1.0 Main concepts

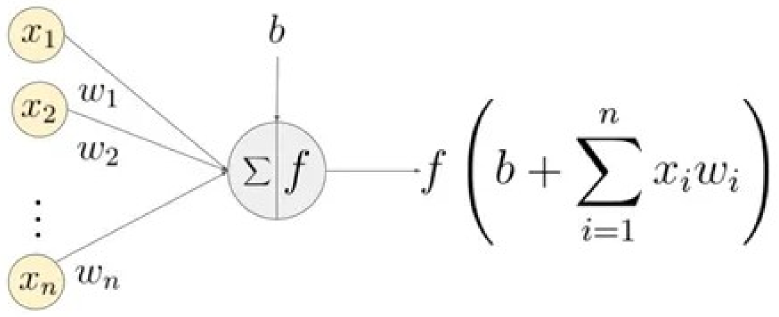
Summary

Why artificial neural network works?

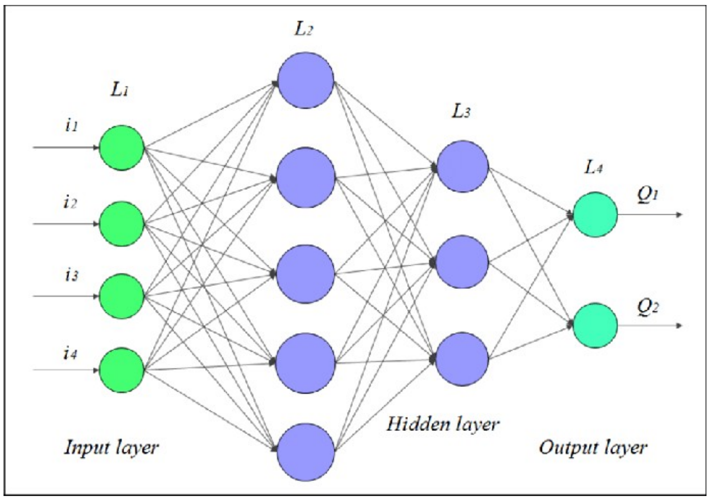
This neural networks represent an approximation of a real multi-variable function, and is mathematically proved that this neural networks can approach the behavior of a hidden real multi-variable function, which means this neural networks can predict the “shape” that this real function would output depending on the value of the variables

Artificial neural network structure

First let’s explain the main piece of this puzzle: the artificial neuron. We represent this neurons as a node with an **input** connected with many edges and one **output** connected with one edge, accompanied with an error value called **bias** and a previous special function which determines the output of a neuron, called **activation function**. Both input and output layers have assigned an special value called **weight** for each edge. Bias and weight values represent the magnitude of the pulse of a neural network, like biological neurons. Activation function determines wether the output value of this neuron is higher or lower (for example, a function called Sigmoid output decimal values between 0 and 1)



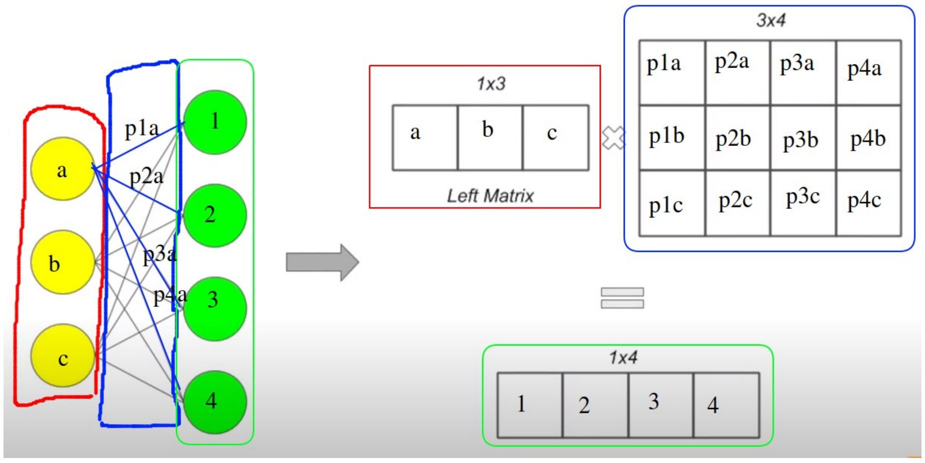
With this neurons we can build a neural network. There are many different artificial neural networks, but the most common are feed-forward networks. They have three important layers, an input layer whose nodes only take the raw input (they are not neurons!), the hidden layer, which has many independent layers, with multiple neurons each one, and an output layer, which has many neurons. Here is an example:



This networks receive the input data from the left (green layer), and propagate the prediction along the network (purple layer), from left to right, finally reaching the output layer (blue? layer). Each example passes through the entire network once, making a different prediction depending on the input received.

