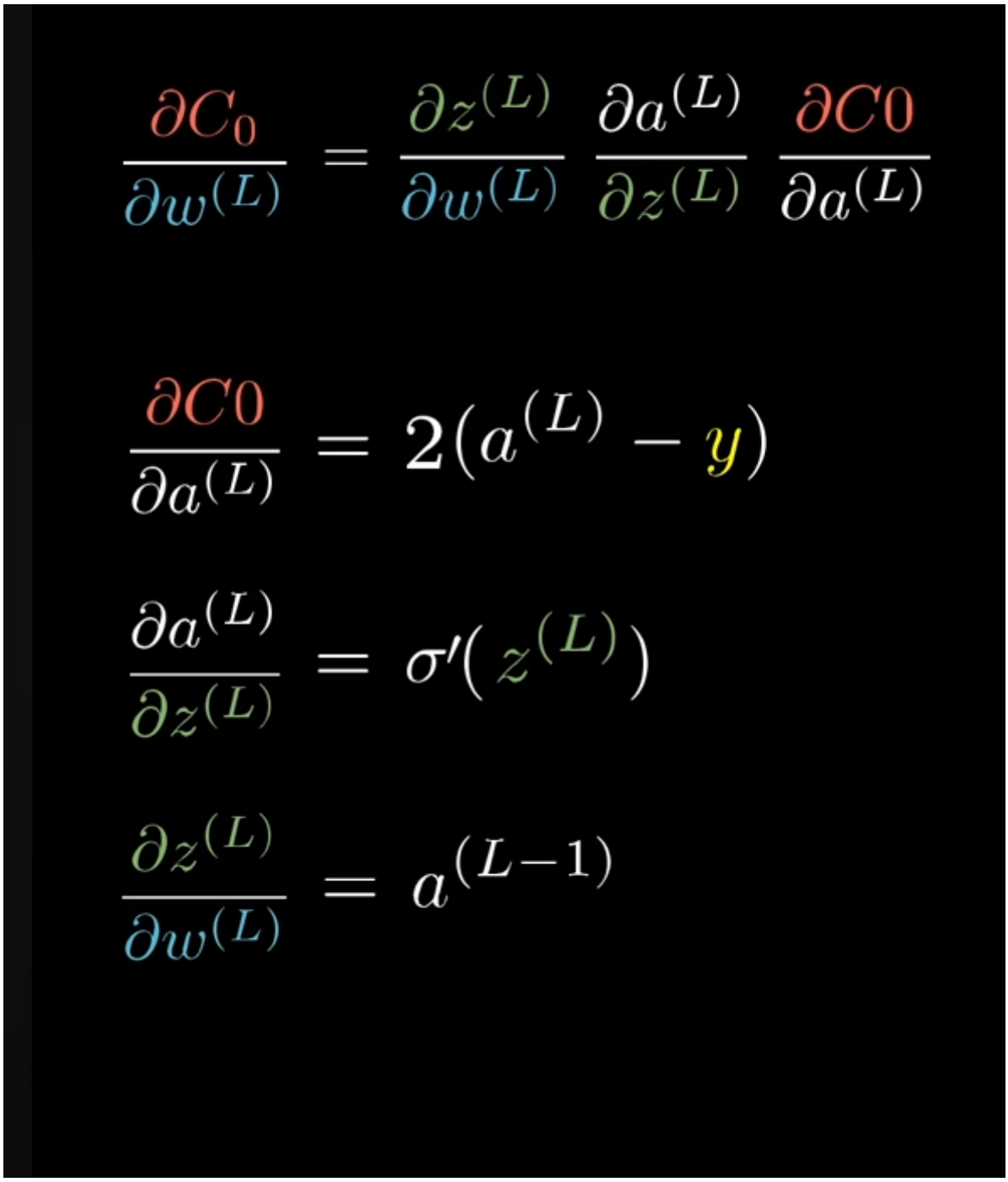


Considering only the two perceptrons of which one is in the last layer and the another is in the second-last layer. The output (The Activation) is found as the above as we have described very well in the past sections of this book. What we have to compute here is how sensible the Cost function is with respect to a minor change in the weight. $\partial C_0/\partial W^{(L)}$ is what we want to find out.

