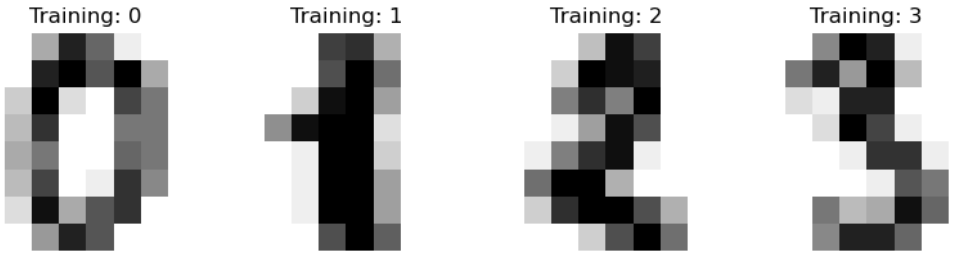
DEEP LEARNING

ARIHARASUDHAN

VISUAL PATTERN RECOG

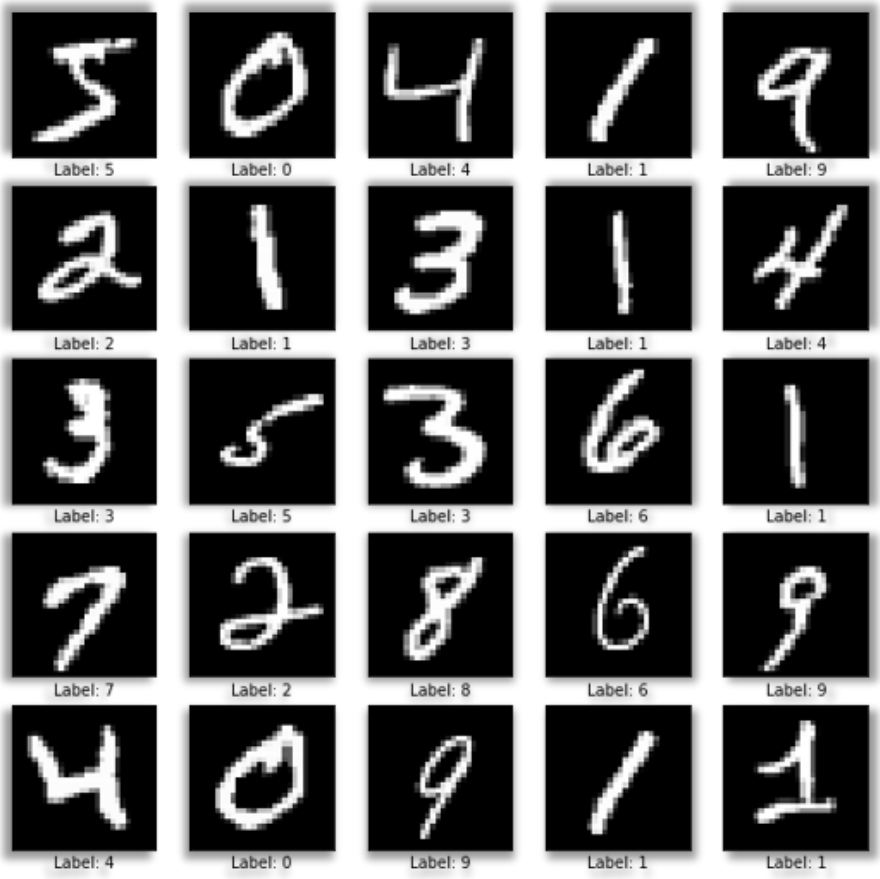
The difficulty of visual pattern recognition becomes apparent if you attempt to write a computer program to recognize digits like those below.



What seems easy when we do it ourselves suddenly becomes extremely difficult. Simple intuitions about how we recogni]e shapes - 9 has a loop at the top, and a vertical stroke in the bottom right - turn out to be not so simple to express algorithmically. When we try to make such rules precise, we get lost in an ocean of sharks namely exceptions which seems hopeless.

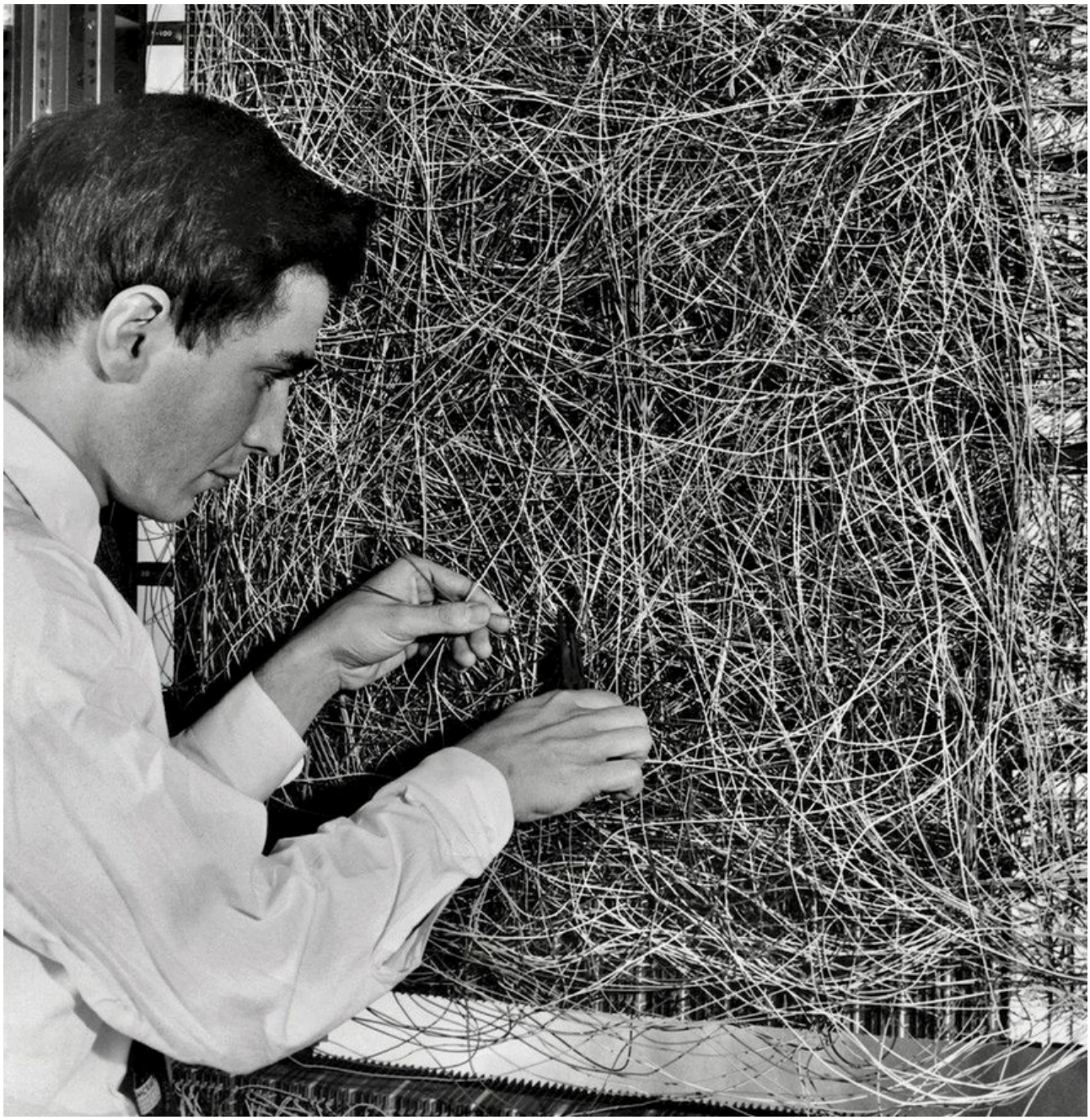
THE NEURAL NETWORK

Neural networks approach the problem in a different way. The idea is to take a large number of handwritten digits, known as training examples, and then develop a system which can learn from those training examples. In other words, the neural network uses the examples to automatically infer rules for recognizing handwritten digits. Furthermore, by increasing the number of training examples, the network can learn more about handwriting, and so improve its accuracy.

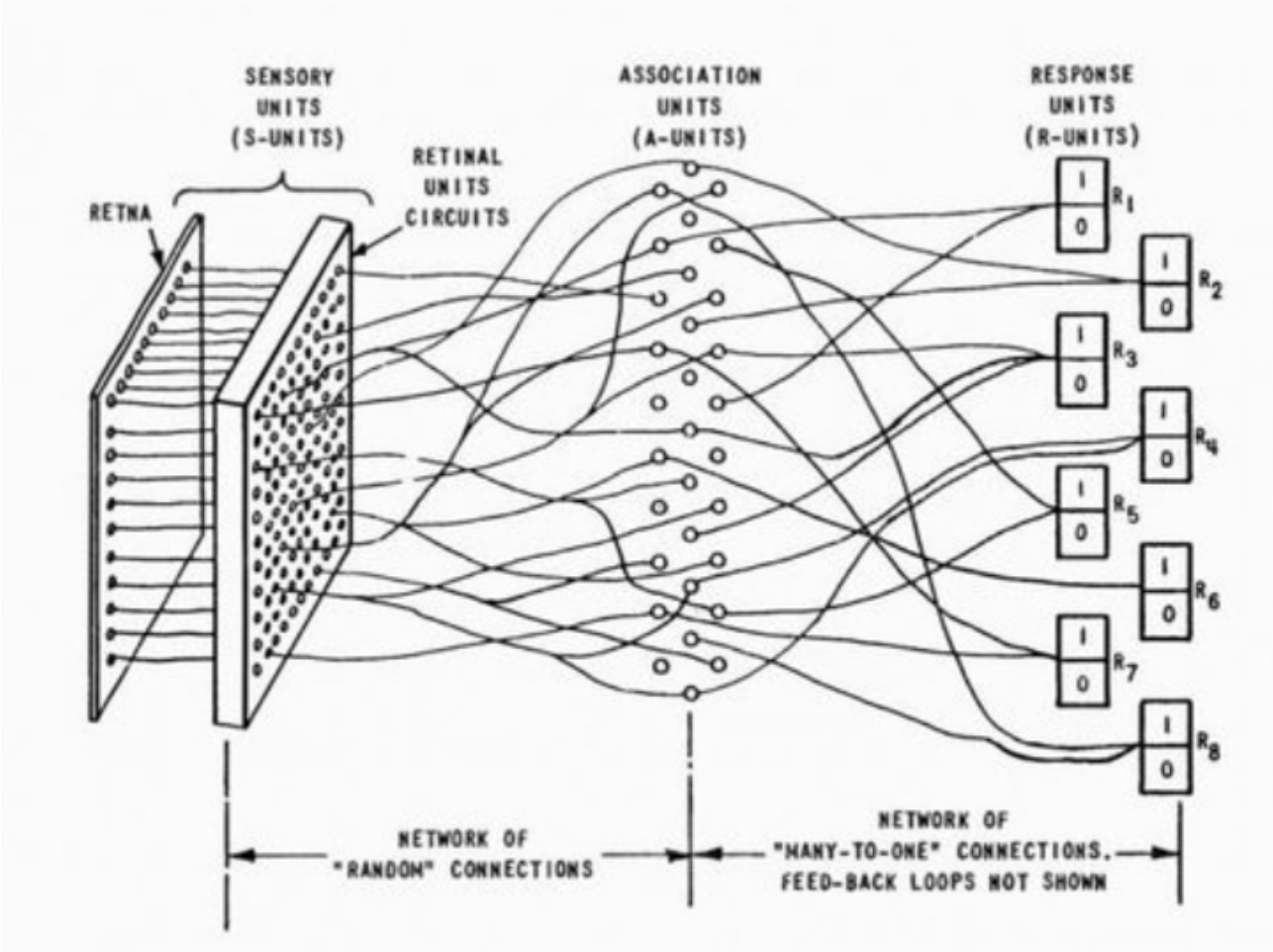


THE PERCEPTRON

A Perceptron takes several binary inputs, and produces a single binary output. The first Neural Network (Perceptron) was created by Mr.Frank Rosenblatt.

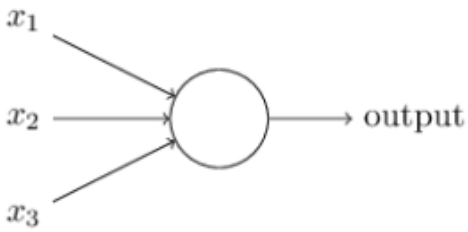


He introduced the concept of weights.



This award in the name of Frank Rosenblatt is **presented for outstanding contributions to the advancement of the design, practice, techniques, or theory in biologically and linguistically motivated computational paradigms**, including neural networks, connectionist systems, evolutionary computation, fuzzy systems, and hybrid intelligent systems.

A simple perceptron may have multiple inputs. (The below shown has only three inputs)

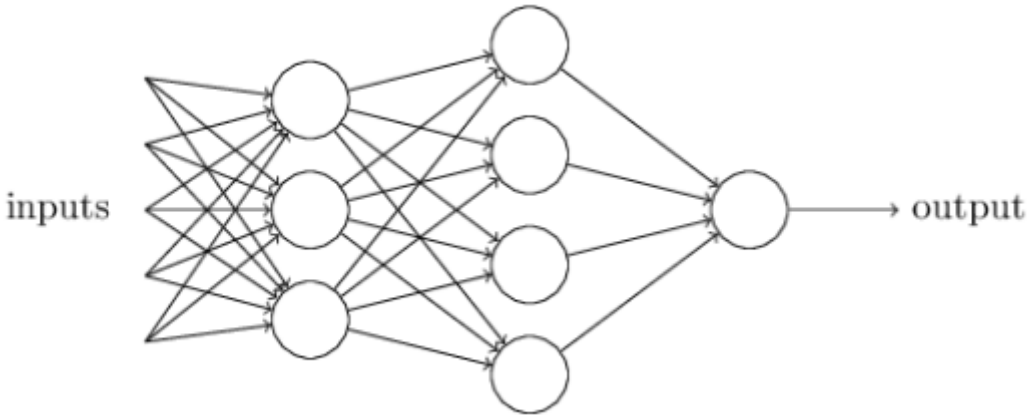


Rosenblatt proposed a simple rule to compute the output. He introduced weights, real numbers expressing the importance of the respective input to the output. The neuron’s output is determined by whether the weighted sum is less than or greater than some threshold value. Just like the weights, the threshold is a real number which is a parameter of the neuron.

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

That's all there is to how a perceptron works!

It should seem plausible that a comple[network of perceptrons could make quite subtle decisions.



A many-layer network of perceptrons can engage in sophisticated decision making.

Let’s simplify the way we describe perceptrons.

$\sum_j w_j x_j > \text{threshold}$ condition, is cumbersome.

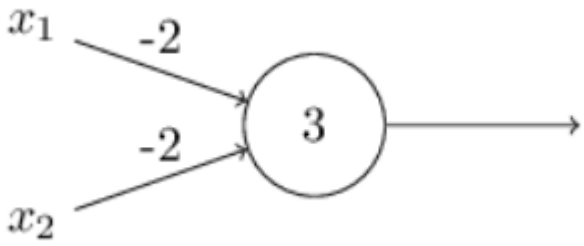
And, we can make two notational changes to simplify it. The first change is to write as a dot product. The second change is to move the threshold to the other side of the inequality, and to replace it by what’s known as the perceptron’s Bias. Using the bias instead of the threshold, the perceptron rule can be rewritten :

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

We can think of the bias as a measure of how easy it is to get the perceptron to output a 1. Or to put it in more biological terms, the bias is a measure of how easy it is to get the perceptron to Iire. For a perceptron with a really big bias, it s extremely easy for the perceptron to output a 1. But if the bias is very negative, then it is difficult for the perceptron to output a 1.

PERCEPTRON FOR GATE

A Perceptron can be used for logical operations such as NAND, OR and so on. For an instance, if we take a look at the following,



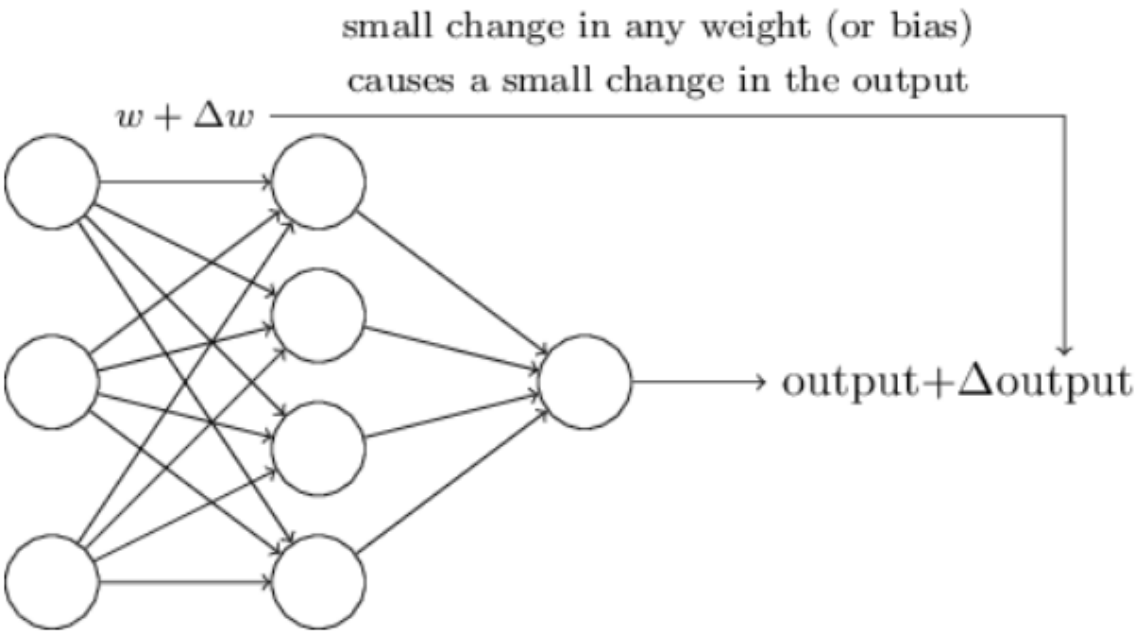
Suppose we have a perceptron with two inputs, each with weight -2, and an overall bias of 3.

1	1	(-2*1)+(-2*1)+3	0
1	0	(-2*1)+(-2*0)+3	1
0	1	(-2*0)+(-2*1)+3	1
0	0	(-2*1)+(-2*0)+3	1

It works as a NAND Gate. It means we can implement other gates also.

SIGMOID NEURON

Suppose we have a network of perceptrons that we'd like to use to learn to solve some problem. For example, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. And we'd like the network to learn weights and biases so that the output from the network correctly classifies the digit. To see how learning might work, suppose we make a small change in some weight or bias in the network. What we'd like is for this small change in weight to cause only a small corresponding change in the output from the network.



This property will make learning possible.

The problem is that this isn t what happens when our network contains perceptrons. ,n fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip.

We can overcome this problem by introducing a new type of artificial neuron called a sigmoid neuron. 6igmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output.

Just like a perceptron, the sigmoid neuron has inputs but instead of being just 0 or 1, these inputs can also take on any values between 0 and 1 say 0.63234.
Also just like a perceptron, the sigmoid neuron has weights for each input and an overall bias. But, the output is not 0 or 1. Instead, it's the sigma of $w_i x_i + b$ where sigma() is called the sigmoid function and is defined by :

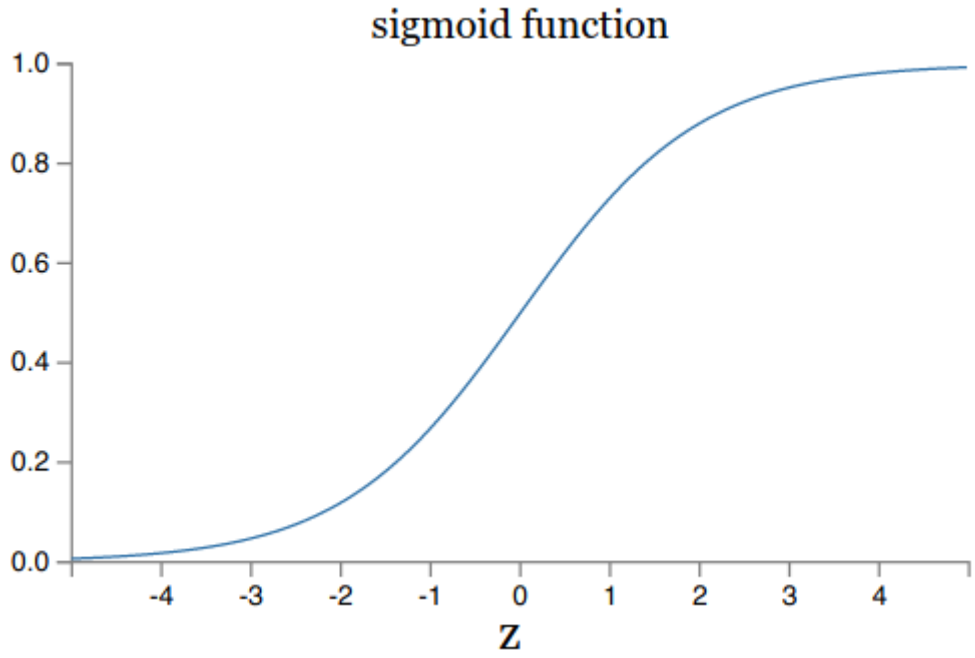
$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

To put it a little bit explicitly,

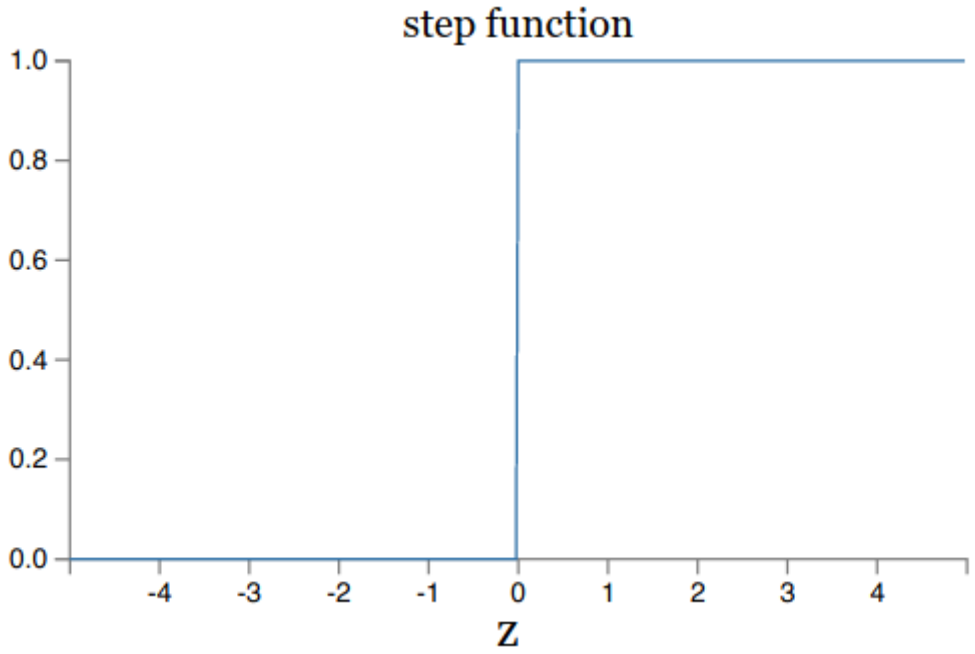
$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}.$$

It resembles a perceptron when the z has a positive large value or a very negative value.