It's important to know that all neurons in the same layer usually use the same activation function, but this functions can be different between layers. Also, usually the activation function of the output layer is different of the activation function of the hidden layers.

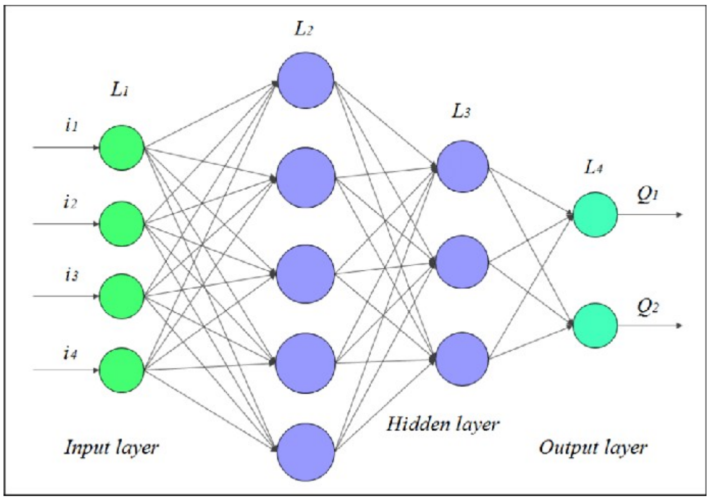
In order to manage this bunch of nodes and edges, matrices are required. A possible way of representing this neural networks as matrices is:



We can represent layer nodes as a matrix of 1 row and the number of nodes columns. Weights are represented ad a matrix with number_nodes_previous_layer rows and number_nodes_current_layer cols.

Also, is necessary to create another matrix with 1 row and the number of nodes columns, so we can store the bias of each neuron.

Remember the previous neural network image shown, we can represent this network like a structure that contains all the layer values (with [al] and without [zl] activation function applied), the bias values for each layer and the weight values for each edge:



```
Network = [
        Layers_values_zl,
        Layers_values_al,
        Layers_bias,
        Layers_weights
]

Layers_values_zl = [
        [ matrix 1x5 ],
        [ matrix 1x3 ],
        [ matrix 1x2 ]
]
```

```
Layers_values_al = [
        [ matrix 1x5 ],
        [ matrix 1x3 ],
        [ matrix 1x2 ]
]

Layers_bias = [
        [ matrix 1x5 ],
        [ matrix 1x3 ],
        [ matrix 1x2 ]
]

Layers_weights = [
        [ matrix 4x5 ],
        [ matrix 5x3 ],
        [ matrix 3x2 ]
]
```

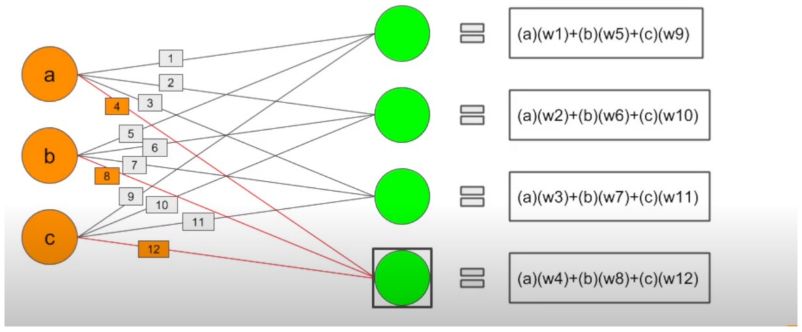
Note we don’t store the bias for input layer, remember its nodes are not neurons

Values on each matrix in Layers_values doesn’t matter, but **all values on each matrix in Layer_bias are usually initialized to 0, and all values on each matrix in Layers_weights are usually initialized to a random value**. Depending on this random values, our network would learn better or worst.

There are many ways to initialize those values, some of them can be found here:
https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/

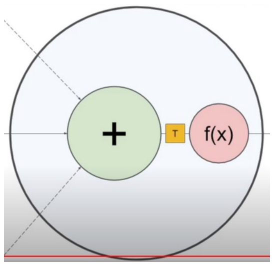
This representation will make things easier to manage as we are going to see now.