It means "A LITTLE LOT". Changing $W^{(L)}$ impacts $Z^{(L)}$ which inturn impacts the activation $a^{(L)}$ and which inturn impacts the the cost function C_0.

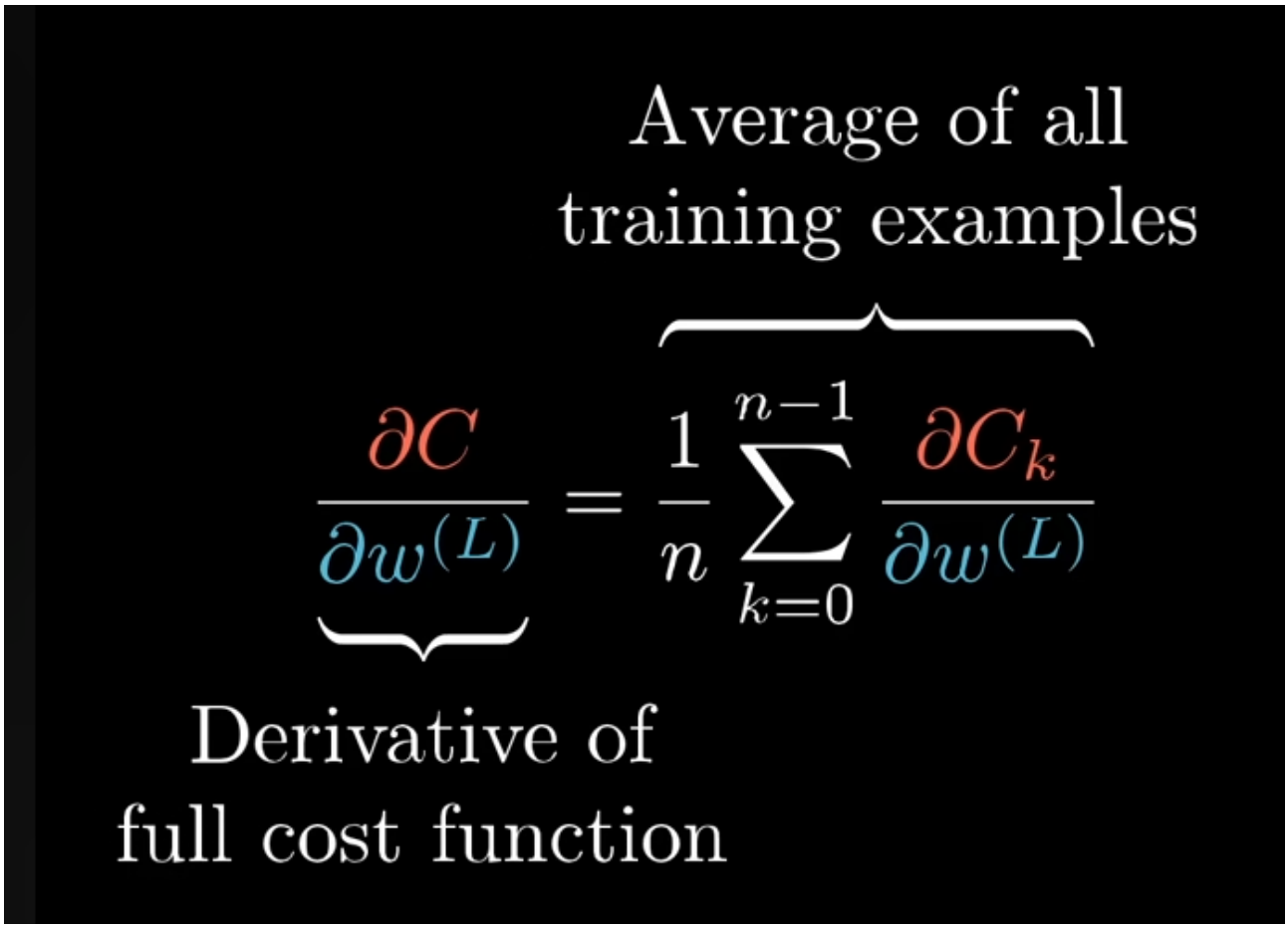


It means “A LITTLE LOT”. We have to observe something before we go ahead.