When plotted, it resembles the following smooth curve.



It is a smoothened form of the following step function.

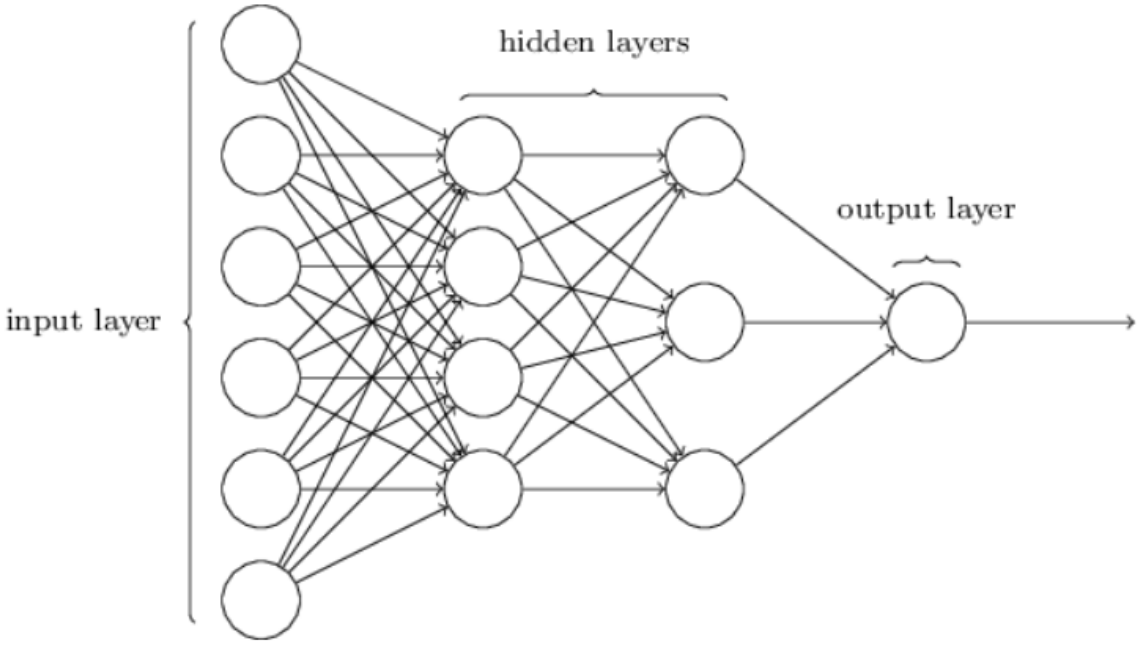


The smoothness of σ means that small changes Δw in the weights and Δb in the bias will produce a small change in the output from the neuron.In fact, calculus tells us that Δoutput is well approximated by,

$$\Delta \text{output} \approx \sum_j \frac{\partial \,\text{output}}{\partial w_j} \Delta w_j + \frac{\partial \,\text{output}}{\partial b} \Delta b,$$

NEURAL NETWORK ARCHITECTURE

We have the following three layers.



The above given neural network has two hidden layers. These kind of networks with more than one hidden layer is called a MultiLayer Perceptrons or MLP’s.

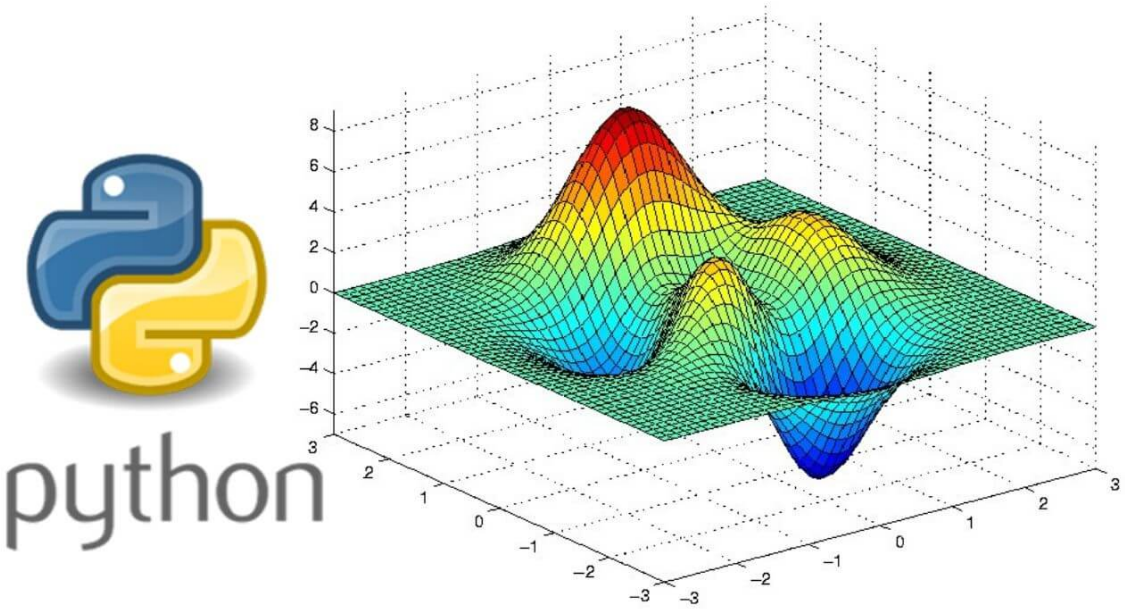
Despite being made up of sigmoid neurons, not perceptrons, this is called a MLP’s in common. These neural networks we’re discussing about are where the output from one layer is used as input to the next layer. Such networks are called Feed Forward neural networks. This means there are no loops. However, there are other models of artificial neural networks in which feedback loops are possible. These models are called recurrent neural networks. The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent. That firing can stimulate other neurons, which may fire a little while later, also for a limited duration. That causes still more neurons to fire, and so over time we get a cascade of neurons firing. Loops don t cause problems in such a model, since a neurons output only affects its input at some later time, not instantaneously.

GRADIENT DESCENT

What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates y(x) for all training inputs, x. To quantify how well we re achieving this goal we define a Lost Function :

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

It is called Mean Squared Error. Whenever the C(w,b) yields a high value, we are not even near to the output. So, we have to try to reduce the cost function value. In other words, we want to find a set of weight and bias values to make the cost function minimized. We can achieve that using the Gradient Descent function.



Now, of course, for the function plotted above, we can eyeball the graph and find the minimum. For a complicated function of many variables, we can’t hope eyeballing the graph to find the minimum.

Definitely, calculus won’t help for a complex function to find the minimum. So, what we can do ? We can imagine a ball rolling down from the peak of a mountain. In which direction , will it flow fast to attain the minimum.

We would randomly choose a starting point for an imaginary ball, and then simulate the motion of the ball as it rolled down to the bottom of the valley. We could do this simulation simply by computing derivatives and (perhaps some second derivatives of C) those derivatives would tell us everything we need to know about the local shape of the valley, and therefore how our ball should roll. If we change the ball's direction to Δv1 in v1 direction and Δv2 in v2 direction. The calculus tells us the change in the cost.

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2.$$

Let's say the $\Delta v = [\ \Delta v2\ \Delta v1\ ]^T$.
Similarly, we can denote the gradient vector to be,

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T.$$

So, we can rewrite as,

$$\Delta C \approx \nabla C \cdot \Delta v.$$

If we do something like,

$$\Delta v = -\eta \nabla C,$$

Then, the considerable change is that,

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2.$$

The value of gradient descent is always negative which we need. It means C will always decrease never increase.
Here η is a small, positive parameter known as the learning rate . We can observer that the value of v changes by Δv.
(Change in v= v-Δv)
So ultimately, we conclude with,

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$

By repeatedly applying this update rule we can roll down the hill, and hopefully find a minimum of the cost function. In other words, this is a rule which can be used to learn in a neural network. But, an important problem is that it will become so slow with larger extent of inputs.

STOCHASTIC GRADIENT DESCENT

An idea called stochastic gradient descent can be used to speed up learning. The idea is to estimate the gradient by computing for a small sample of randomly chosen training inputs. By averaging over this small sample it turns out that we can quickly get a good estimate of the true gradient , and this helps speed up gradient descent, and thus learning. We'll label those random training inputs X1,X2,...Xn and refer to them as a Mini-Patch. (Provided the sample size m is large enough we expect that the average value of the will be roughly equal to the average over all)

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

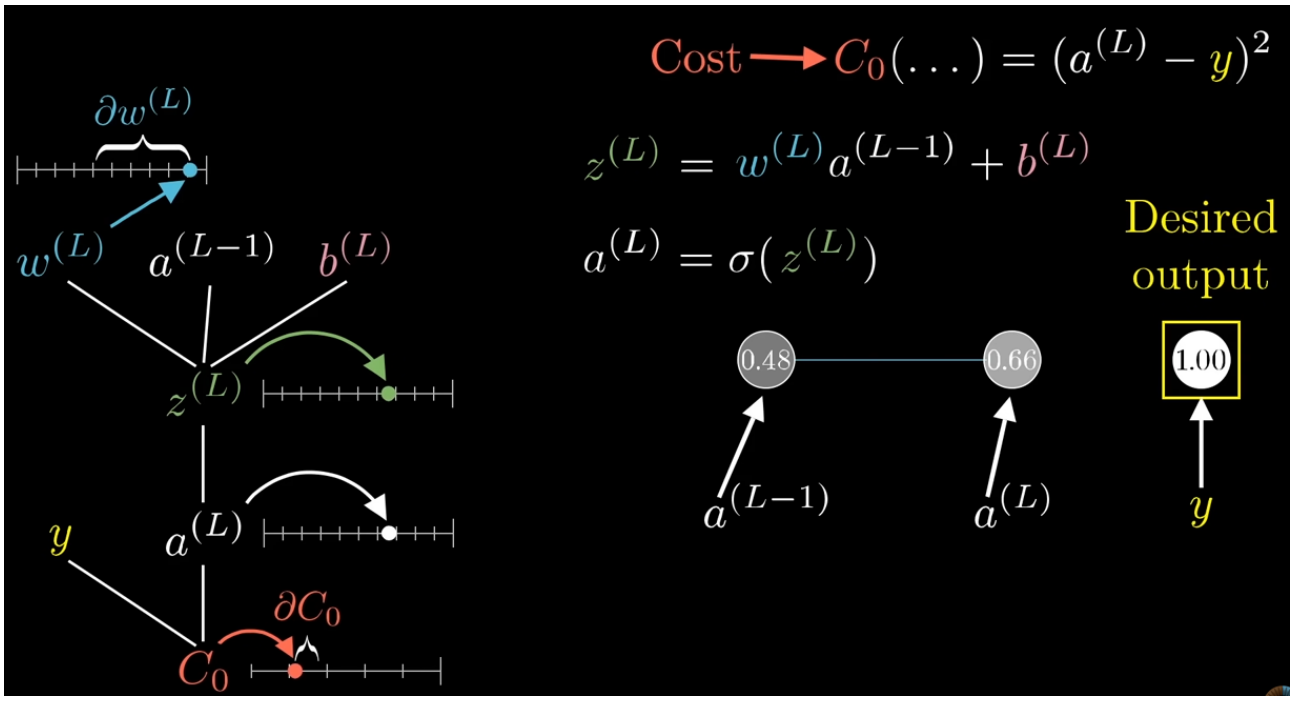
Importants are,

$$w_k \rightarrow w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \rightarrow b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

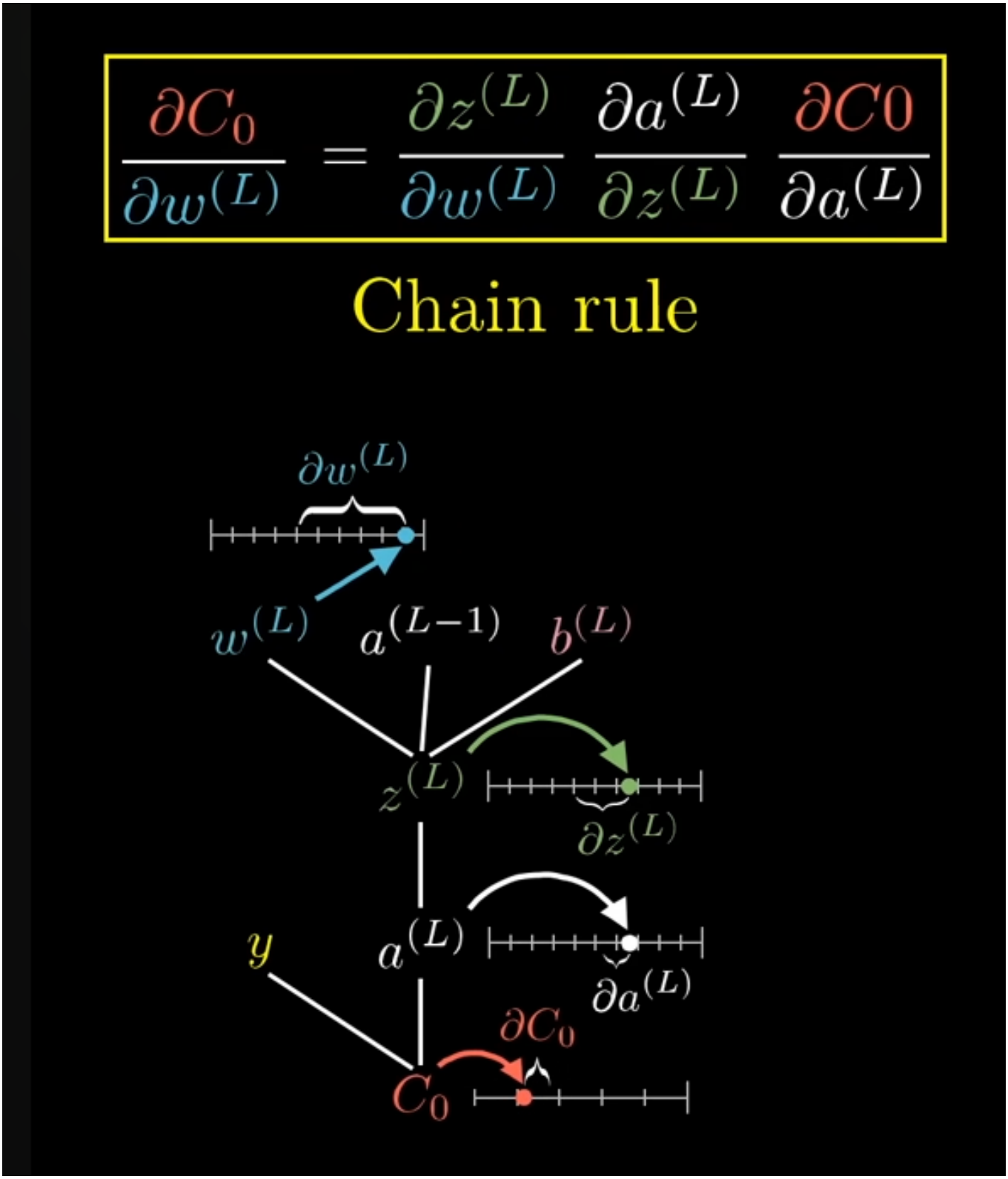
We pick out another randomly chosen mini-batch and train with those. And so on, until we've exhausted the training inputs, which is said to complete an epoch of training. It that point, we start over with a new training epoch.

BACK-PROPAGATION CALCULUS

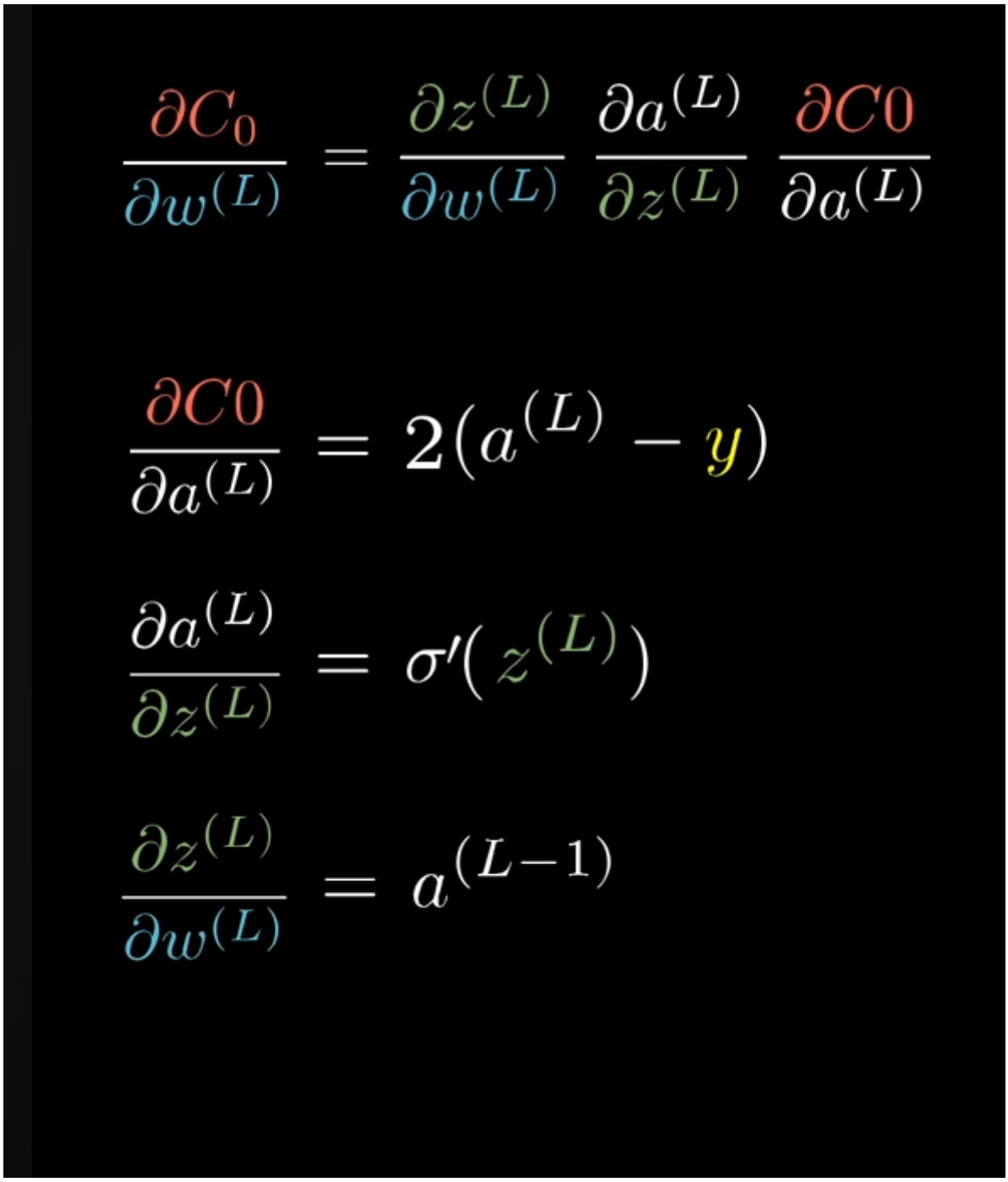


Considering only the two perceptrons of which one is in the last layer and the another is in the second-last layer. The output (The Activation) is found as the above as we have described very well in the past sections of this book. To compute : $\partial C_0/\partial W^{(L)}$

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.