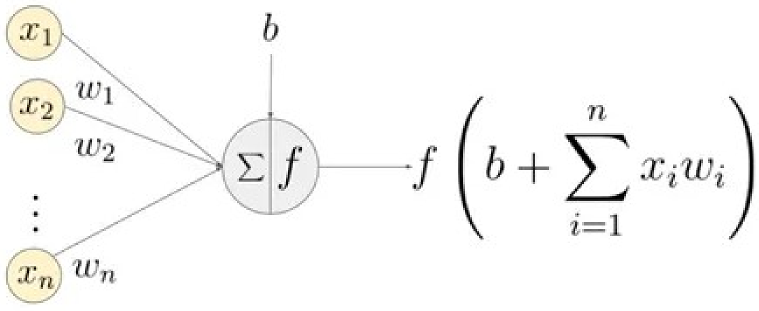
How neural network predicts: forward propagation



A neural network first get the values introduced at input layer. Next computes for each neuron in the next layer the sum of product of each previous neuron multiplied by the weight of the weight which connects both nodes. For example, the surrounded green neuron computes the sum of values a, b c, each one multiplied respectively by the weights of 4, 8 and 12 edges

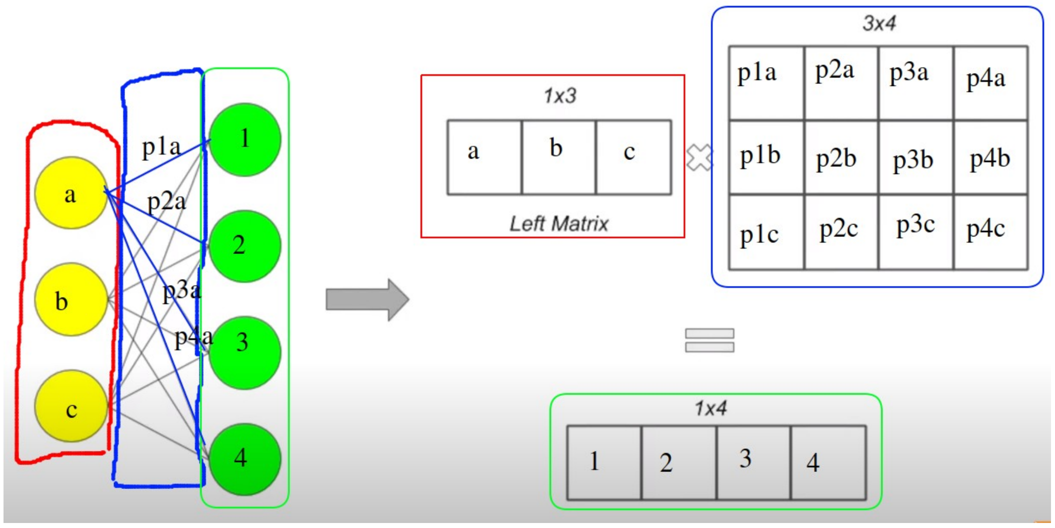
Next, for each neuron in current layer, we sum each corresponding bias value. In next image, we can see the sum (green circle) we computed before, and T orange square, which represents the value of the sum plus the bias of this neuron, the b variable of the other image





Finally, we pass this value (T) through the activation function of this neuron, and obtain its output value.

We can use matrices to automatize this process. Simply we need to calculate the matrix product of the matrix of the previous layer by the matrix weights of the current layer. Next, sum the result matrix with the bias matrix of current layer, and finally apply the activation function for each value.



A possible pseudo-code would be (input_list is a list with input values):

```
Network[ Layers_values_zl ].append( matrix_product(matrix_1_row(input_list), Network[ Layers_weights][0]) )

Network[ Layers_values_al ].append( activ_function_each_element(Network[ Layers_values_zl ][0]) )

for L in range(number_layers-1):

        Network[ Layers_values_zl ].append( matrix_product(Network[ Layers_values_al ][L], Network[ Layers_weights][L+1]) )

        Network[ Layers_values_al ].append( activ_function_each_element(Network[ Layers_values_zl ][L+1]) )
```

We will repeat this operation between every pair of layers in neural network, from left to right, until reaching the output layer. We will append al_bias in

How neural network learns: back propagation

When initializing a neural network, remember we set all bias values to 0, and all weights a random value. Obviously, this network isn't going to make good predictions

So, how do we "teach" our neural network how to predict better? The short answer is **changing the bias and weight values.**

Here comes a big problem. Imagine one network consist of thousands of layers, and each layer has thousands of neurons. How are we going to adjust all this values? Luckily we can use an algorithm called back propagation, which takes many examples with a set of its input values and their expected values.