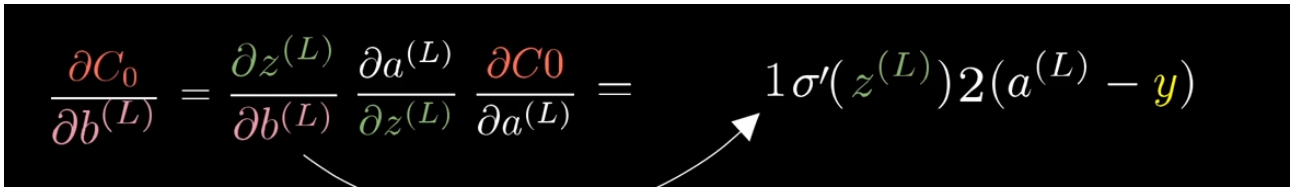


From the last equation, it is obvious that the change in the output is strongly dependent on the previous layers output.

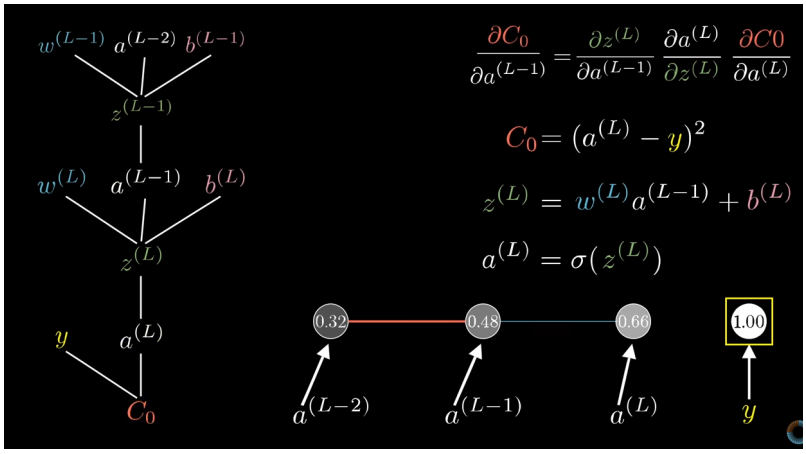
The derivative includes finding the average of all the training samples.