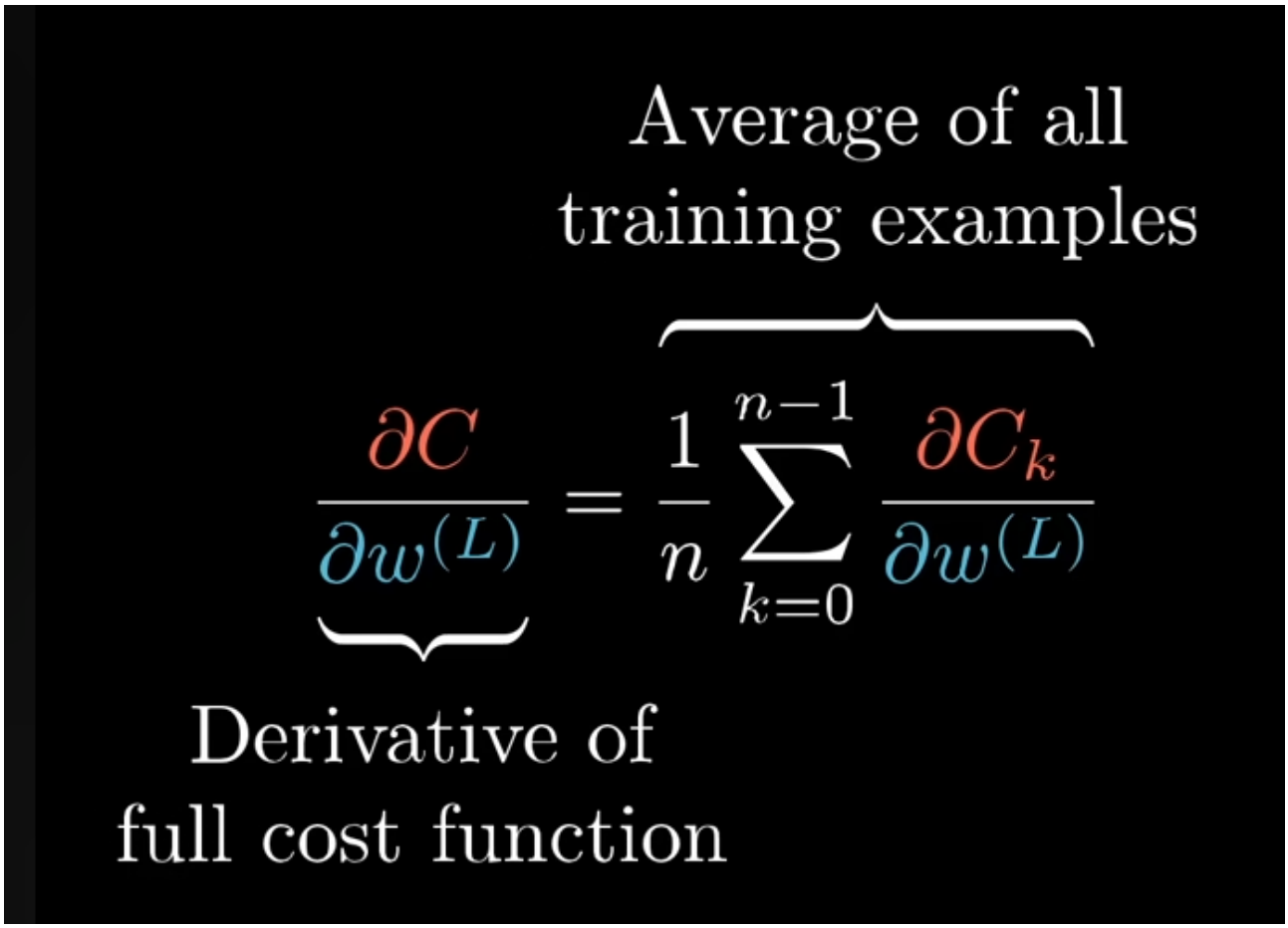


It means “A LITTLE LOT”. We have to observe something before we go ahead.

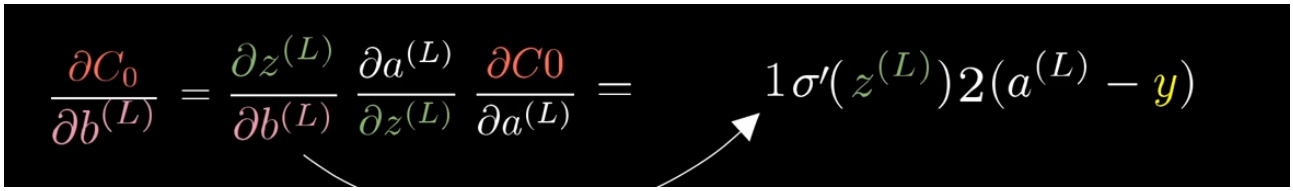


From the last equation, it is obvious that the change in the output is strongly dependent on the previous layers output.

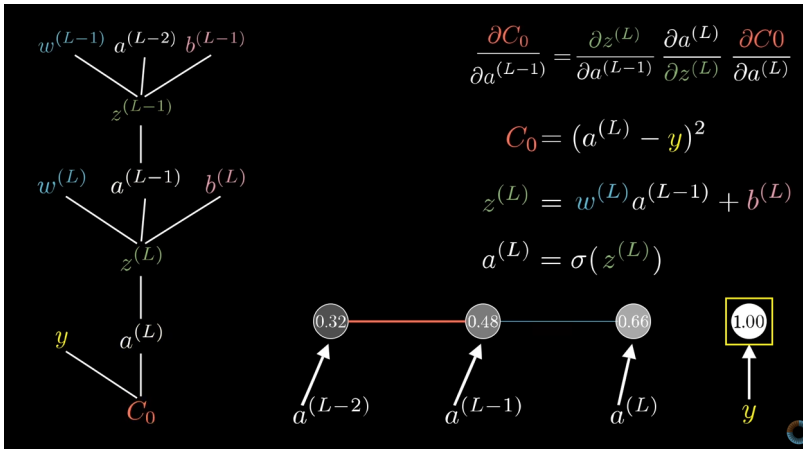
The derivative includes finding the average of all the training samples.



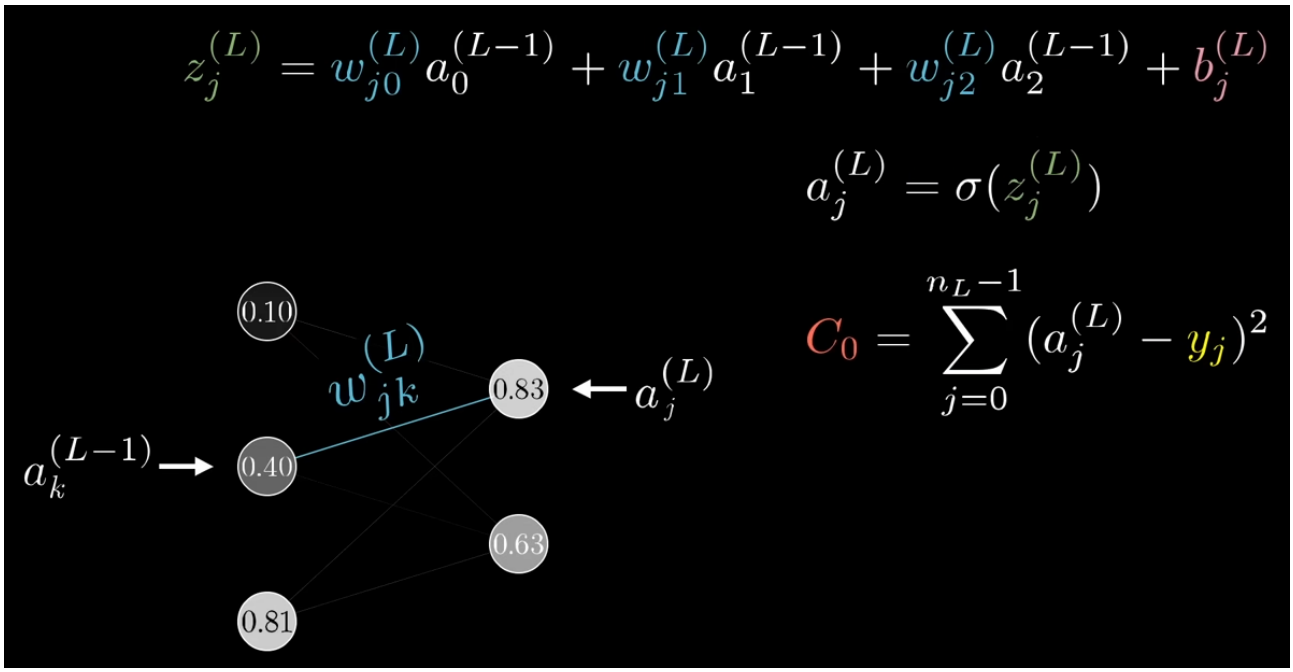
It is identical with the bias values added.



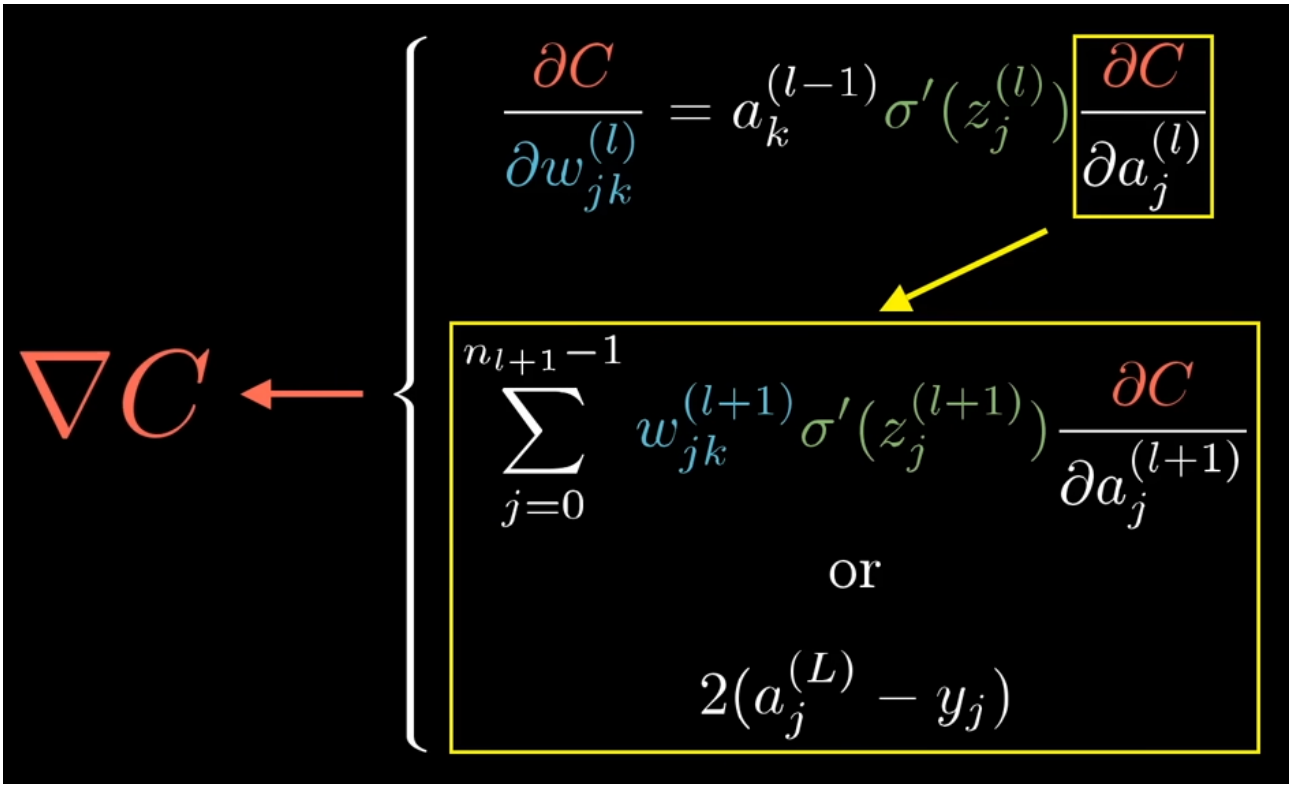
By using this approach, we can compute the same for the previous layers.



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.



Summary: the equations of backpropagation

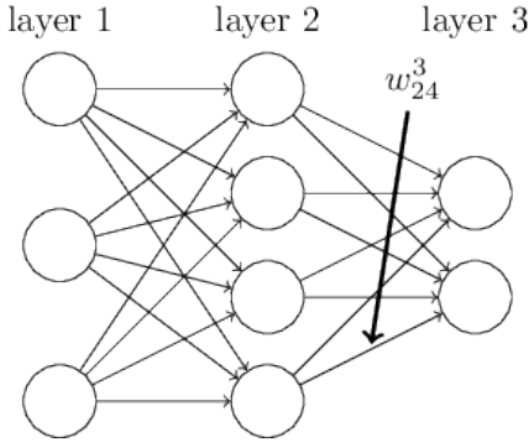
$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

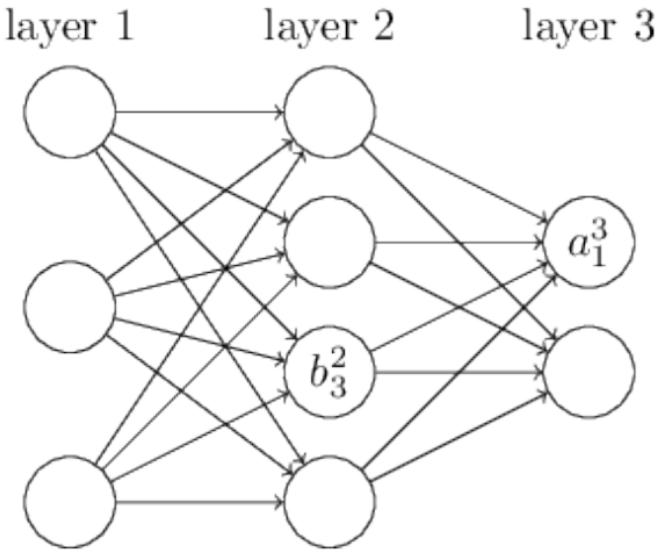
$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

REPRESENTATIONS



w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

Let's follow up the above given notations to identify the each and every neurons.



The activation also can be represented using the following notation then.

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right),$$

It can be compactly written as,

$$a^l = \sigma(w^l a^{l-1} + b^l).$$

WOW!

GOAL OF BACK-PROPAGATION

The Goal of Back-Propagation is to find out the partial derivatives of the Cost with respect to weight and bias. That's all.

$$\partial C / \partial w \text{ and } \partial C / \partial b$$

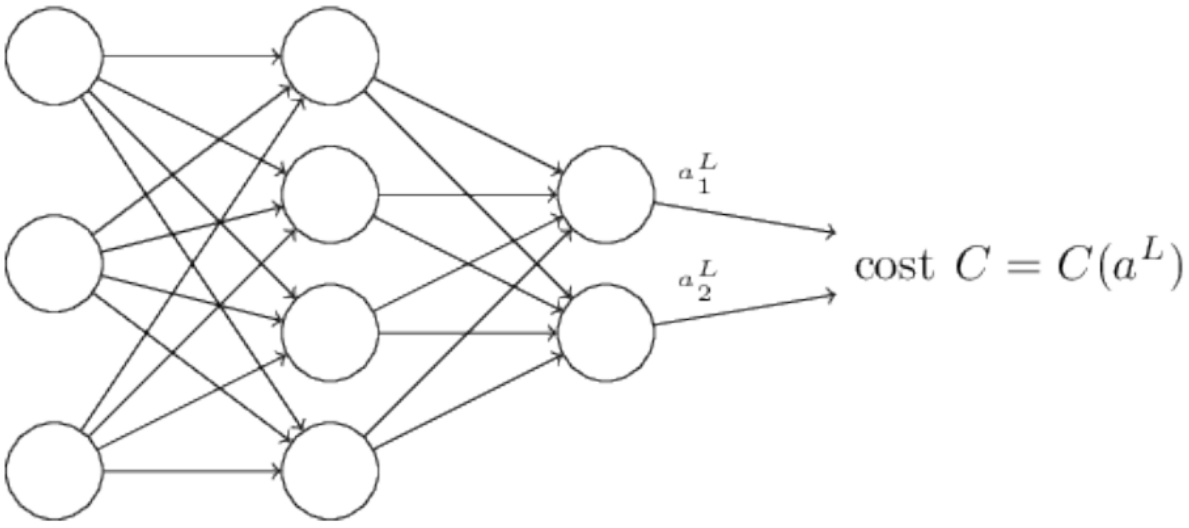
Let's have an example Cost Function. We'll use the quadratic cost function. The quadratic cost has the form,

$$C = \frac{1}{2n} \sum_x \left\| y(x) - a^L(x) \right\|^2$$

Where n is the total number of training examples; y = y(x) is the corresponding desired output; L denotes the number of layers in the network; and $a^L = a^L(x)$ is the vector of activations output from the network when x is input. The two main assumptions we need to make over the cost function is that,

>>> The cost can be written as the average of the costs for each individual training examples.

>>> Secondly, the cost is the function of the output from the neural network.



The quadratic cost function satisfies this requirement, since the quadratic cost for a single training example x may be written as

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2,$$

THE HADAMARD PRODUCT

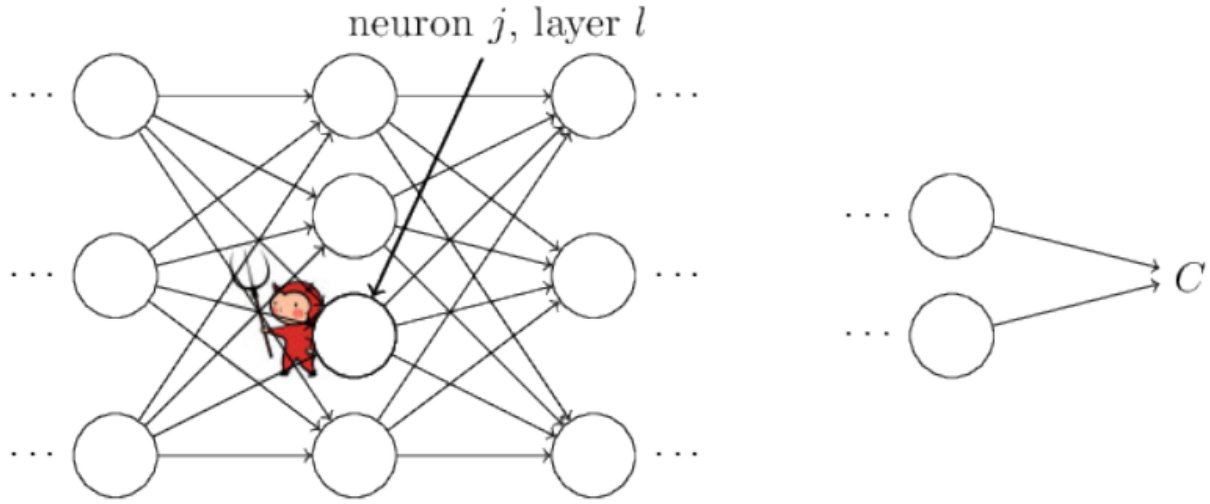
The backpropagation algorithm is based on common linear algebraic operations – things like vector addition, multiplying a vector by a matrix, and so on. But one of the operations is a little less commonly used. In particular, suppose s and t are two vectors of the same dimension. Then we use s t to denote the elementwise product of the two vectors. Thus the components of s t are just $(s.t)_j = s_j.t_j$. As an example,

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1*3 \\ 2*4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}.$$

This kind of elementwise multiplication is sometimes called the Hadamard product or Schur product. We'll refer to it as the Hadamard product. Good matrix libraries usually provide fast implementations of the Hadamard product, and that comes in handy when implementing backpropagation.

BACK-PROPAGATION CALCULUS II

Backpropagation is about understanding how changing the weights and biases in a network changes the cost function. Ultimately, this means computing the partial derivatives ∂C/∂w and ∂ C/∂ b . To compute these, we have to compute an intermediate value , The Error. To understand it, let's imagine a demon in our neural network as shown below.



The demon sits at the j-th neuron in layer l. As the input to the neuron comes in, the demon messes with the neuron's operation. It adds a little change Δz to the neuron's weighted input, so that instead of outputting σ(z), the neuron instead outputs σ(z + Δz).

This change propagates through later layers in the network, finally causing the overall cost to change by an amount ,

$$\frac{\partial C}{\partial z_j^l} \Delta z_j^l.$$

Generally we can define the error as,

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}.$$

This is the change in the cost with respective to the change in the activation function. But, in the output layer, it can be seen as:

※ $$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L).$$

This is a very natural expression. The first term on the right, ∂C/∂a , just measures how fast the cost is changing as a function of the j-th output activation.

If, for example, C doesn't depend much on a particular output neuron, j, then δ will be small, which is what we'd expect. The second term on the right, σ', measures how fast the activation function σ is changing. Besides, this equation can be written in a matrix form as shown below,

$$\delta^L = \nabla_a C \odot \sigma'(z^L).$$

In the case of the quadratic cost we have ∇a C = (a^L − y), and so the fully matrix-based form is,

$$\delta^L = (a^L - y) \odot \sigma'(z^L).$$

An equation for the error δ l in terms of the error in the next layer, δ^{l+1}

※
$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l),$$

We have an equation which describes the change in the cost with respective to the change in the bias value.