We first forward this input values through the entire network as seen before. Then we move to the output layer and review how foolish our neural network is (basically, how bad it predicted the expected values).

Next, we compute a value that indicates how bad each neuron at output layer predicted its expected value, with something called an **error/cost function**

Then we compute for each weight and bias from the network a number it need to be subtracted in order to fit better the neural network to this training example. We store all this values in a vector called **gradient vector**. We need to **back propagate** this error from the output layer until reaching the input layer, in order to calculate all this bias and weight values.

Finally we subtract all the bias and weights by their corresponding element in this gradient vector.

This process needs to be repeated for each example in our training set, and also we need to pass all this examples several times until the network start making better predictions

Theoretically we need to calculate this gradient vector for each example and subtract the network by this vector, but in practice, we store this gradients for a small set of training examples and average this values for each weight and bias

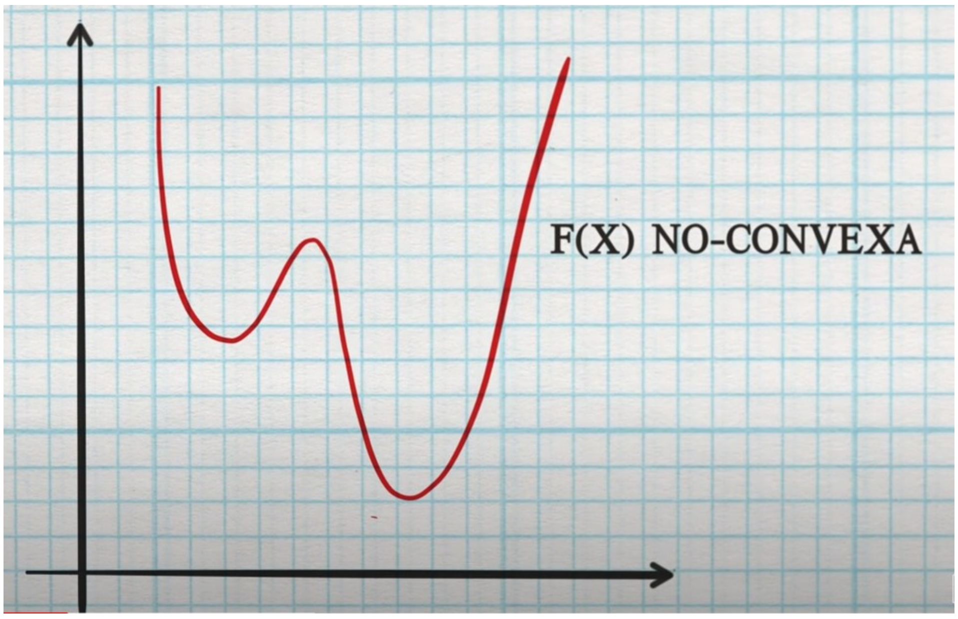
We first are going to see why we need to calculate this cost function value.

Later we will see what a gradient vector is and why we need to subtract its values to the weights and biases of a neural network

Cost/error function

Imagine a function with one variable which could represent how good or bad does our network predict values for every input values. Its values would become higher the worst they predict, or tend to 0 the better they predict. This function is called the **cost/error function**

In this scenario, our goal would be to reach where does our cost function has the lowest error value. This task would be easy if our function shape were like a parable, but in real scenarios, this function have multiple points where we can find a local minimum, which means, a place in the function where its first derivative is equals to 0. That means, going another step forward or backward would increase the error value. This image represents that function



Gradient vector

Remember, a neural network represents a multi-variable which tries to approach the behavior of a hidden real multi-variable function. Keep this in mind.

Also, remember we can get the cost/error function that represents how good or bad does our network predicts. This function could have multiple “valleys” or local minimums, a place we wanna reach.

How can we reach on of the two “valleys” of this function (previous image)? The **gradient vector** would tell us “which direction and sense in order to reach a higher error value”. Obviously we don’t want to increase this value but decrease it, so we need to move on the same direction but opposite sense.