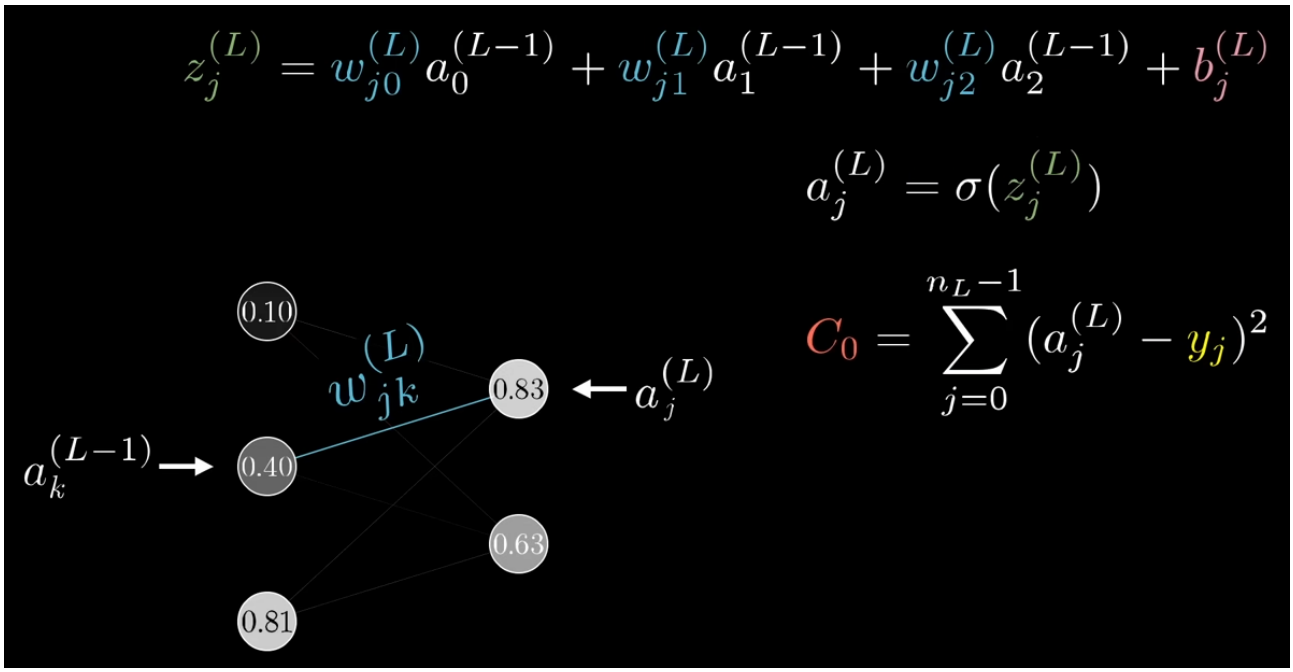
It is identical with the bias values added.



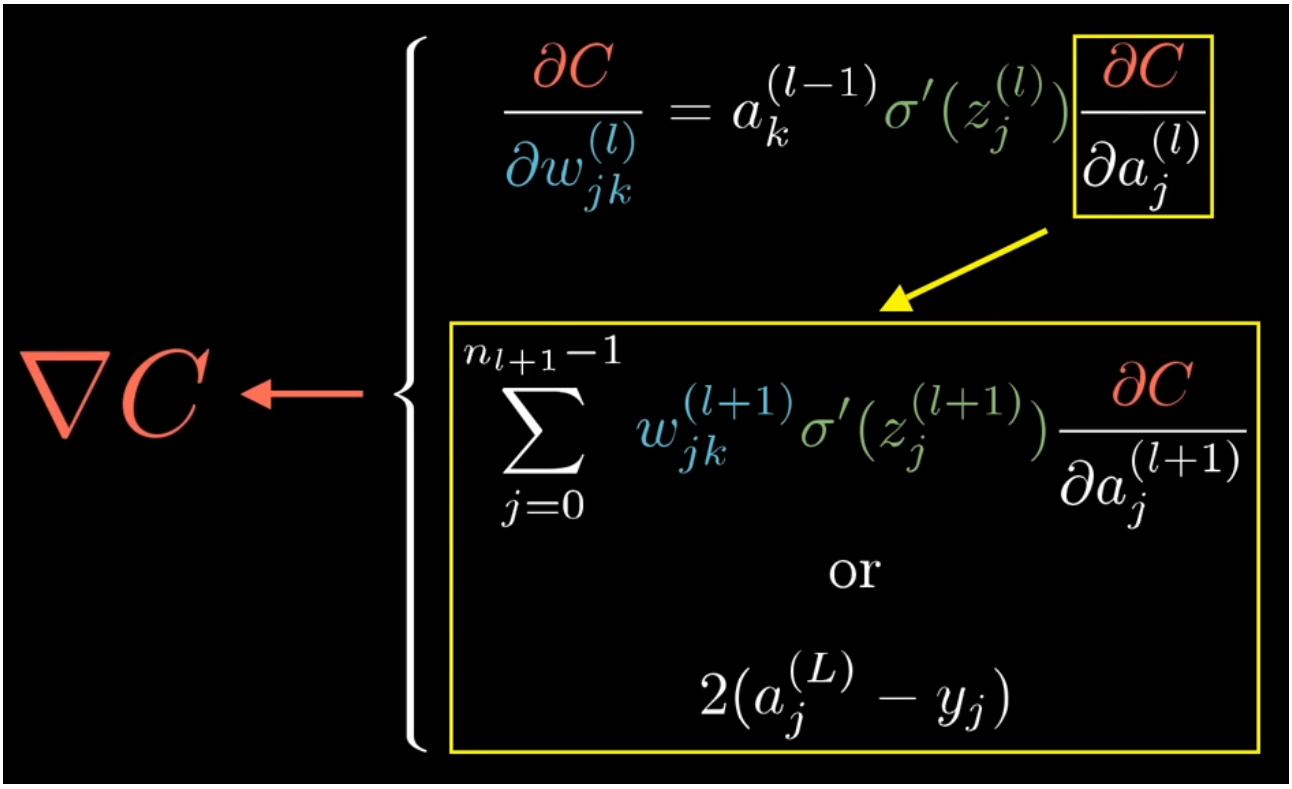
By using this approach, we can compute the same for the previous layers.