※
$$\frac{\partial C}{\partial b^l_j} = \delta^l_j.$$

An equation for the rate of change of the cost with respect to any weight in the network: In particular :

※ $$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j.$$

BACK-PROPAGATION ALGORITHM

An equation for the rate of change of the cost with respect to any weight in the network :

1. **Input** x: Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z_l)$.
3. **Output error** δ^L: Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. **Backpropagate the error**: For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5. **Output**: The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

SIMPLE NEURAL NETWORK

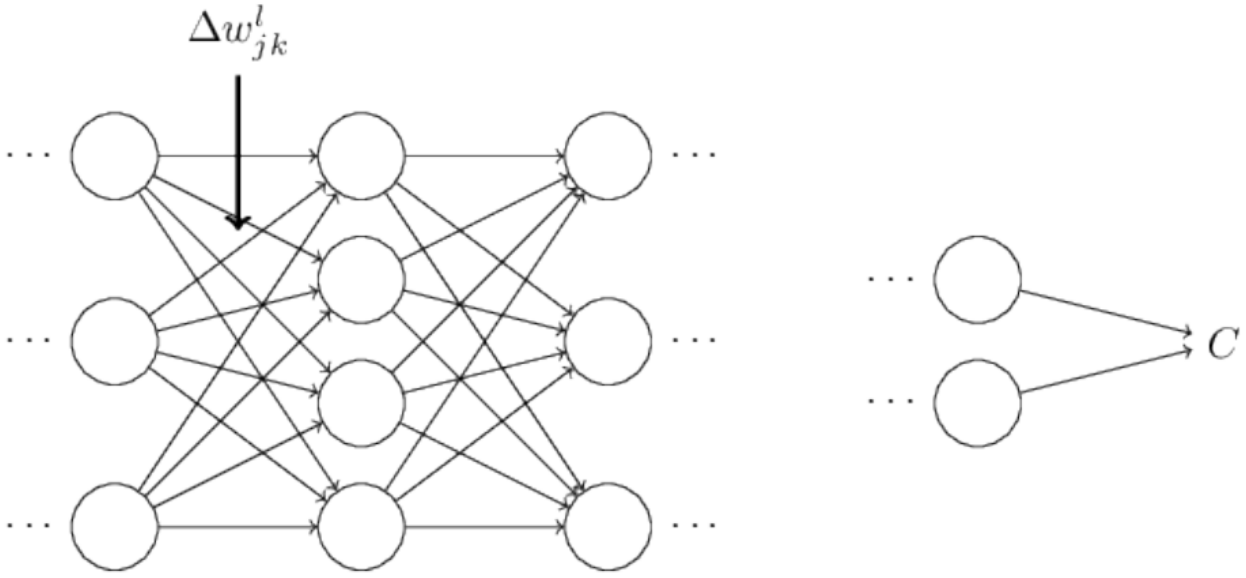
```python
import numpy as np

class Network( object ):
    # INTIALIZE THE WEIGHTS AND BIASES
    def __init__ (self , sizes ):
        self. num_layers = len( sizes )
        self.sizes = sizes
        self. biases = [np. random . randn (y, 1) for y in sizes [1:]]
        self. weights = [np. random . randn (y, x) for x, y in zip( sizes [: -1] , sizes [1:]) ]

    # FEED FORWARD
    def feedforward(self , a):
        for b, w in zip(self.biases , self. weights ):
            a = sigmoid (np.dot(w, a)+b)
        return a

    # STOCHASTIC GRADIENT FUNCTION
    # Training Data : [(1h,1),(2h,2),(3h,3),.........,(9h,9)]
    def SGD(self , training_data , epochs , mini_batch_size , eta , test_data =None):
        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))
```

```python
    def update_mini_batch (self , mini_batch , eta):
        nabla_b = [np. zeros (b. shape ) for b in self. biases ]
        nabla_w = [np. zeros (w. shape ) for w in self. weights ]
        for x, y in mini_batch :
            delta_nabla_b , delta_nabla_w = self. backprop (x, y)
            nabla_b = [nb+dnb for nb , dnb in zip(nabla_b , delta_nabla_b )]
            nabla_w = [nw+dnw for nw , dnw in zip(nabla_w , delta_nabla_w )]
        self. weights = [w -( eta/len( mini_batch ))*nw for w, nw in zip(self.weights , nabla_w )]
        self. biases = [b -( eta/len( mini_batch ))*nb for b, nb in zip(self.biases , nabla_b )]

    def backprop (self , x, y):
        nabla_b = [np. zeros (b. shape ) for b in self. biases ]
        nabla_w = [np. zeros (w. shape ) for w in self. weights ]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations , layer by layer
        zs = [] # list to store all the z vectors , layer by layer
        for b, w in zip(self.biases , self. weights ):
            z = np.dot(w, activation )+b
            zs. append (z)
            activation = sigmoid (z)
            activations . append ( activation )
        # backward pass
        delta = self. cost_derivative ( activations [-1], y) * sigmoid_prime (zs [ -1])
        nabla_b [ -1] = delta
        nabla_w [ -1] = np.dot(delta , activations [ -2]. transpose ())
        for l in xrange (2, self. num_layers ):
            z = zs[-l]
            sp = sigmoid_prime (z)
            delta = np.dot(self. weights [-l+1]. transpose () , delta ) * sp
            nabla_b [-l] = delta
            nabla_w [-l] = np.dot(delta , activations [-l -1]. transpose ())
        return (nabla_b , nabla_w )

    def cost_derivative (self , output_activations , y):
        return ( output_activations -y)
```

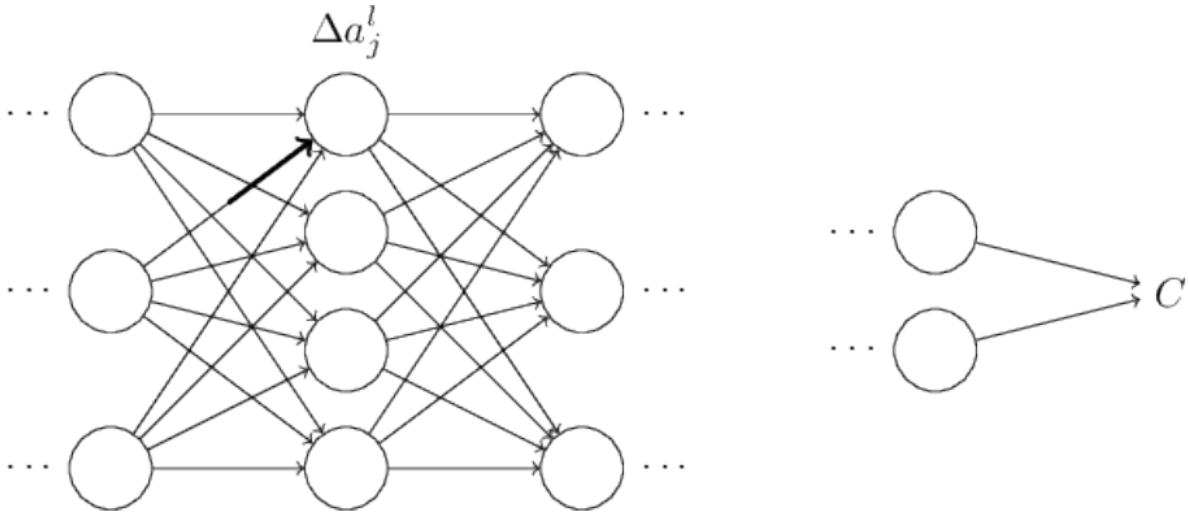
```python
def sigmoid (z):
    return 1.0/(1.0+ np.exp(-z))

def sigmoid_prime (z):
    return sigmoid (z)*(1 - sigmoid (z))
```

THE BIG PICTURE

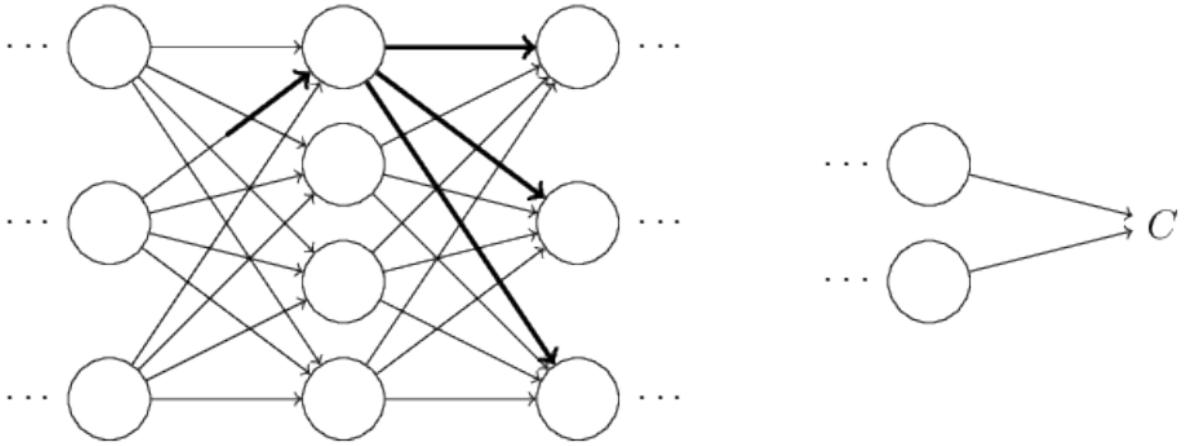
To improve our intuition about what the algorithm is doing, let's imagine that we've made a small change Δw to some weight in the network, w :



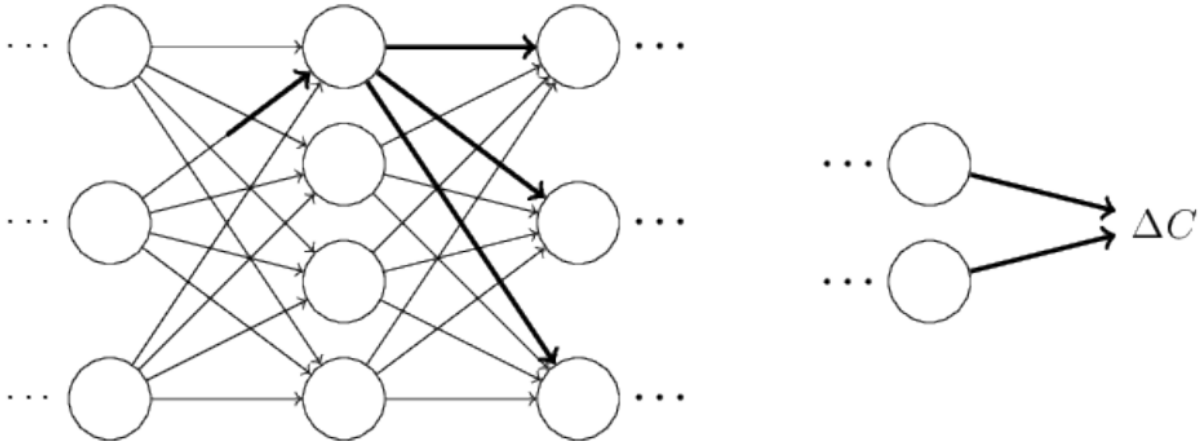
That change in weight will cause a change in the output activation from the corresponding neuron :



That, in turn, will cause a change in all the activations in the next layer :



Those changes will in turn cause changes in the next layer, and then the next, and so on all the way through to causing a change in the final layer, and then in the cost function :



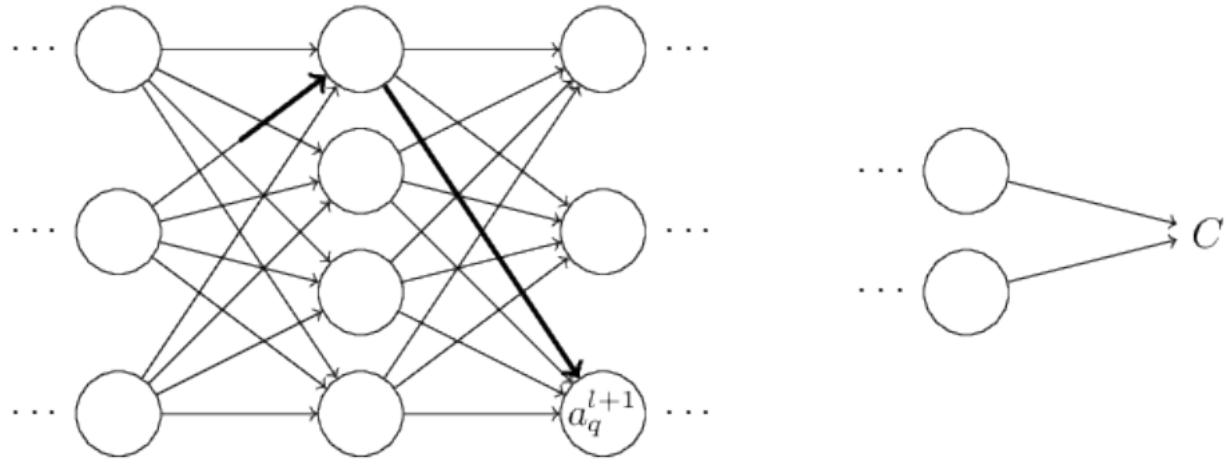
The change ΔC in the cost is related to the change Δw in the weight by the equation :

$$\Delta C \approx \frac{\partial C}{\partial w^l_{jk}} \Delta w^l_{jk}.$$

The change Δw causes a small change Δa in the activation :

$$\Delta a_j^l \approx \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$

We can consider the following network :

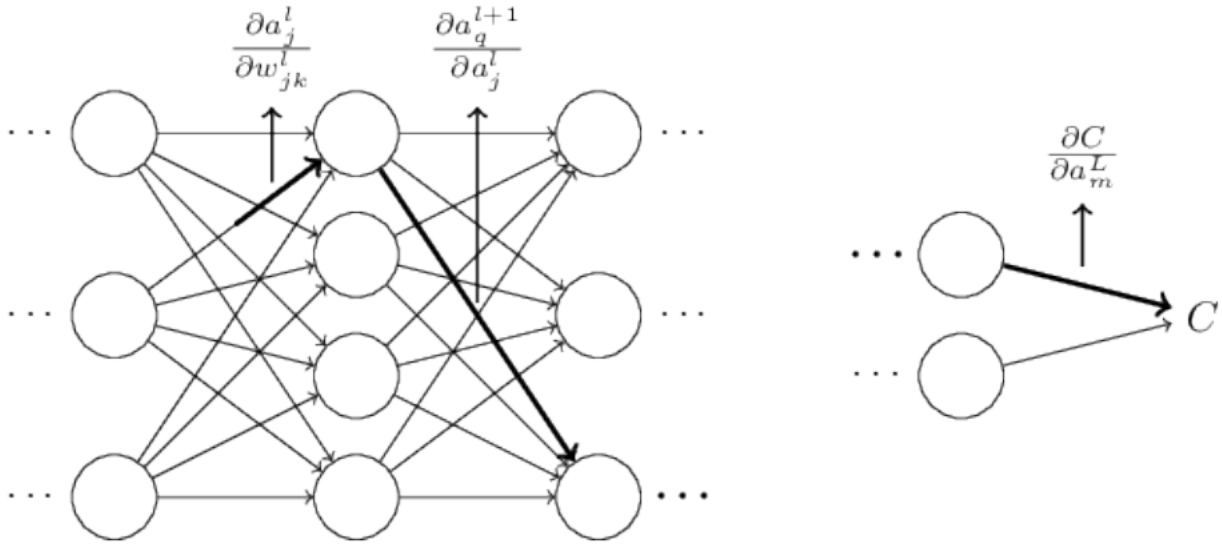


From the truth which states that the change in one layer depends on that of the previous layer ,

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \Delta a_j^l.$$

On substituting the equation,

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$



Collectively, the change in the cost can be written as,

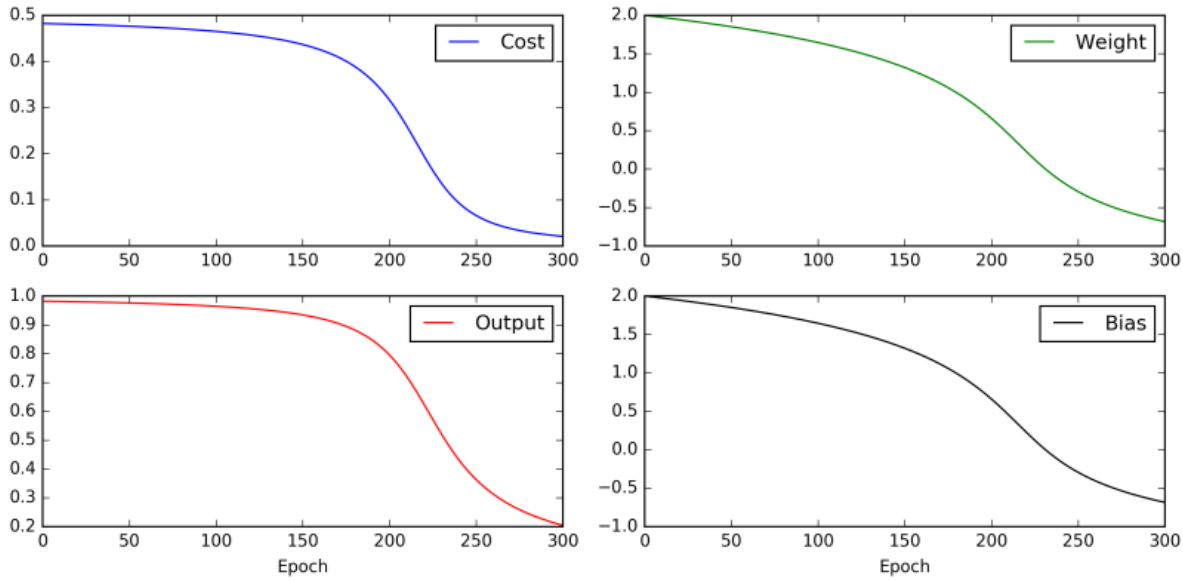
$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l,$$

To compute the total change in C it is plausible that we should sum over all the possible paths between the weight and the final cost, i.e.,

$$\Delta C \approx \sum_{mnp\ldots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l,$$

THE SLOW LEARNER

We the humans learn fast when we are badly wrong. But, the artificial neuron has lot of difficulty in learning.



How can we address the learning slowdown? It turns out that we can solve the problem by replacing the quadratic cost with a different cost function, known as the cross-entropy.

THE CROSS ENTROPY COST

We define the cross-entropy cost function for this neuron by,

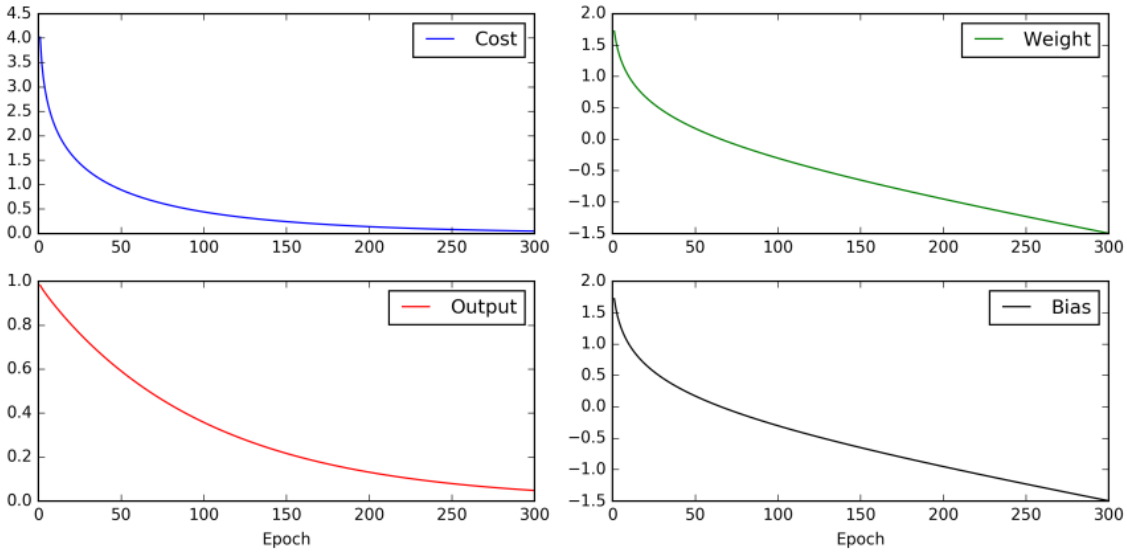
$$C = -\frac{1}{n}\sum_{x}[y \ln a + (1-y)\ln(1-a)],$$

where n is the total number of items of training data, the sum is over all training inputs, x, and y is the corresponding desired output.

Two properties in particular make it reasonable to interpret the cross-entropy as a cost function. First, it's non-negative, that is, C > 0. Second, if the neuron's actual output is close to the desired output for all training inputs, x, then the cross-entropy will be close to zero.

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y).$$

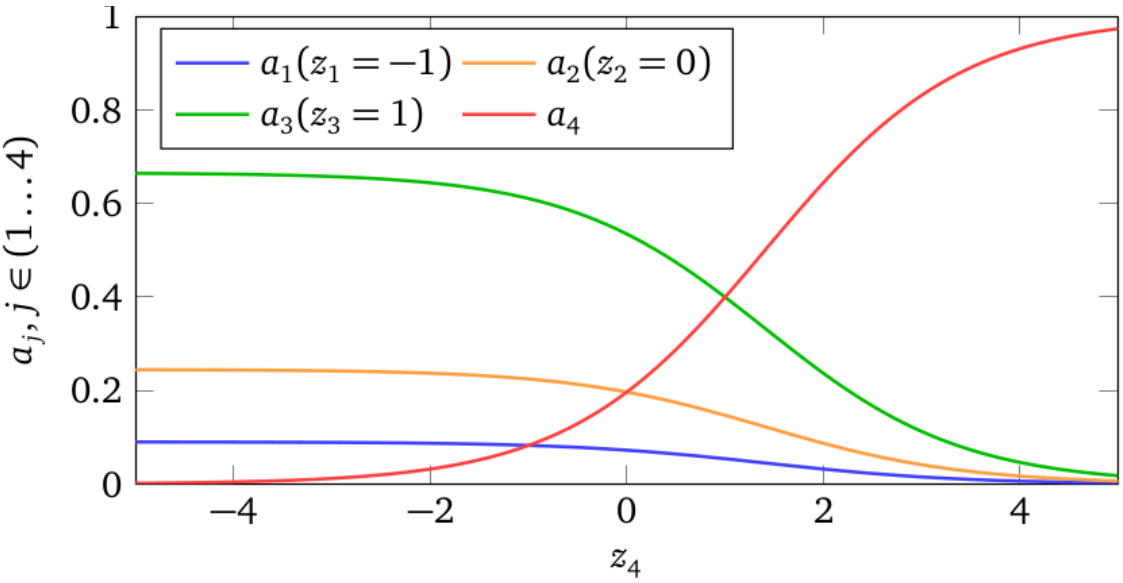
It tells us that the rate at which the weight learns is controlled by σ(z) − y, i.e., by the error in the output. The larger the error, the faster the neuron will learn. This is just what we'd intuitively expect. In particular, it avoids the learning slowdown caused by the σ0 (z) term in the analogous equation for the quadratic cost, Equation.