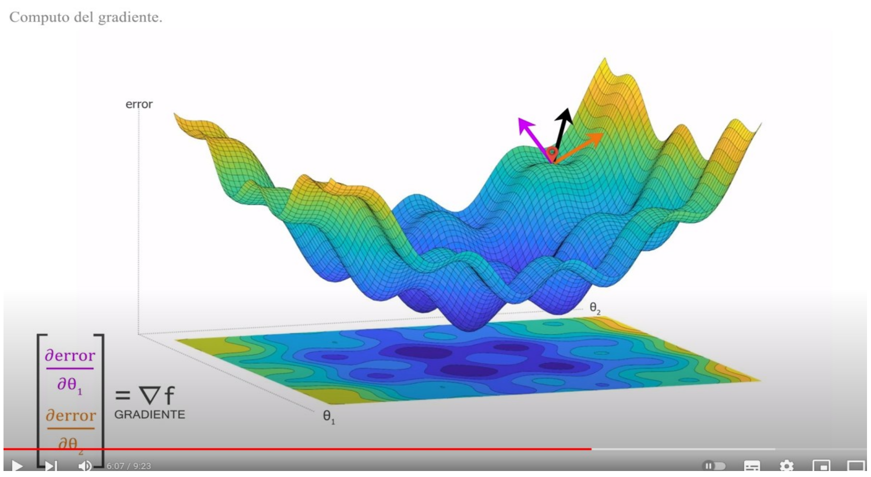
We can update the value of this gradient vector whenever we take a new step, until we reach one of this desired local minimum. That would be the same as always trying to fall until hitting ground.

In addition we can consider how long our steps can become. In neural networks it is known as **training rate**. We can adjust this value in order to avoid taking too much big steps and going to another function point with a bigger error value or becoming stack on a bad local minimum.

Formally, a gradient vector is (obviously) a vector which points the direction and sense of the best possible step to increase this value, so **reverting its sense (multiplying all of its values by -1)** we can move forward this cost/error function and find a local minimum

Trying to find the global minimum is basically luck, there are techniques that allow us to increase the probability of finding it, but we can’t assure we are going to find it. Luckily finding a local minimum is usually acceptable.

A famous analogy is using this gradient vector to find one of the lowest valleys in this scenario:


Computo del gradiente.
error
θ2
θ1
$\frac{\partial error}{\partial \theta_1}$
$\frac{\partial error}{\partial \theta_2}$
$= \nabla f$
GRADIENTE
6:07 / 9:23

However, this task is more difficult because real multi-variable functions have more than two variables so we cannot represent them like a tridimensional space.

Let’s show an theoretical implementation example, like we did with the neural network:

Gradient_vector = [

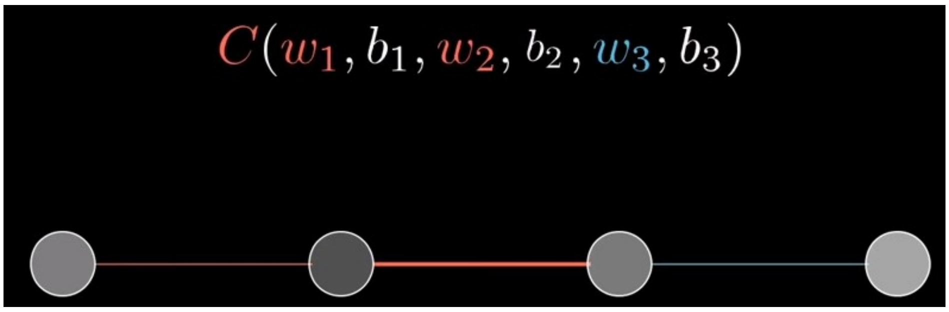
```
	Layers_error_bias,
	Layers_error_weights
]

Layers_error_bias = [
	[ matrix 1x5 ],
	[ matrix 1x3 ],
	[ matrix 1x2 ]
]

Layers_error_weights = [
	[ matrix 4x5 ],
	[ matrix 5x3 ],
	[ matrix 3x2 ]
]
```