For all the layers, we can give a generalised form.



These Chain Rule expressions determine the each component in the gradient that helps minimize the cost of the network by repeatedly stepping down the hill.



★ XOR Operation : Using ANN

```
import numpy as np

def sigmoid (x):
    return 1/(1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 – x)

# Input Datasets
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
expected_output = np.array([[0],[1],[1],[0]])
epochs = 10000
lr = 1
inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2,2,1

# Random Weights and Bias Initialization
hidden_weights = np.random.uniform(size=(inputLayerNeurons,hiddenLayerNeurons))
hidden_bias =np.random.uniform(size=(1,hiddenLayerNeurons))
output_weights = np.random.uniform(size=(hiddenLayerNeurons,outputLayerNeurons))
output_bias = np.random.uniform(size=(1,outputLayerNeurons))

print("Initial hidden weights: ",end='')

print(*hidden_weights)

print("Initial hidden biases: ",end='')

print(*hidden_bias)

print("Initial output weights: ",end='')

print(*output_weights)

print("Initial output biases: ",end='')

print(*output_bias)
```

```
# Training Algorithm
for _ in range(epochs):
      #Forward Propagation
      hidden_layer_activation = np.dot(inputs,hidden_weights)
      hidden_layer_activation += hidden_bias
      hidden_layer_output = sigmoid(hidden_layer_activation)

      output_layer_activation = np.dot(hidden_layer_output,output_weights)
      output_layer_activation += output_bias
      predicted_output = sigmoid(output_layer_activation)

      # Backpropagation
      error = expected_output - predicted_output
      d_predicted_output = error * sigmoid_derivative(predicted_output)
      error_hidden_layer = d_predicted_output.dot(output_weights.T)
      d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

      # Updating Weights and Biases
      output_weights += hidden_layer_output.T.dot(d_predicted_output)* lr
      output_bias += np.sum(d_predicted_output,axis=0,keepdims=True)* lr
      hidden_weights += inputs.T.dot(d_hidden_layer)* lr
      hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True)* lr

print("Final hidden weights: ",end='')
print(*hidden_weights)
print("Final hidden bias: ",end='')
print(*hidden_bias)
print("Final output weights: ",end='')
print(*output_weights)
print("Final output bias: ",end='')
print(*output_bias)

print("\nOutput from neural network after 10,000 epochs: ",end='')
print(*predicted_output)
```

```
Output from neural network after 10,000 epochs: [0.01302119] [0.98887123] [0.98888554] [0.01144319]
```

FINE