THE SOFTMAX

The idea of softmax is to define a new type of output layer for our neural networks. It begins in the same way as with a sigmoid layer, by forming the weighted inputs. However, we don't apply the sigmoid function to get the output. Instead, we apply the so-called softmax function to the z. According to this function, the activation a of the output neuron is,

$$a_j^L = \frac{e^{x_j^L}}{\sum_k e^{z_k^L}}$$

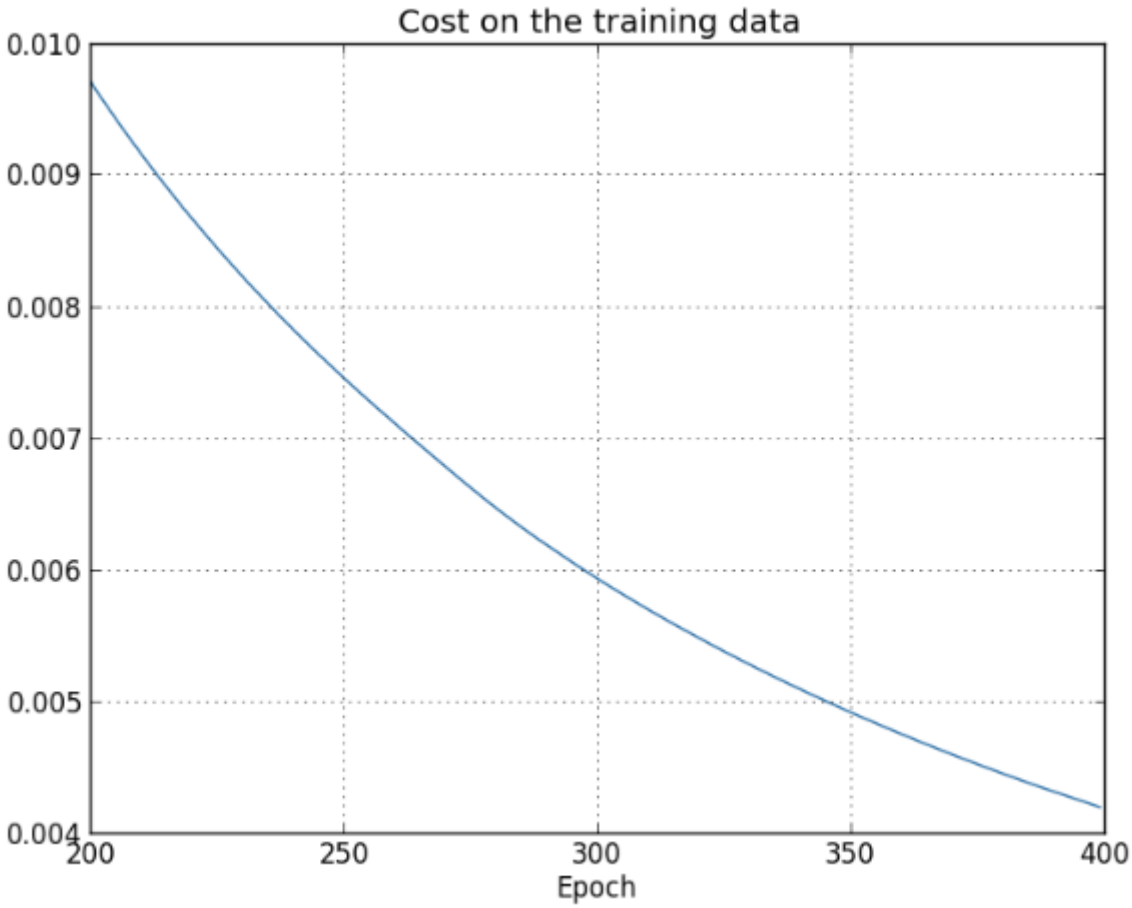


It's also not obvious that this will help us address the learning slow-down problem. The total change in the other activations exactly compensates for the change in a. The reason is that the output activations are guaranteed to always sum up to 1.

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1.$$

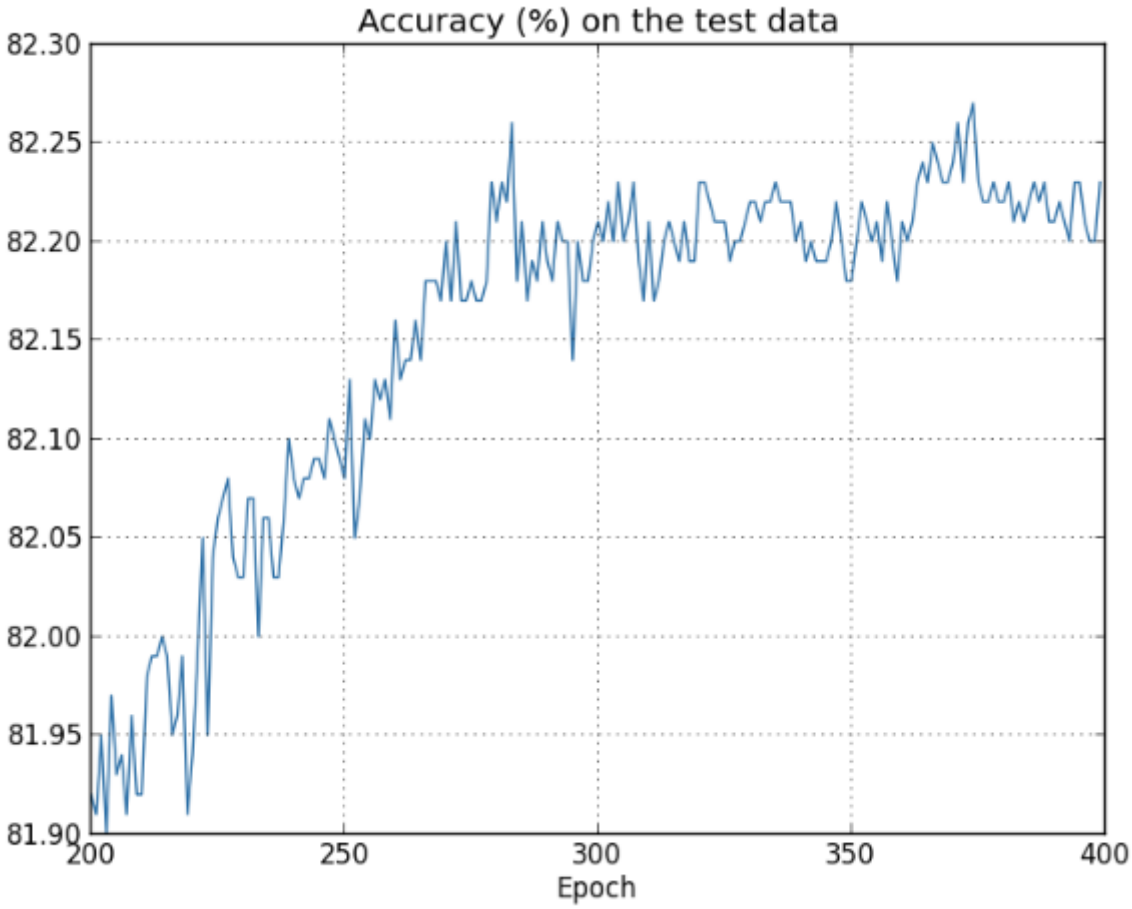
We see that the output from the softmax layer is a set of positive numbers which sum up to 1.

OVER-FITTING



This looks encouraging, showing a smooth decrease in the cost, just as we expect. Note that I've only shown training epochs 200 through 399.

This gives us a nice up-close view of the later stages of learning, which, as we'll see, turns out to be where the interesting action is. Let's now look at how the classification accuracy on the test data changes over time :



If we just look at that cost, it appears that our model is still getting "better". But the test accuracy results show the improvement is an illusion. What our network learns after epoch 280 no longer generalizes to the test data.

So, it’s not a useful learning. We say the network is overfitting or overtraining beyond epoch 280.

OVERCOMING OVER-FITTING

Increasing the training data, reducing the network size are some of the ways to overcome the problem of over-fitting. Ofcourse, larger network does well. We reduce the neural network size reluctantly. There are some other techniques fortunately available to overcome the problem of overfitting. One of them is the so-called Regularization. The most commonly used regularization technique is called as The L2 Regularization or The Weight Decay.

We can achieve this by adding the sum of the squares of the weights in the network which is scaled by λ/2n where λ>0 is known as the regularization parameter and n is the size of the training set. A Quadratic cost function can be regularized as the below.

$$C = \frac{1}{2n} \sum_x \|y - a^L\|^2 + \frac{\lambda}{2n} \sum_w w^2.$$

Again taking the partial derivative will give us the followings.

$$\begin{aligned}\frac{\partial C}{\partial w} &= \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w \\ \frac{\partial C}{\partial b} &= \frac{\partial C_0}{\partial b}.\end{aligned}$$

Clearly on regularization, the gradient descent value for the bias is not changing. But, it changes for the weight.

$$b \to b - \eta \frac{\partial C_0}{\partial b}.$$

The learning rule for the weights becomes:

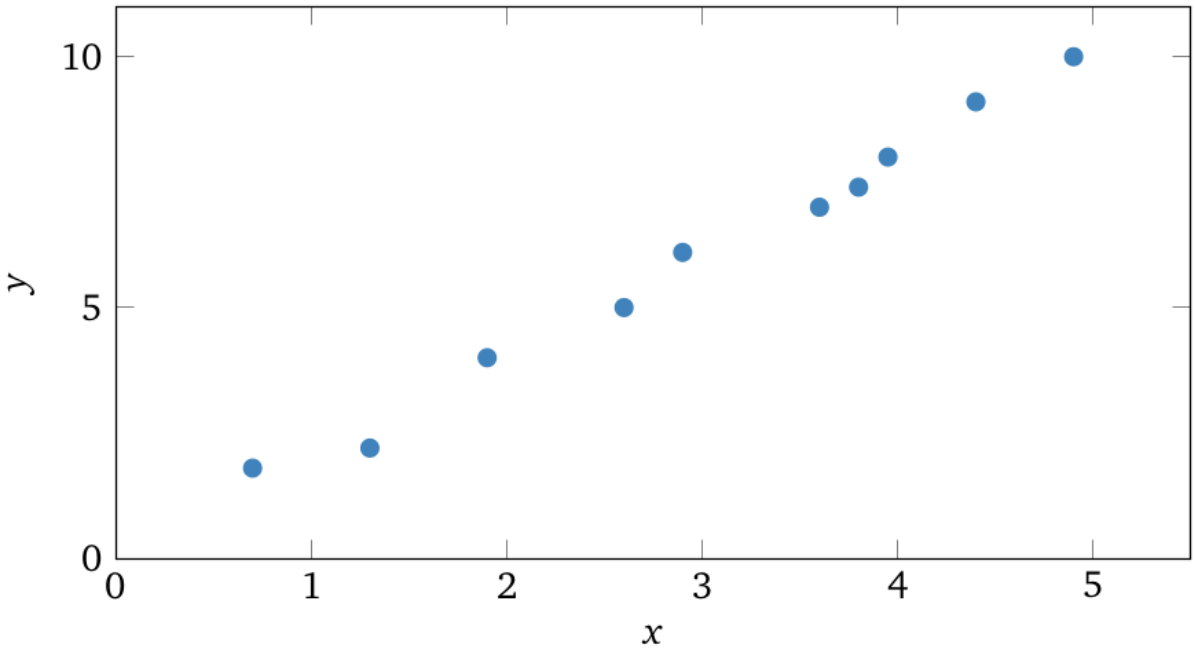
$$w \to w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{n} w = \left(1 - \frac{\eta\lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}.$$

This is exactly the same as the usual gradient descent learning rule, except we first rescale the weight w by a factor (1−ηλ/n). This rescaling is sometimes referred to as weight decay, since it makes the weights smaller. For the biases, the regularized learning rule is that of the unregularized.

$$b \to b - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial b},$$

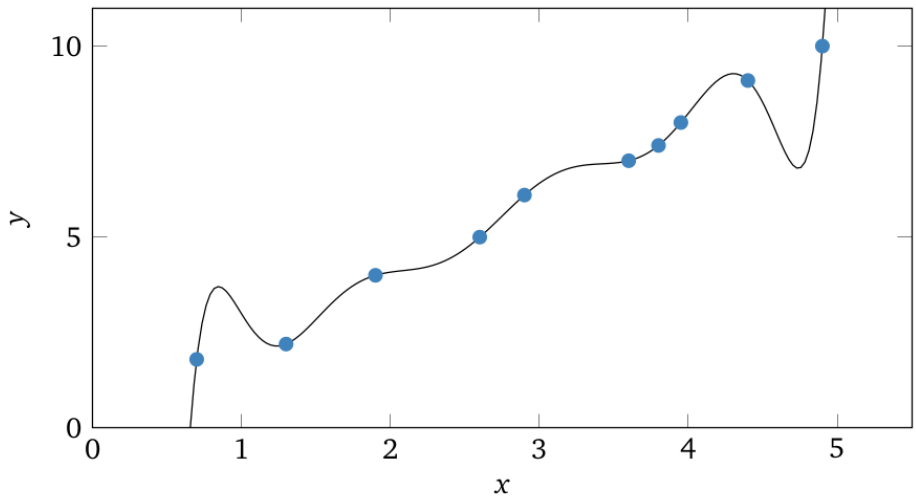
HOW REGULARIZATION HELPS

"Smaller weights are, in some sense, lower complexity, and so provide a simpler and more powerful explanation for the data, and should thus be preferred" - Let's unpack the story and examine it critically. To do that, let's suppose we have a simple data set for which we wish to build a model :

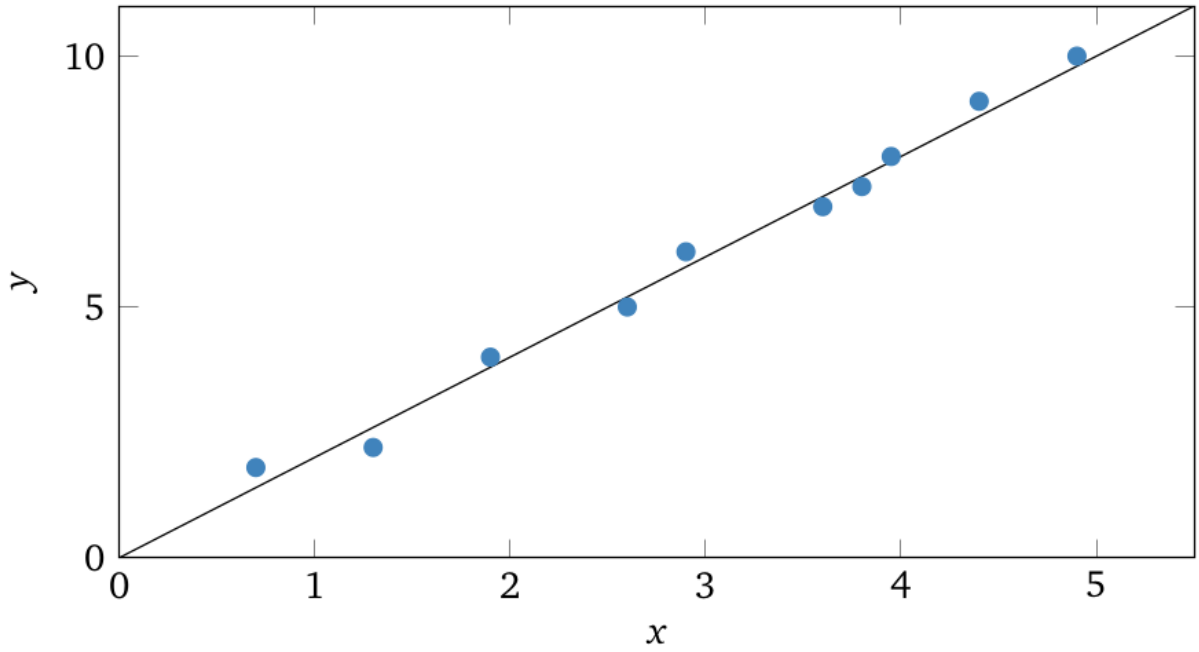


Now, let's go with a polynomial on understanding which , the neural network understanding will be quite facilitated.

$y = a_0x^9 + a_1x8 + \ldots + a_9$



Let’s consider another polynomial y=2x.



@ Which of these is the better model ?
@ Which is more likely to be true ?
@ Which model is more likely to generalize well to other examples of the same underlying real world phenomenon ?
These are so difficult to answer. It’s not a priori possible to say which of these two possibilities is correct. (Or, indeed, if some third possibility holds). Logically, either could be true. And it’s not a trivial difference. It’s true that on the data provided there’s only a small difference between the two models. But suppose we want to predict the value of y corresponding to some large value of x, much larger than any shown on the graph above.

If we try to do that there will be a dramatic difference between the predictions of the two models, as the 9th order polynomial model comes to be dominated by the x9 term, while the linear model remains, well, linear.
While the 9th order model works perfectly for these particular data points, the model will fail to generalize to other data points, and the noisy linear model will have greater predictive power.
The smallness of the weights means that the behaviour of the network won't change too much if we change a few random inputs here and there. That makes it difficult for a regularized network to learn the effects of local noise in the data. Think of it as a way of making it so single pieces of evidence don't matter too much to the output of the network. Instead, a regularized network learns to respond to types of evidence which are seen often across the training set. By contrast, a network with large weights may change its behaviour quite a bit in response to small changes in the input. And so an unregularized network can use large weights to learn a complex model that carries a lot of information about the noise in the training data.

In a nutshell, regularized networks are constrained to build relatively simple models based on patterns seen often in the training data, and are resistant to learning peculiarities of the noise in the training data. The hope is that this will force our networks to do real learning about the phenomenon at hand, and to generalize better from what they learn.

OTHER REGULARIZATIONS

L1 Regularization : In this approach we modify the unregularized cost function by adding the sum of the absolute values of the weights.

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|.$$

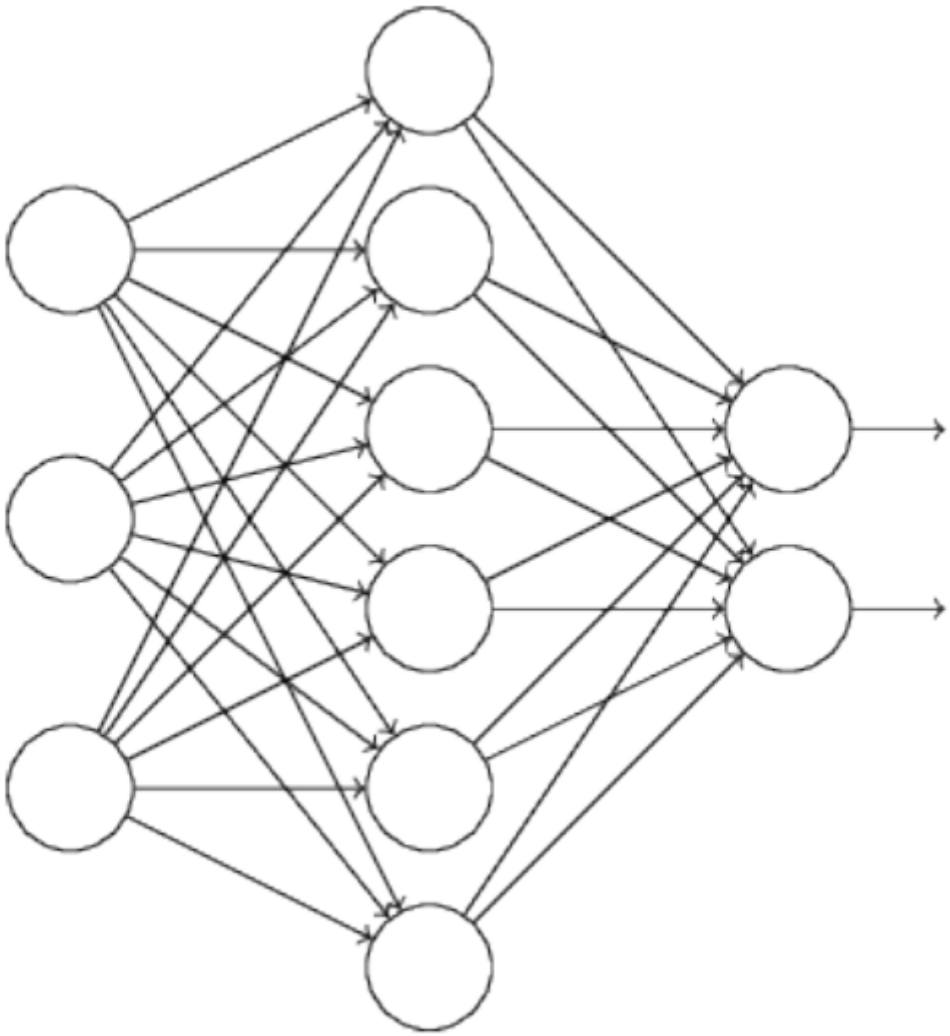
This is similar to L2 regularization, penalizing large weights, and tending to make the network prefer small weights. Of course, the behaviour of L1 Regularization varies from that of the L2 Regularization in penalizing the large weights.

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} \mathrm{sgn}(w),$$

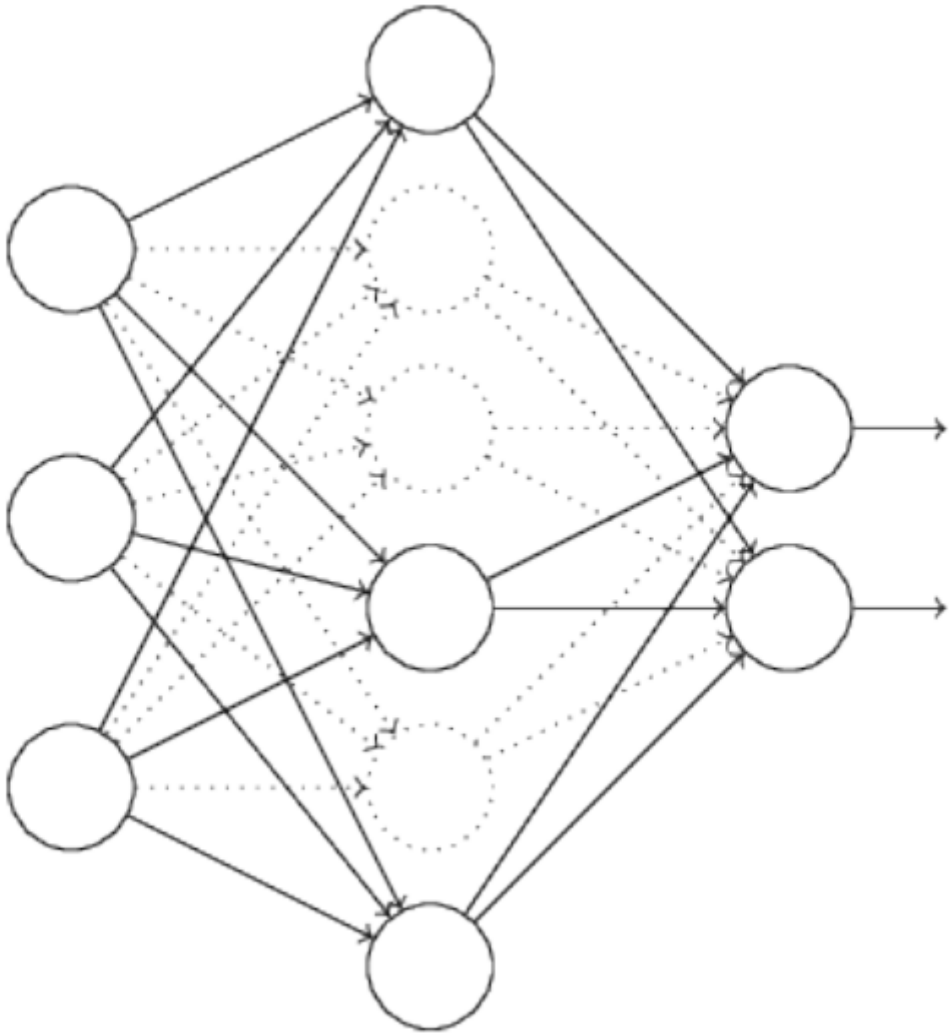
$$w \rightarrow w' = w\left(1 - \frac{\eta\lambda}{n}\right) - \eta \frac{\partial C_0}{\partial w}.$$

In L1 regularization, the weights shrink by a constant amount toward 0. In L2 regularization, the weights shrink by an amount which is proportional to w. And so when a particular weight has a large magnitude, |w|, L1 regularization shrinks the weight much less than L2 regularization does. By contrast, when |w| is small, L1 regularization shrinks the weight much more than L2 regularization.