All matrix values will equal to 0

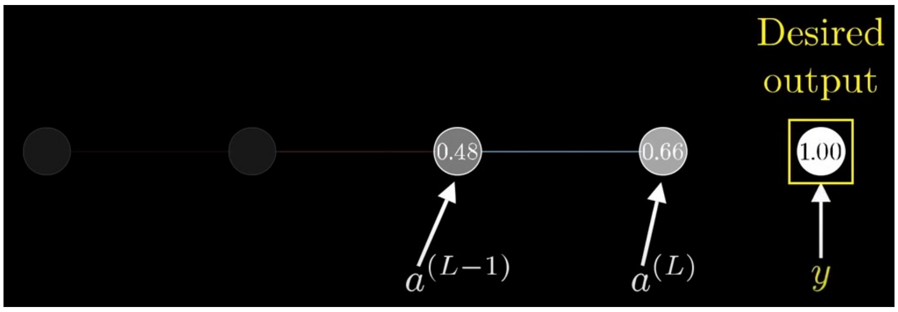
Math behind network learning

Now we are going to dive into the math behind the back propagation algorithm. Remember, the main idea of back propagation algorithm is adjusting the bias of the nodes an the weights of the edges. Also we are going to compute the gradient vector, which allows us training our network automatically.

Let's start with a simpler example, a neural network where each layer has only one node. We will need to adjust three weights and 3 biases



Now we forward the input value and move to the output layer.



We are going to represent the current network output as a(L), and our previous network output as a(L-1). Our neuron in the output layer predicted the 0.66 value, but we expected 1.00 as output value for this training example.

There are many cost/error functions, but we are going to introduce the mean squared error function, which formula is:

$$C_0(\ldots) = (a^{(L)} - y)^2$$

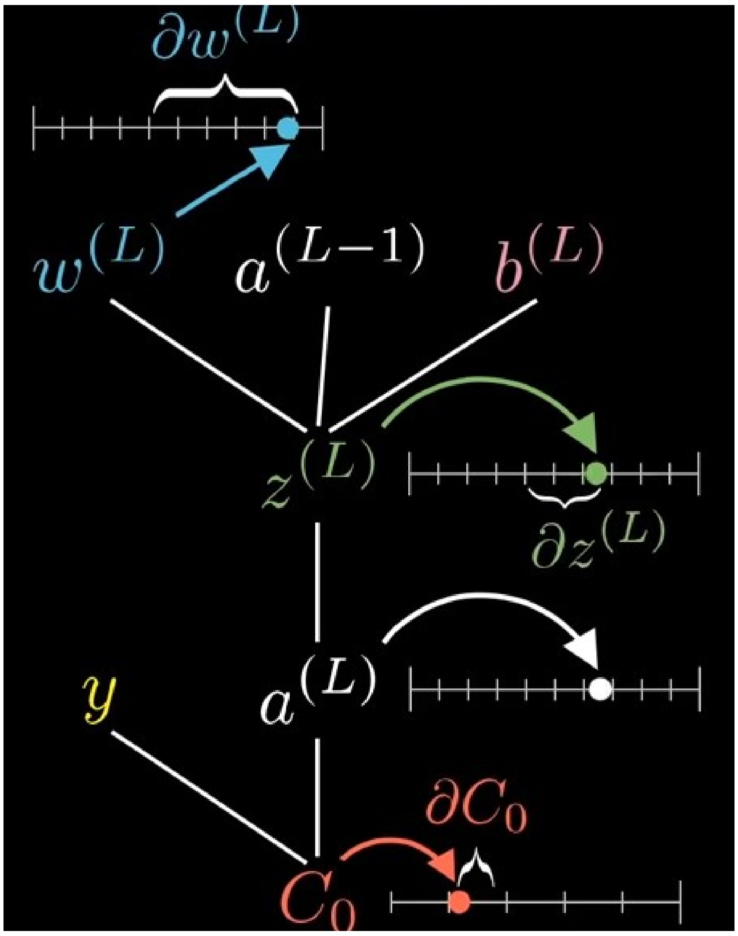
$$\text{For example: } (0.66 - 1.00)^2$$

We are not going to use this value directly to back propagate it, but cost comes from here. We can use it to debug wether the network is predicting well or not. Noww we can deduce this expressions:

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$

First we are going to calculate the weight error. Pay attention to this representation:



We can see that z(L) value depends on w(L), a(L) value depends on z(L), and Co(L) value depends on a(L) value. We can see there is a relation between this variables. This chain relation allow us to express the partial derivative of C0 respect of w(L), which is our goal. This is how the expression look like:

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}}$$

We can also express the derivative of the cost respect of b(L), and that’s because z(L) depends on b(L) value too:

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}}$$

Finally we can also deduce the derivative of the cost respect of a(L-1). This will allow us continue computing the derivatives for the previous layers:

$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}}$$

With this expressions we can calculate all the bias and weight errors. The next step is to obtain this derivatives. **The first one is the derivative of the cost respect of a(L), only in the output layer, and considering the MSE as cost/error function:**

$$\frac{\partial C0}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

Otherwise (we are computing the error between two hidden layers), this value will equal to the derivative of the cost respect of a(L-1) obtained in the previous iteration.