The DROP-OUT :



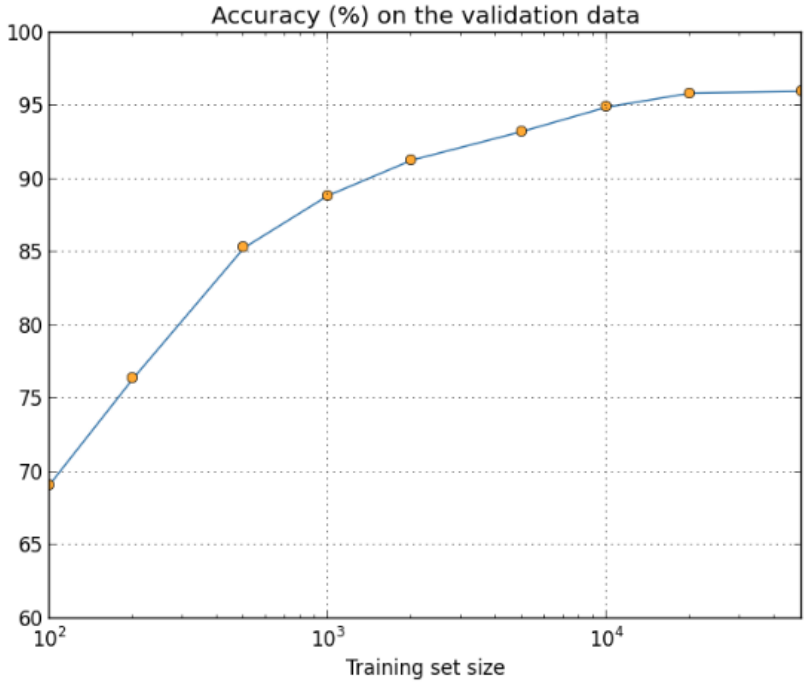
In particular, suppose we have a training input x and corresponding desired output y. Ordinarily, we'd train by forward-propagating x through the network, and then backpropagating to determine the contribution to the gradient. With dropout, this process is modified. We start by randomly (and temporarily) deleting half the hidden neurons in the network, while leaving the input and output neurons untouched. After doing this, we'll end up with a network along the following lines.



We forward-propagate the input x through the modified network, and then backpropagate the result, also through the modified network. After doing this over a mini-batch of examples, we update the appropriate weights and biases. We then repeat the process, first restoring the dropout neurons, then choosing a new random subset of hidden neurons to delete, estimating the gradient for a different mini-batch, and updating the weights and biases in the network.
By repeating this process over and over, our network will learn a set of weights and biases. Of course, those weights and biases will have been learnt under conditions in which half the hidden neurons were dropped out. When we actually run the full network that means that twice as many hidden neurons will be active. To compensate for that, we halve the weights outgoing from the hidden neurons.
Why would we expect it to help with regularization? To explain what's going on, I'd like you to briefly stop thinking Why would we expect it to help with regularization? To explain what's going on, I'd like you to briefly stop thinking about dropout, and instead imagine training neural networks in the standard way (no dropout).

In particular, imagine we train several different neural networks, all using the same training data. Of course, the networks may not start out identical, and as a result after training they may sometimes give different results. When that happens we could use some kind of averaging or voting scheme to decide which output to accept. For instance, if we have trained five networks, and three of them are classifying a digit as a “3”, then it probably really is a “3”. The other two networks are probably just making a mistake. This kind of averaging scheme is often found to be a powerful (though expensive) way of reducing overfitting. The reason is that the different networks may overfit in different ways, and averaging may help eliminate that kind of overfitting.

INCREASING TRAINING SET SIZE :



OTHER MODELS OF NEURON

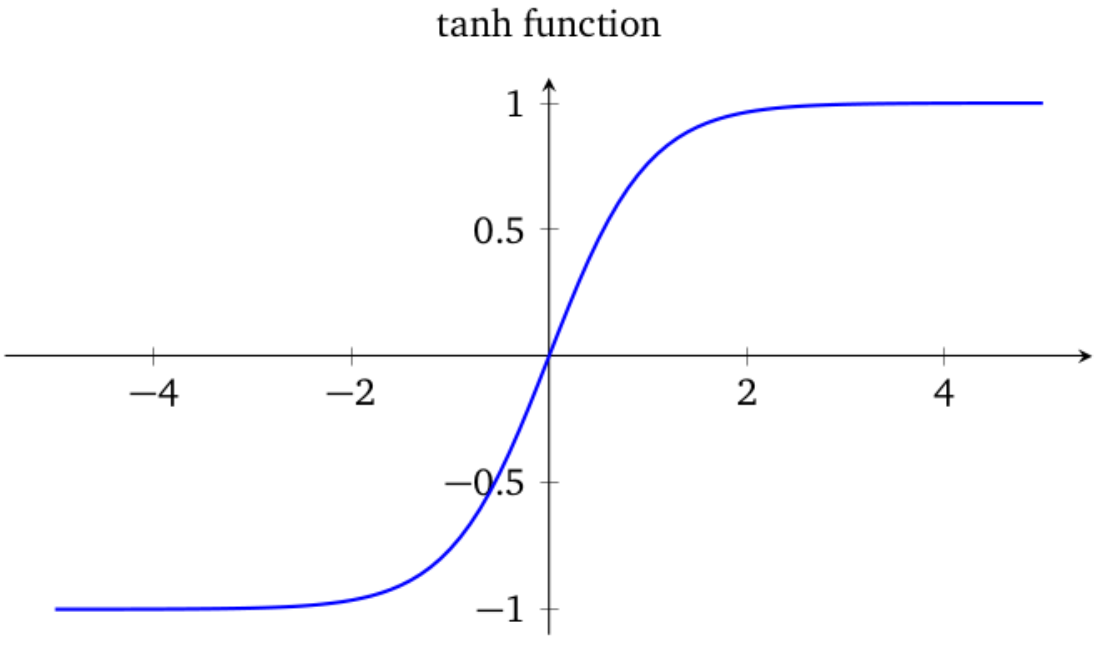
TANH : The simplest variation is the tanh neuron, which replaces the sigmoid function by the hyperbolic tangent function. The output of a tanh neuron with input x, weight vector w, and bias b is given by,

$$\tanh(w \cdot x + b),$$

The activation function can be written as,

$$\sigma(z) = \frac{1 + \tanh(z/2)}{2},$$

The tanh is just a rescaled version of the sigmoid function. We can also see graphically that the tanh function has the same shape as the sigmoid function,

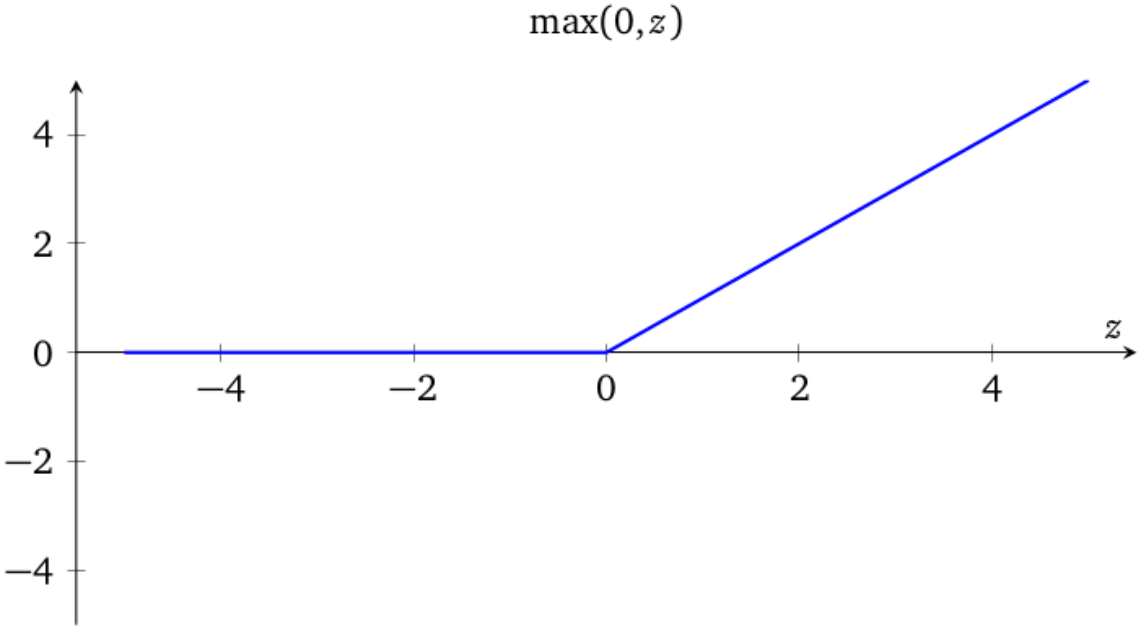


The tanh neurons range from −1 to 1.

RECTIFIED LINEAR NEURON : Another variation on the sigmoid neuron is the rectified linear neuron or rectified linear unit. The output of a rectified linear unit with input x, weight vector w, and bias b is given by,

$$\max(0, w \cdot x + b).$$

Graphically, the rectifying function max(0, z) looks like this :



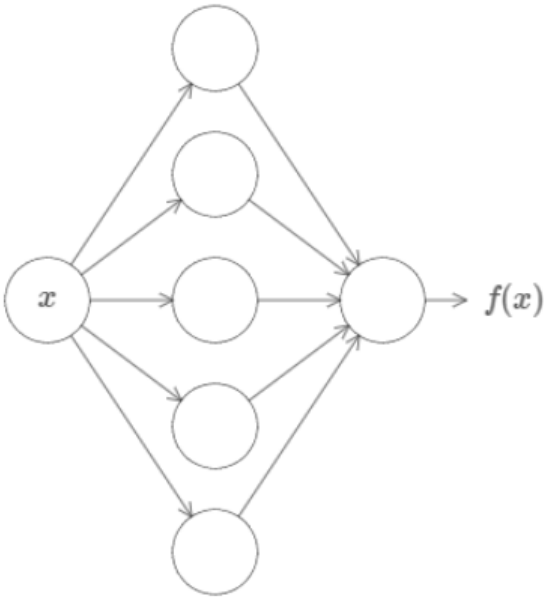
Obviously such neurons are quite different from both sigmoid and tanh neurons. However, like the sigmoid and tanh neurons, rectified linear units can be used to compute any function, and they can be trained using ideas such as backpropagation and stochastic gradient descent.

THE UNIVERSALITY

A Striking Fact is that the neural network can compute any function. The term used to describe this behaviour of the neural network is called , “The Universality”.

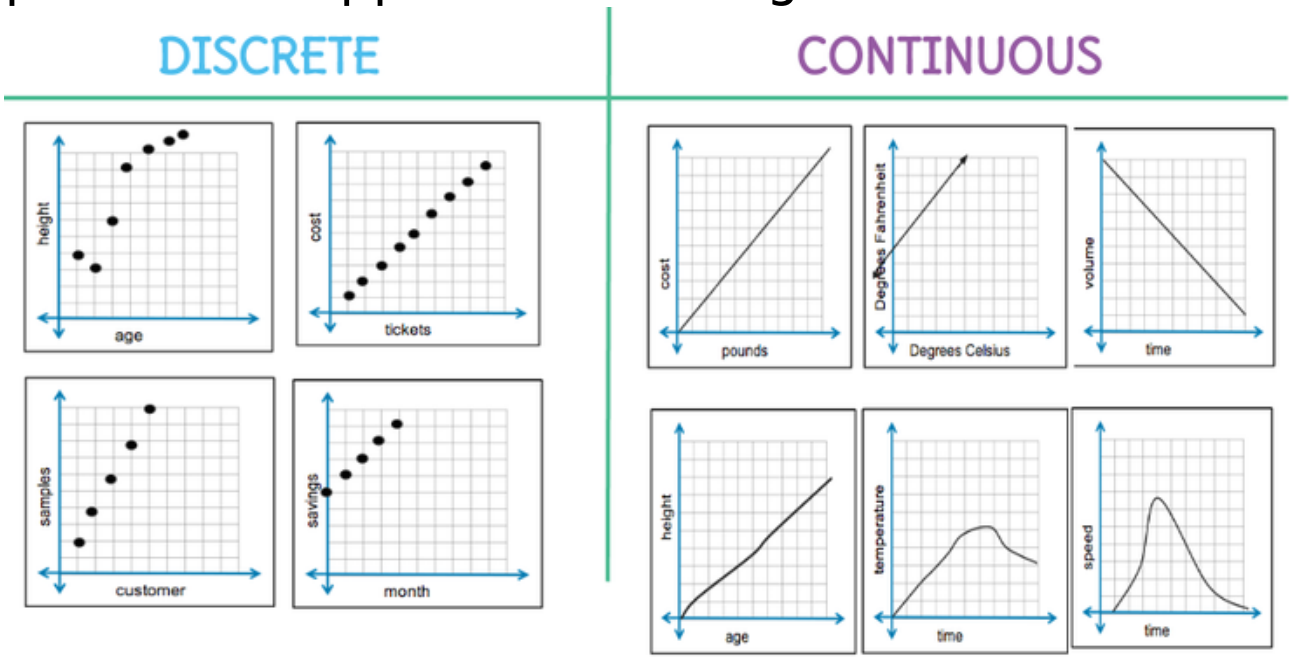
Two Caveats :

1. First, this doesn’t mean that a network can be used to exactly compute any function. Rather, we can get an approximation that is as good as we want. By increasing the number of hidden neurons we can improve the approximation. For an instance, if we have a network of three neurons in the hidden layer, For most functions only a low-quality approximation will be possible using three hidden neurons. By increasing the number of hidden neurons (say, to five) we can typically get a better approximation :

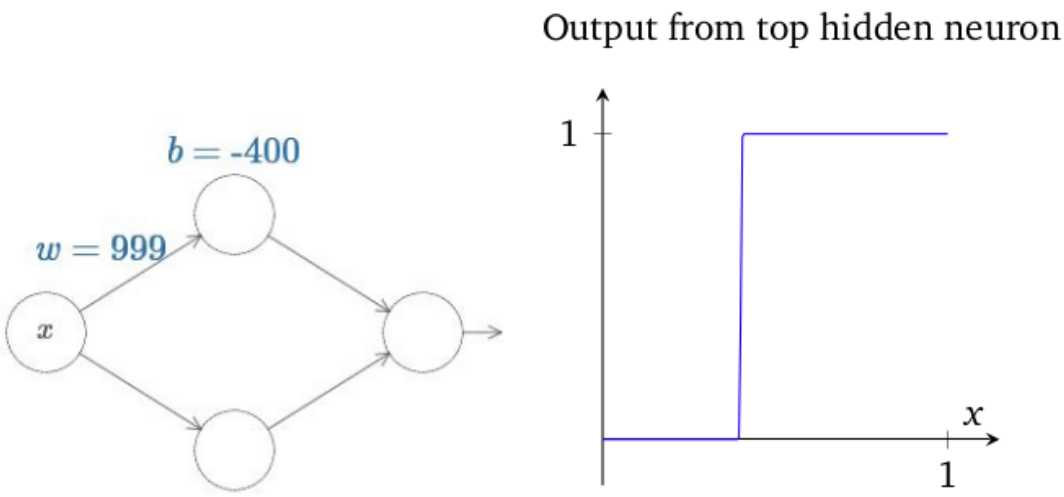


Two Caveats :

1. First, this doesn't mean that a network can2. The second caveat is that the class of functions which can be approximated in the way described are the continuous functions. If a function is discontinuous, i.e., makes sudden, sharp jumps, then it won't in general be possible to approximate using a neural net.

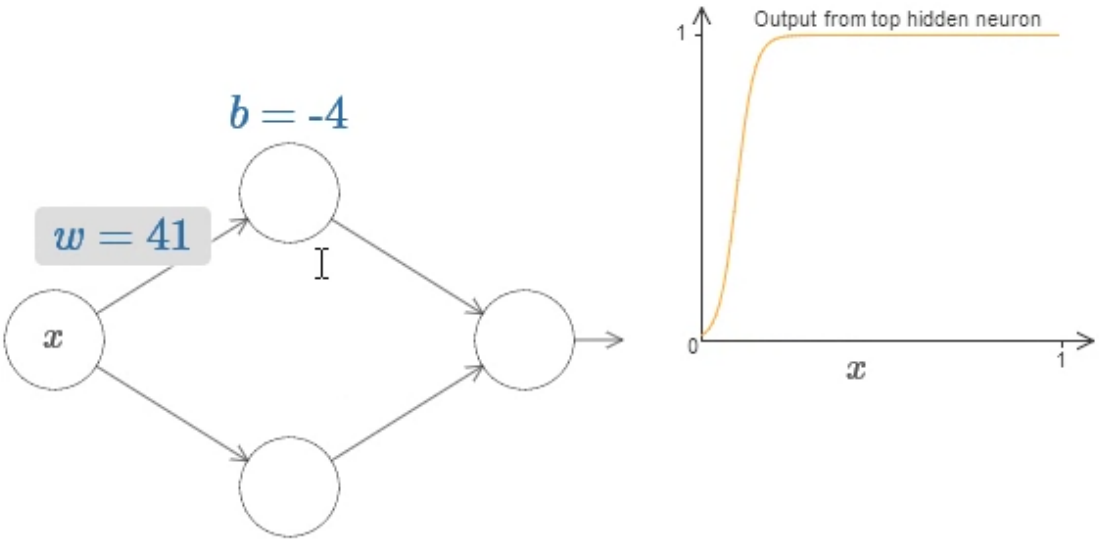


With One Input and One Output :

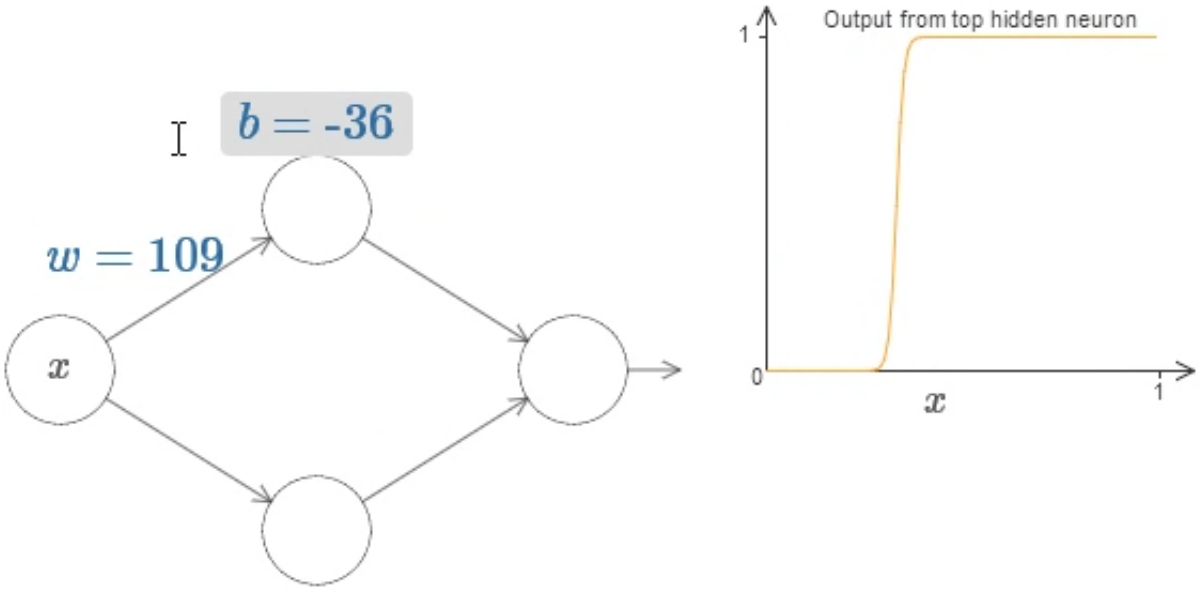


At what value of x does the step occur? Put another way, how does the position of the step depend upon the weight and bias? The position of the step is proportional to b, and inversely proportional to w.

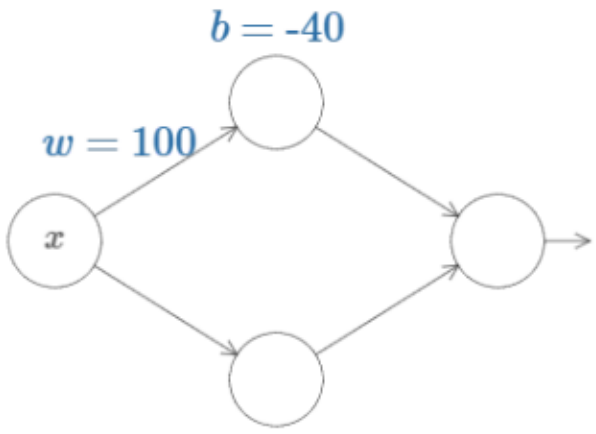
Altering Weight :

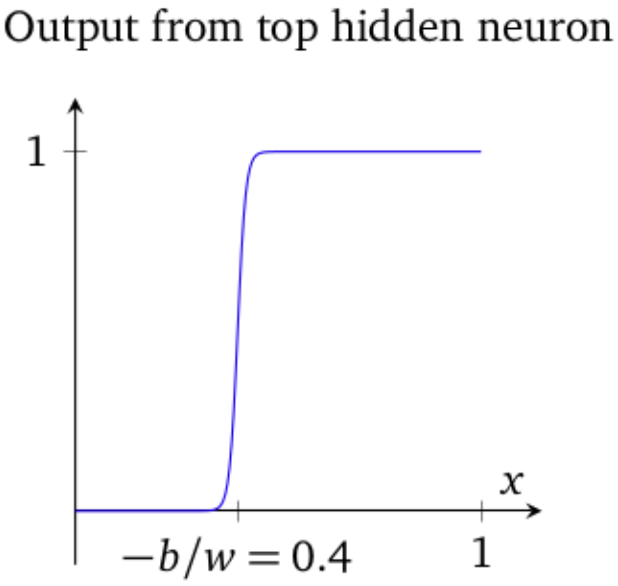


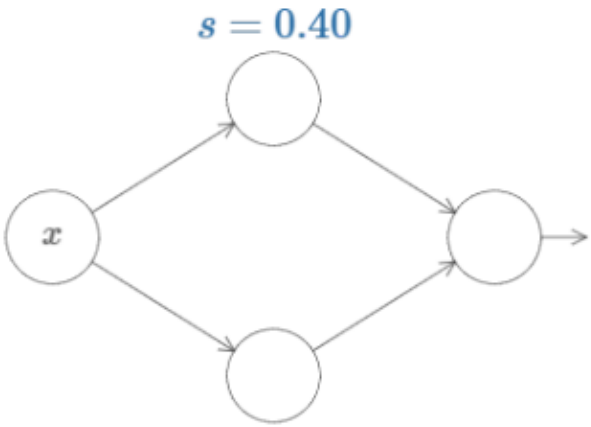
Altering Bias :

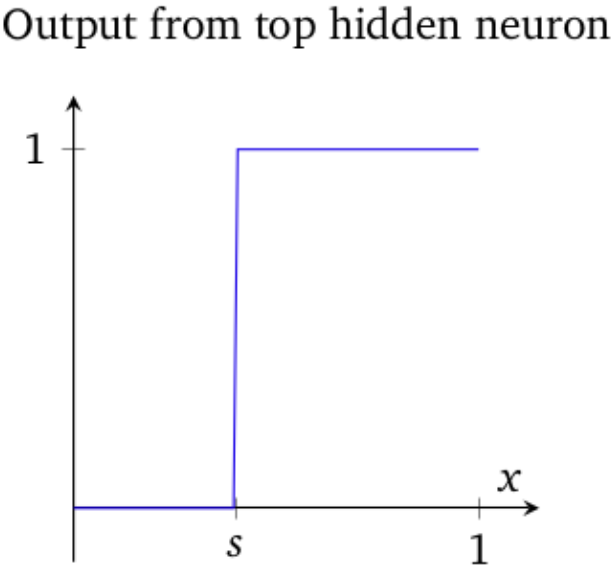


The step is at the position -b/w usually called, the step position.

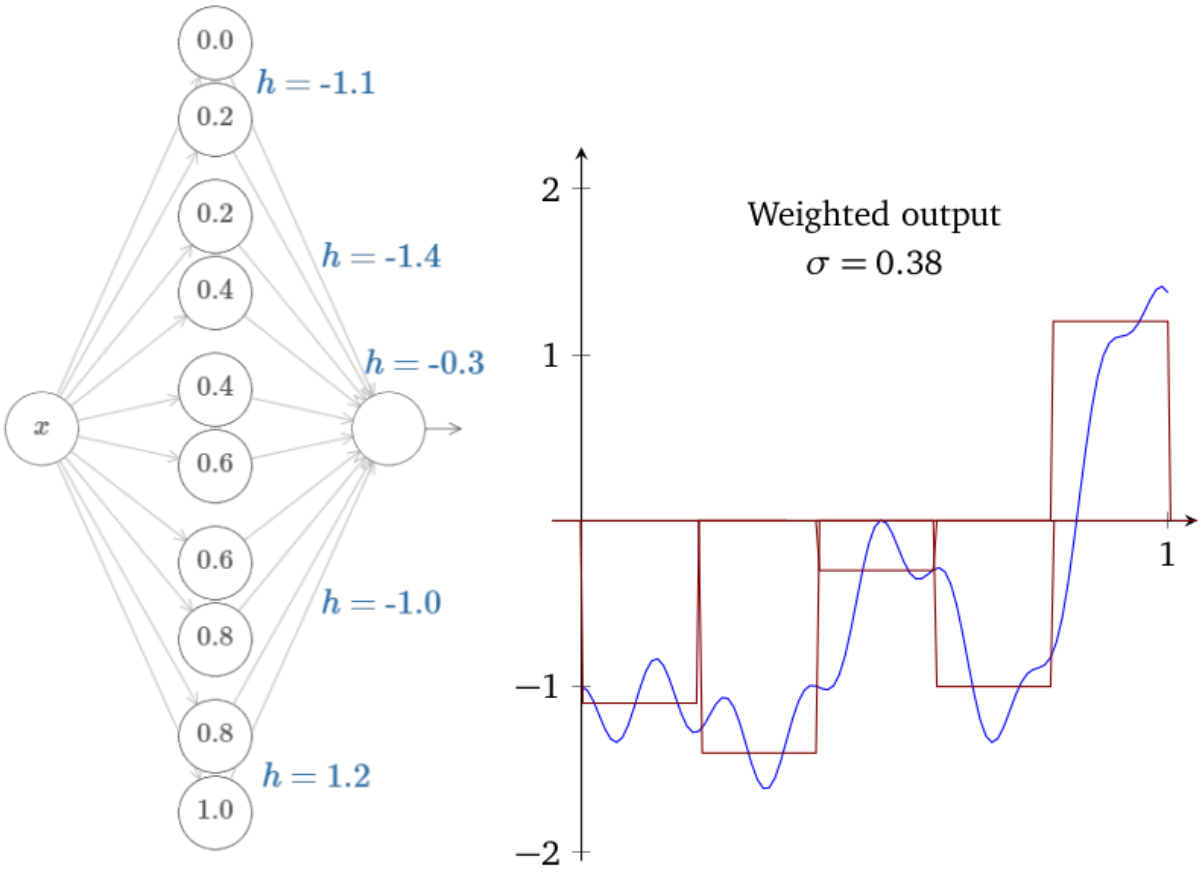








You should find it fairly easy to get a good match to the goal function.

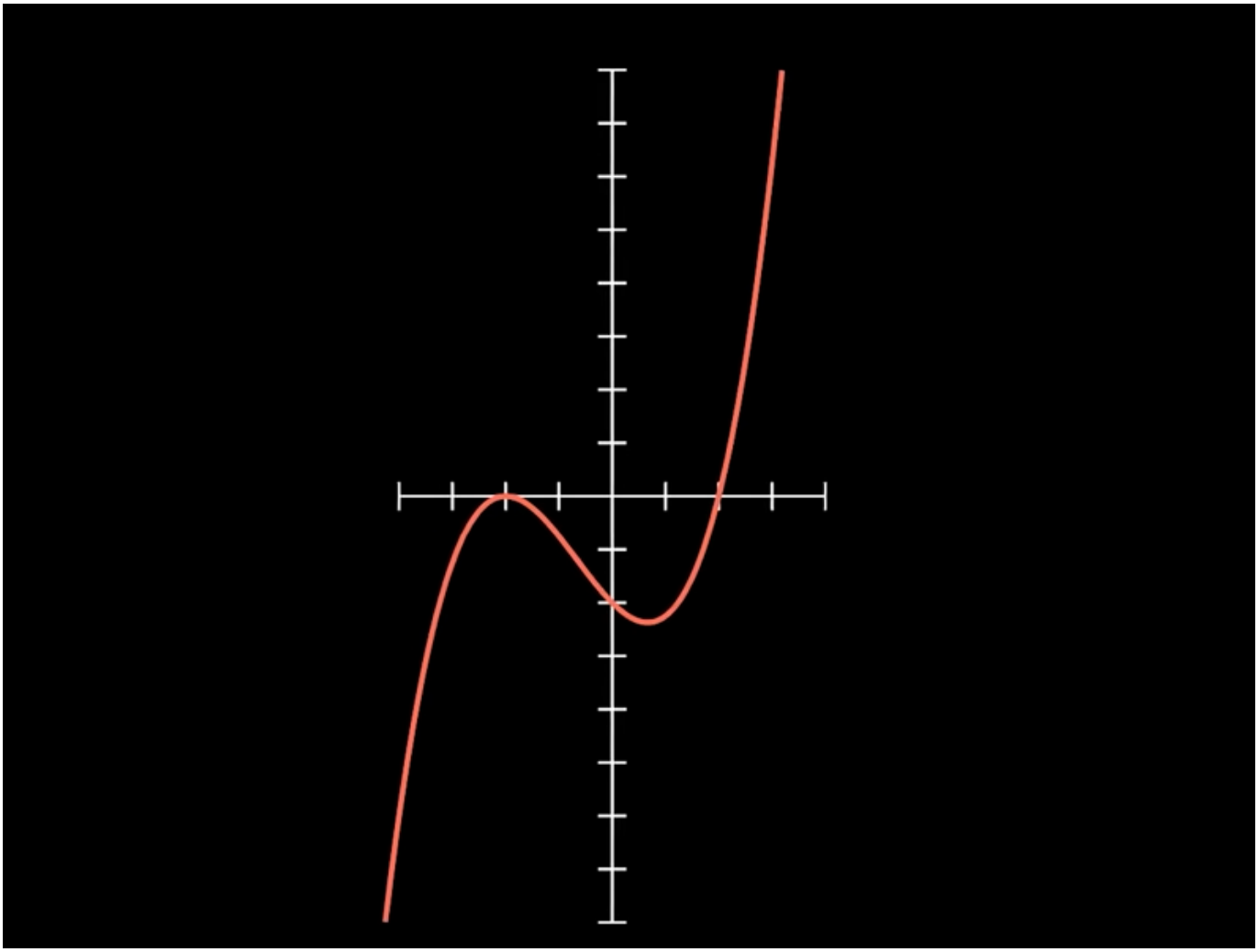


The Function Approximator :

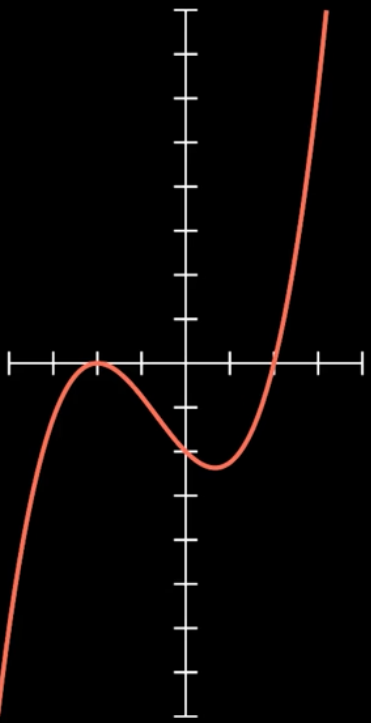
Any sort of function can be simulated once the enough inputs and the corresponding outputs are provided. What we need is a function approximator , the so-called Neural Network.

$$f(x) \approx T(x)$$

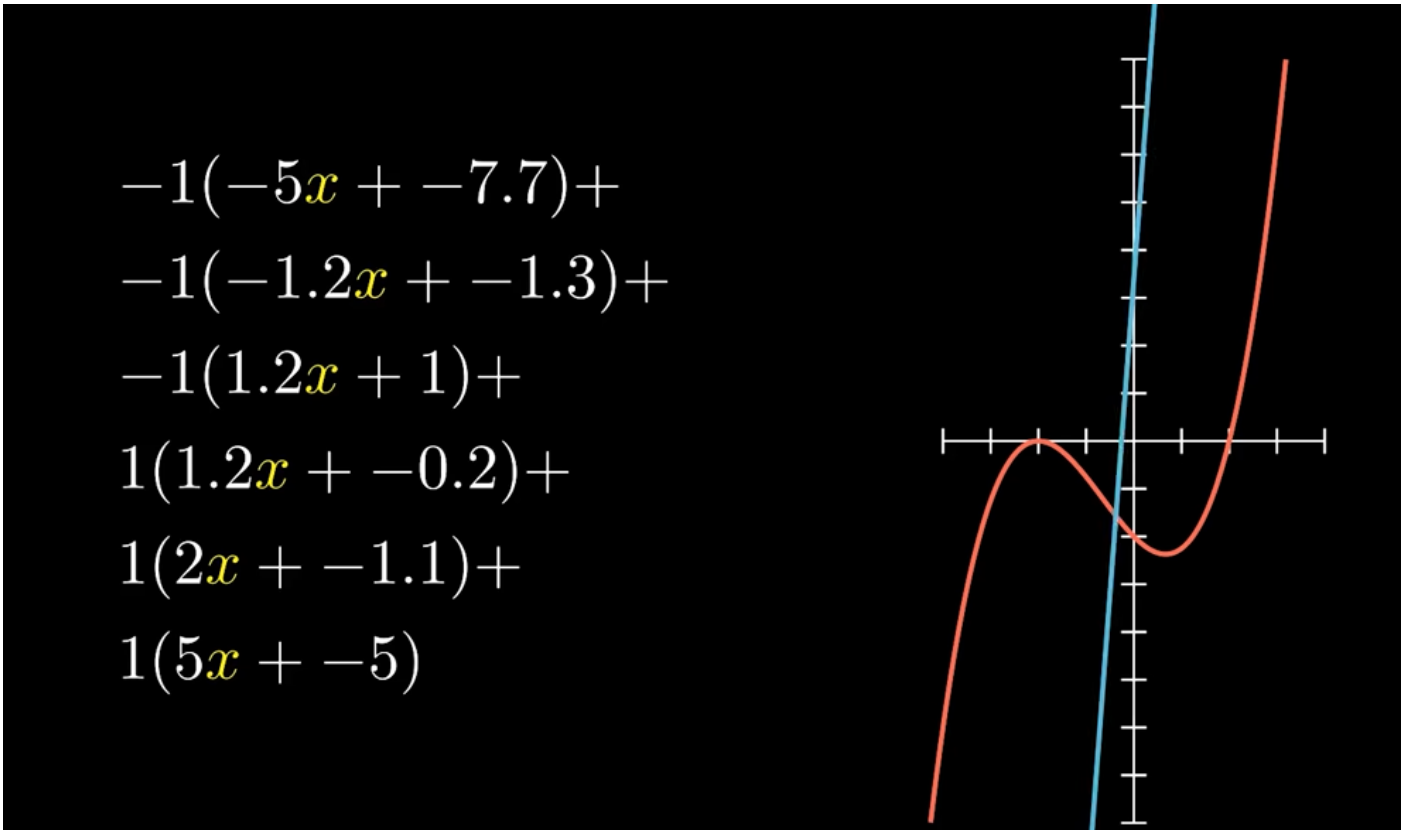
Let’s consider a target function as shown below :



$$N_0(x) = -5x + -7.7$$
$$N_1(x) = -1.2x + -1.3$$
$$N_2(x) = 1.2x + 1$$
$$N_3(x) = 1.2x + -0.2$$
$$N_4(x) = 2x + -1.1$$
$$N_5(x) = 5x + -5$$

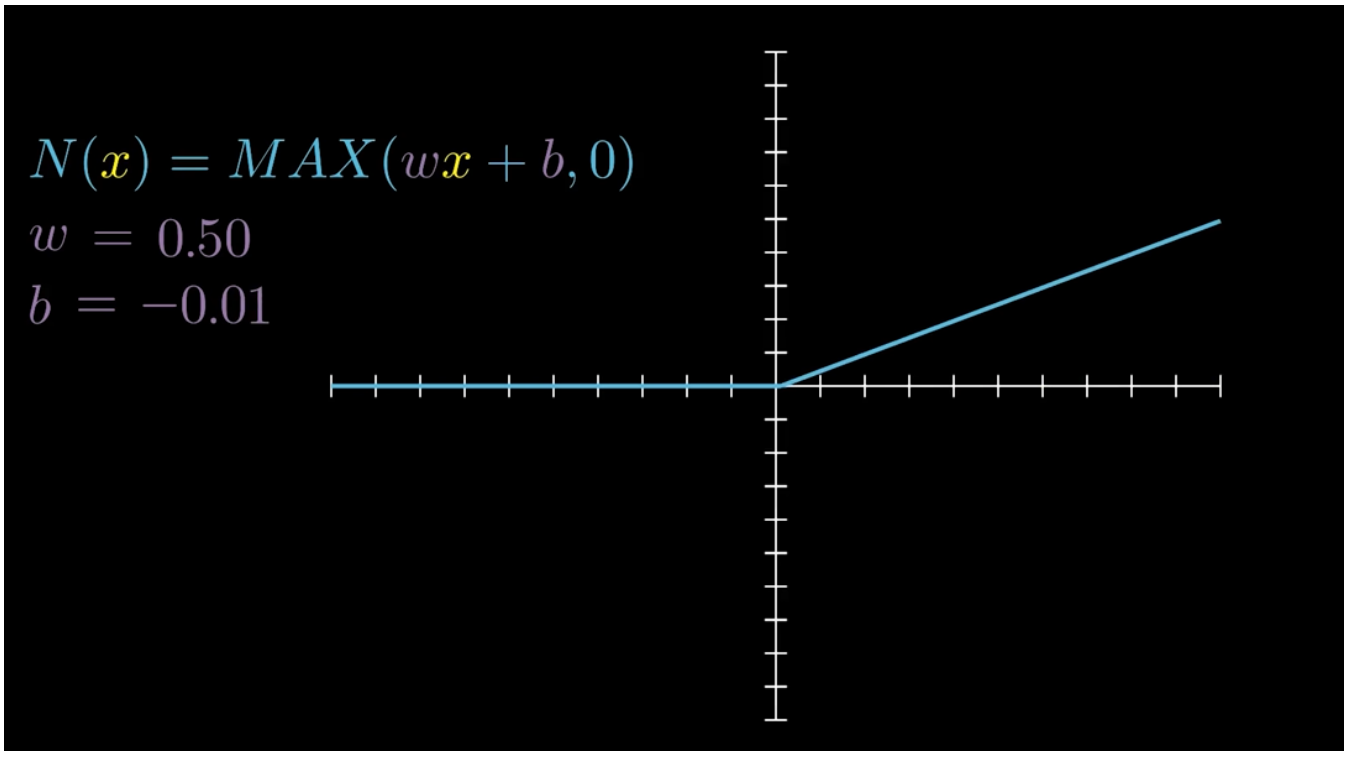


What could be the function found for the manually given parameters ?

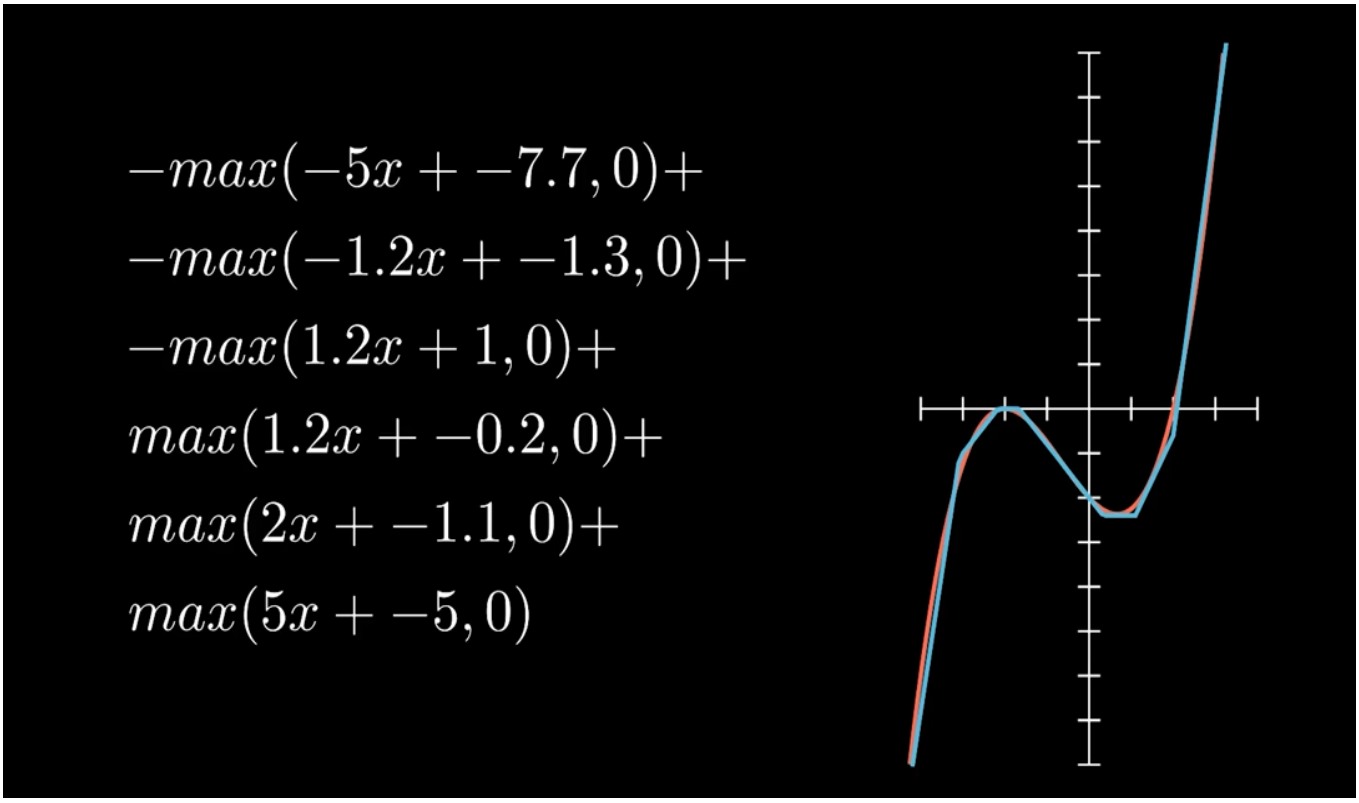


It is not even close! But, we can make it so...