Next, the derivative of a(L) respect of z(L) equals:

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

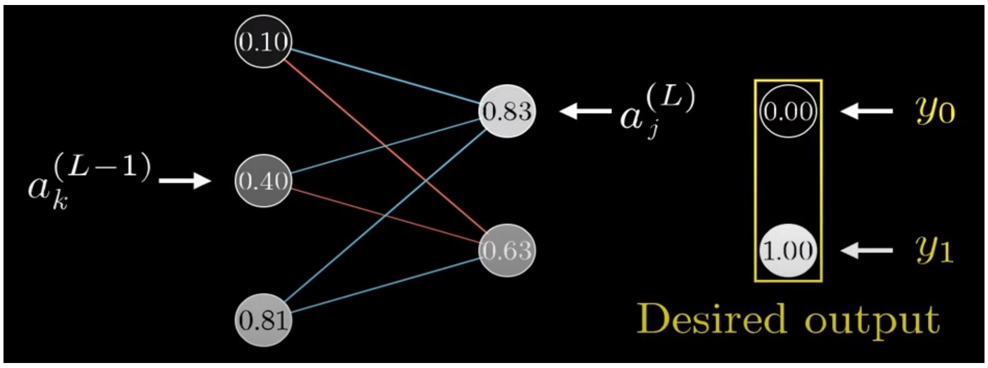
Now, remember z(L) expression, z(L) = a(L-1) * w(L) + b(L). The derivative of z(L) respect of w(L) equals to:

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

Finally, the derivative of the z(L) respect of b(L) equals to:

$$\frac{\partial C_0}{\partial b^{(L)}} = 1$$

This example looks very simplified, but only a little bit of changes would be enough to generalize for every neural network. Pat attention to this edges and nodes representation:



We can summary all derivatives as this expressions:

L = output layer

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \boxed{a_k^{(L-1)}} \, \boxed{\sigma'(z_j^{(L)})} \, \boxed{2 \cdot (a_j^{(L)} - y_j)}$$

$$\frac{\partial C_0}{\partial b_j^{(L)}} = \boxed{1} \, \boxed{\sigma'(z_j^{(L)})} \, \boxed{2 \cdot (a_j^{(L)} - y_j)}$$

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L - 1} \left(\boxed{w_{jk}^{(L)}} \, \boxed{\sigma'(z_j^{(L)})} \, \boxed{2 \cdot (a_j^{(L)} - y_j)} \right)$$

$L \neq$ output layer

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \boxed{a_k^{(L-1)}}\ \boxed{\sigma'\left(z_j^{(L)}\right)}\ \boxed{\frac{\partial C_0}{\partial a_j^{(L)}} \text{ from previous iteration}}$$

$$\frac{\partial C_0}{\partial b_j^{(L)}} = \boxed{1}\ \boxed{\sigma'\left(z_j^{(L)}\right)}\ \boxed{\frac{\partial C_0}{\partial a_j^{(L)}} \text{ from previous iteration}}$$

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \left(\boxed{w_{jk}^{(L)}}\ \boxed{\sigma'\left(z_j^{(L)}\right)}\ \boxed{\frac{\partial C_0}{\partial a_j^{(L)}} \text{ from previous iteration}} \right)$$

Now the derivative of the cost respect of a node from the previous layer equals to the sum of the product of the weight, the derivative activation function of z_j(L) and the error from the current iteration. Just think the value of this node in the previous layer will take part on all the values of the nodes of the current layer, so going backwards need to sum all the results of the nodes in the current layer.

All this derivatives look like a headache, but luckily we can use matrix’s power to simplify all this process.

Using the matrix’s power to calculate the gradient vector

We previously have seen some cumbersome derivatives we need to put in practice. How are we going to archive this goal? Well, using the matrix’s power!

Let’s move to the output layer, we first can calculate all the errors for the biases in the current layer. This is the pseudo-code to archive it:

```
d_MSE = [ 2*(output_layer[i] – expected_output[i]) for i in range(len( output_layer )) ]
```

```
Error_bias = matrix_1_row( [ d_ActiveFunc(zl[i])*d_MSE[i] for i in range(len( output_layer )) ] )
```

$$\frac{\partial C_0}{\partial b_j^{(L)}} = \boxed{1}\ \boxed{\sigma'\left(z_j^{(L)}\right)}\ \boxed{2\cdot\left(a_j^{(L)}-y_j\right)}$$

Error bias equals to a matrix with 1 row and number of nodes in output layer cols. We will insert this matrix as the first element of Gradient vector[Layers_error_bias]

Next, we are going to calculate all the weights errors, simply using matrix’s power:

```
Error_weights = matrix_product( matrix_transpose(al_1), Error_bias )
```

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \boxed{a_k^{(L-1)}}\ \boxed{\sigma'\left(z_j^{(L)}\right)}\ \boxed{2\cdot\left(a_j^{(L)}-y_j\right)}$$

al_1 is the matrix of the output values in the previous layer. We have done all operations with a single line of code!

Now we are going to calculate the errors for each of the nodes in the previous layer. We are going to use matrix’s power again:

```
Error_prev_layer = matrix_product( Error_bias, matrix_transpose(weigths_current_layer) )
```

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1}\left(\boxed{w_{jk}^{(L)}}\ \boxed{\sigma'\left(z_j^{(L)}\right)}\ \boxed{2\cdot\left(a_j^{(L)}-y_j\right)}\right)$$

Now we move to the previous layer (next iteration), and now we need to calculate the Error_bias. We can archive it using this pseudo-code (zl_1 is the matrix of all values of z in the previous layer, this next iteration):

```
Error_bias_prev = [ Error_bias[i] *  d_ActiveFunc(zl_1[i]) for i in range(len(Error_bias)) ]
```

$$\frac{\partial C_0}{\partial b_j^{(L)}} = \boxed{1} \; \boxed{\sigma'\left(z_j^{(L)}\right)} \; \boxed{\frac{\partial C_0}{\partial a_j^{(L)}} \;\text{from previous iteration}}$$

We can continue until reaching the input layer.

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \boxed{a_k^{(L-1)}} \; \boxed{\sigma'\left(z_j^{(L)}\right)} \; \boxed{\frac{\partial C_0}{\partial a_j^{(L)}} \;\text{from previous iteration}}$$

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \left(\boxed{w_{jk}^{(L)}} \; \boxed{\sigma'\left(z_j^{(L)}\right)} \; \boxed{\frac{\partial C_0}{\partial a_j^{(L)}} \;\text{from previous iteration}} \right)$$

Now, remember the structured of the Network (page 3) and the Gradient vector (pages 7-8) lists we exposed before. A possible pseudo-code would be (expected_output is a list of expected output values):

```
d_MSE = [ 2*(Network[ Layers_values_al ][number_layers-2][0][i] - output_layer[i]) for i in range(len( output_layer )) ]

Error_bias = matrix_1_row( [ d_ActiveFunc(Network[ Layers_values_zl ][number_layers-2][0][i])*d_MSE[i] for i in range(len( d_MSE )) ] )

for L in range(number_layers-2,0): #last L value would be 1

    Gradient_vector[Layers_error_bias].insert_left(Error_bias)

    Error_weights = matrix_product( matrix_transpose(Network[ Layers_values_al ][L-1]), Error_bias )

    Gradient_vector[Layers_error_weights].insert_left(Error_weights)

    Error_prev_layer = matrix_product( Error_bias, matrix_transpose(Network[Layers_weights][L]) )

    Error_bias = matrix_1_row( [ d_ActiveFunc(Network[ Layers_values_zl ][L-1][0][i])*Error_prev_layer[0][i] for i in range(len( Error_prev_layer[0] )) ] )
```

You are welcome to review the provided python code and try to establish the relationship between the theory and the practical part. Good luck!

If you wanna execute the program you can try it out too! You won't probably always make good predictions, so we are going to analyze all variables involved in network training and how to approach to best possible predictions.

Bibliography

Common activation functions: https://www.v7labs.com/blog/neural-networks-activation-functions

Why do we need activation functions? (Spanish): https://www.youtube.com/watch?v=_0wdproot34

How gradient vector works in ann (Spanish): https://www.youtube.com/watch?v=A6FiCDoz8_4

Key concepts for understanding better ann (Spanish): https://medium.com/metadatos/todo-lo-que-necesitas-saber-sobre-el-descenso-del-gradiente-aplicado-a-redes-neuronales-19bdbb706a78

Nice video explaining math behind back propagation algorithm: https://www.youtube.com/watch?v=tIeHLnjs5U8