To make it better, what about using the ReLU activation function ?



Now, with the ReLU activation function used, we could get a way-better approximation.
Now, with the ReLU activation function used, we could get a way-better approximation.



Importantly :

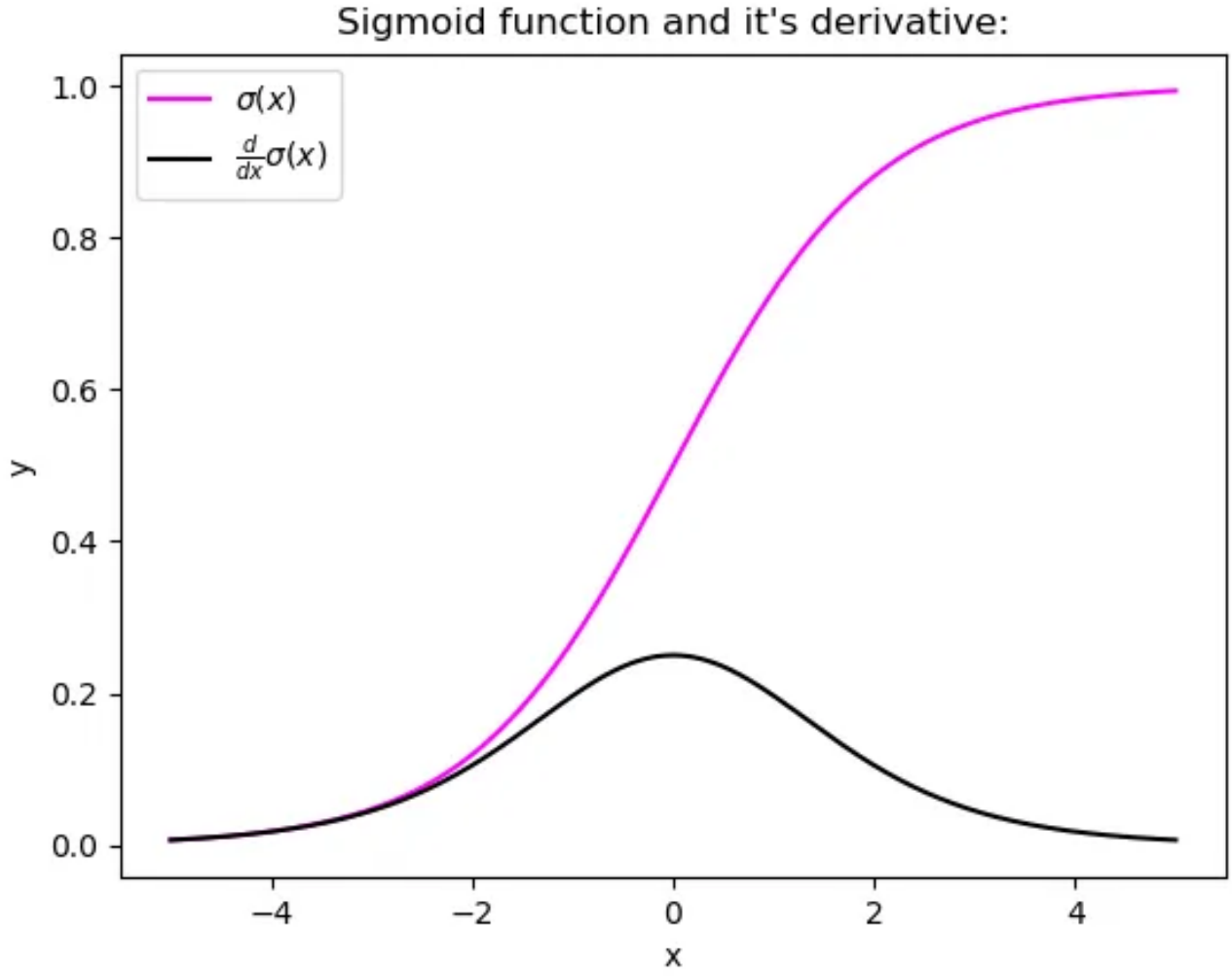
If you don’t have enough data, your approximation will be all wrong. It doesn’t matter how many neurons you have, how sophisticated your network is when you don’t have sufficient data. So, DATA IS IMPORTANT.

THE PROBLEMS :

Vanishing Gradient Problem : It is the problem that causes a major difficulty while training a model. It involves the weights in the earlier layer of the network. The gradient in the early layers of the network becomes really small. i.e., It becomes vanishingly small.

The gradient is usually calculated with respect to a particular weight. The calculated gradient is used to update that weight. The weight gets updated in someway that is proportional to the gradient. The updated value of weight will be really really very very small as the gradient is vanishingly small resulting the network being unable to approximate.

This is the so-called , Vanishing Gradient Problem. Gradient Descent requires calculating the derivative of the cost function w.r.t weights and biases. In order to do that we must apply the chain rule, because the derivative we need to calculate is a composition of two functions. As dictated by the chain rule we must calculate the derivative of the sigmoid function. One of the reasons that the sigmoid function is popular activation function with neural networks, is because its derivative is easy to compute.

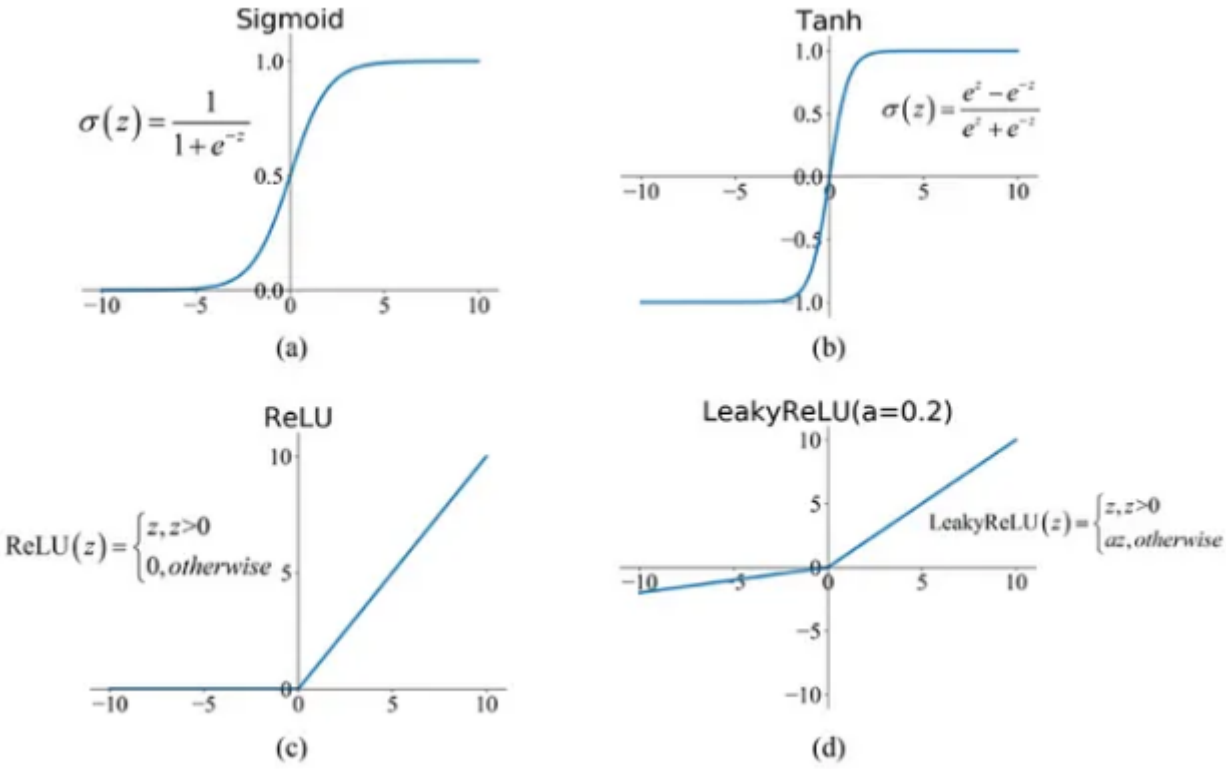


But if you look at the figure above, you will notice the derivative of sigmoid function saturates at the ends of the function (i.e. when the input is large, positive or negative), the derivative values are close to 0. And at each step when propagating the error back to the network, the previous layer multiplies with a smaller derivative value, thus derivative keeps getting closer to 0. This is why the gradients start to vanish closer to the input layers.

TO SOLVE :

1. Take care on weight initialization | Xavier initialization is an attempt to do exactly that. The weights of the network are selected for certain intermediate values, such a way that the variance is not huge when we pass through different layers of the network. This initialization process is known as Xavier initialization.

2. Sigmoid function is popular activation function because its derivative is easy to compute, but we saw that it has problems of saturation when the inputs are too large or small. We have better activation functions. Rectified linear units (ReLU) activation function do not saturate for positive values.



Different types of activation functions

This activation only saturates on one direction and thus are more resilient to the vanishing of gradients. ReLU activation function is not perfect. It suffers from a problem known as the dying ReLUs: during training, some neurons effectively die, meaning they stop outputting anything other than 0. Leaky ReLU helps overcome this problem.

3. **Residual Networks :** It introduces bypass connections that connect layers further behind the preceding layer to a given layer. This allows gradients to propagate faster to deep layers before they can be attenuated to small or zero values.

Exploding Gradient Problem : If the gradient becomes large, the model becomes unstable and model unable to learn from the training data. Exploding gradient problem can be identified if there is large changes in the loss from update to update.

If the model loss goes to NaN during the training is an indication of exploding gradient problem.Exploding gradient problem is due to large weights.

Exploding Gradient Problem can be address by :

1. Using the Gradient Clipping (Threshold)
2. Using the Weight Regularization
3. Re-design the model with fewer layers

Gradient Clipping :

The ***error derivative is changed or clipped to a threshold*** during backward propagation.

By rescaling the error derivative, the updates to the weights will also be rescaled, dramatically decreasing the likelihood of an overflow or underflow.

VG Problem :

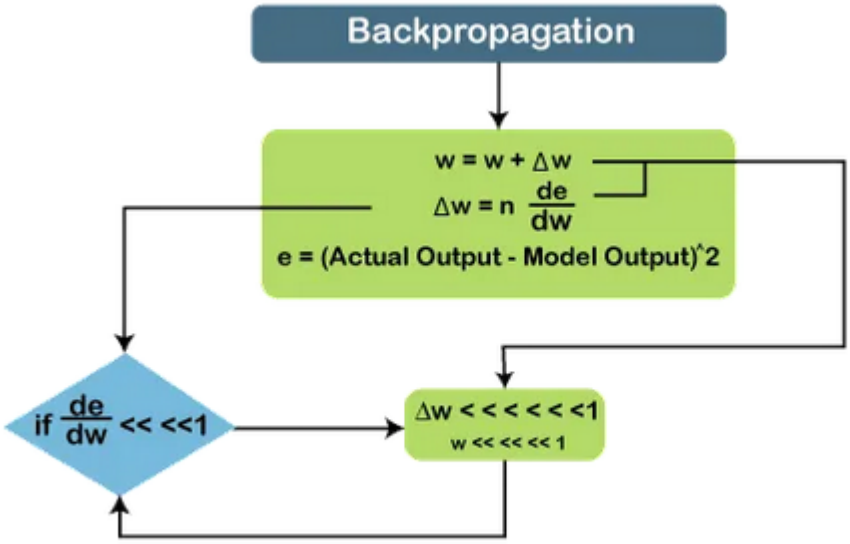


Image source : javepoint

EG Problem :

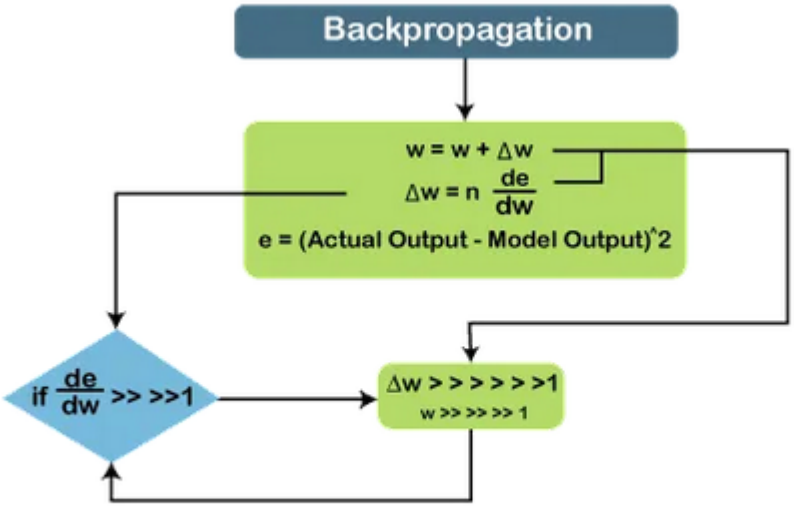


Image source : javapoint

SOME OTHER NEURAL NETWORKS :

CNN – Convolutional Neural Network :

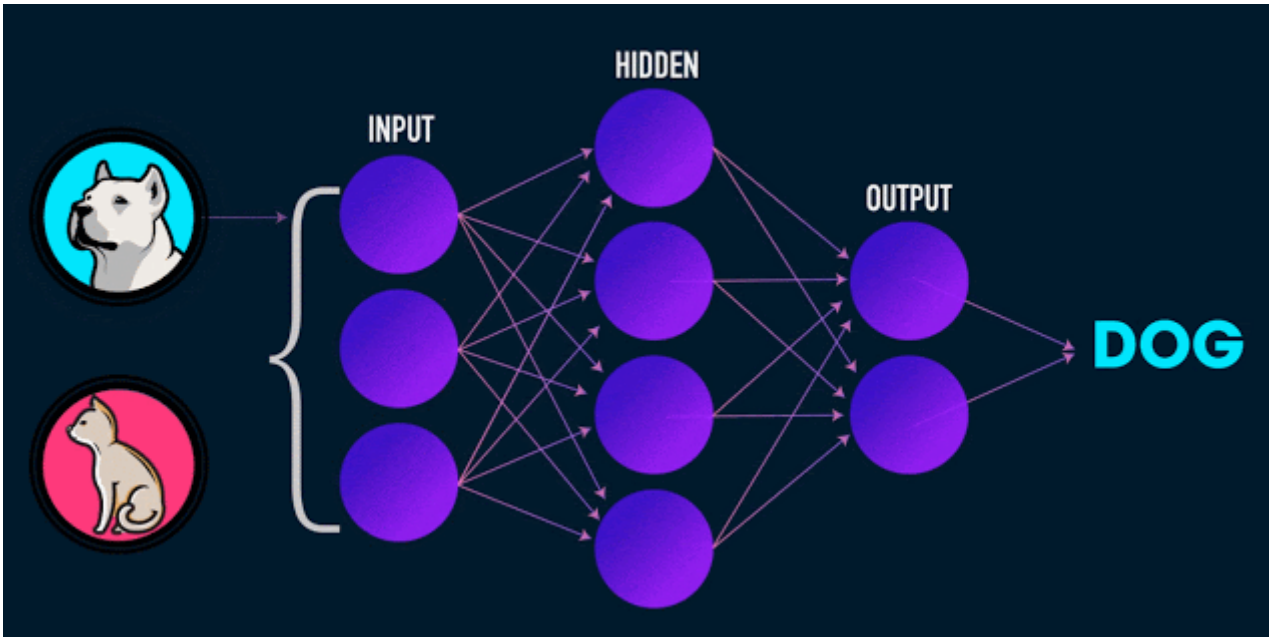
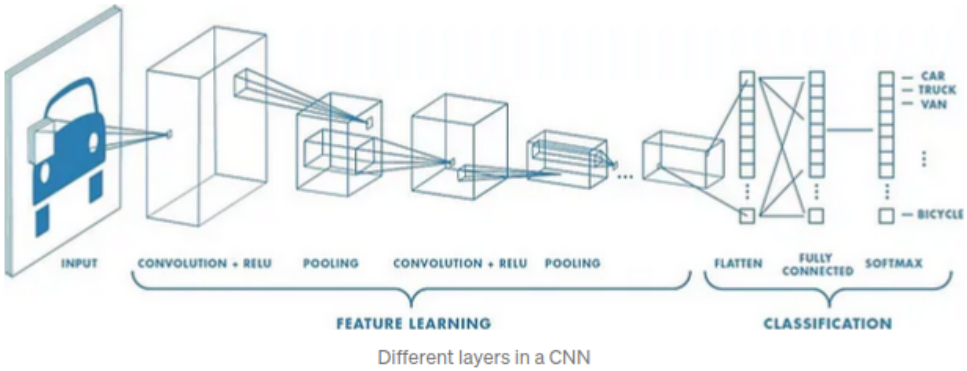


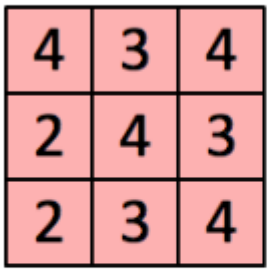
Image classification is the task of taking an input image and outputting a class or a probability of classes that best describes the image. In CNN, we take an image as an input, assign importance to its various aspects/features in the image and be able to differentiate one from another. The pre-processing required in CNN is much lesser as compared to other classification algorithms.



Different layers in a CNN

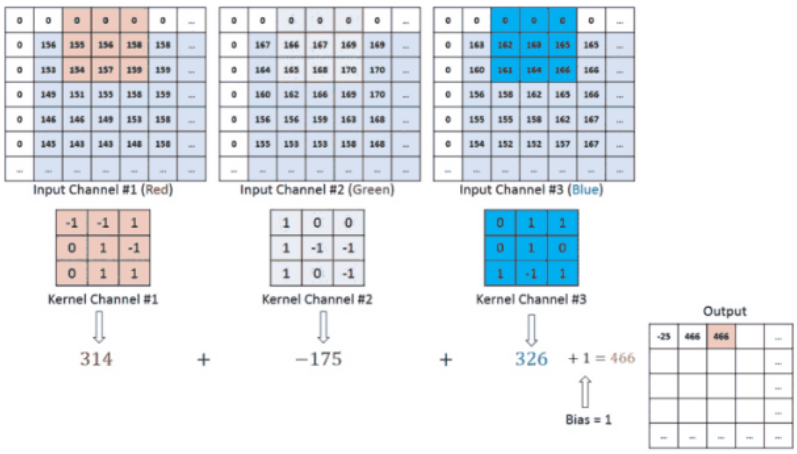
A CNN typically has three layers: a convolutional layer, pooling layer, and fully connected layer.
The main objective of convolution is to extract features such as edges, colours, corners from the input. As we go deeper inside the network, the network starts identifying more complex features such as shapes,digits, face parts as well. This layer performs a dot product between two matrices, where one matrix (known as filter/kernel) is the set of learnable parameters, and the other matrix is the restricted portion of the image.

Image



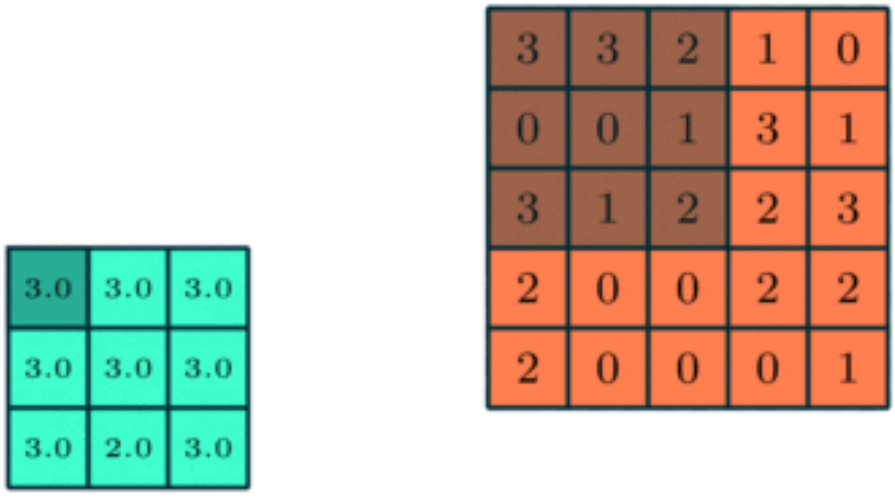
Convolved Feature

For RGB images, the convolving part can be visualized as follows :



Convolution operation on an MxNx3 image matrix with a 3×3×3 Kernel

The Pooling layer is solely to decrease the computational power required to process the data. It is done by decreasing the dimensions of the featured matrix even more. In this layer, we try to extract the dominant features from a restricted amount of neighborhood. Let us make it clear by taking an example.



The orange matrix is our featured matrix, the brown one is a pooling kernel and we get our blue matrix as output after pooling is done. So, here what we are doing is taking the maximum amongst all the numbers which are in the pooling region and shifting the pooling region each time to process another neighborhood of the matrix.

There are two types of pooling techniques: *AVERAGE-pooling* and *MAX- pooling.* The difference between these two is, in *AVERAGE-pooling*, we take the average of all the values of pooling region and in *MAX-pooling,* we just take the maximum amongst all the values lying inside the pooling region. So, after pooling layer, we have a matrix containing main features of the image and this matrix has even lesser dimensions, which will help a lot in the next step. The final fully connected layer does the classification process.

TOPICS COVERED

Contents

THANK YOU MICHAEL NEILSEN ❤