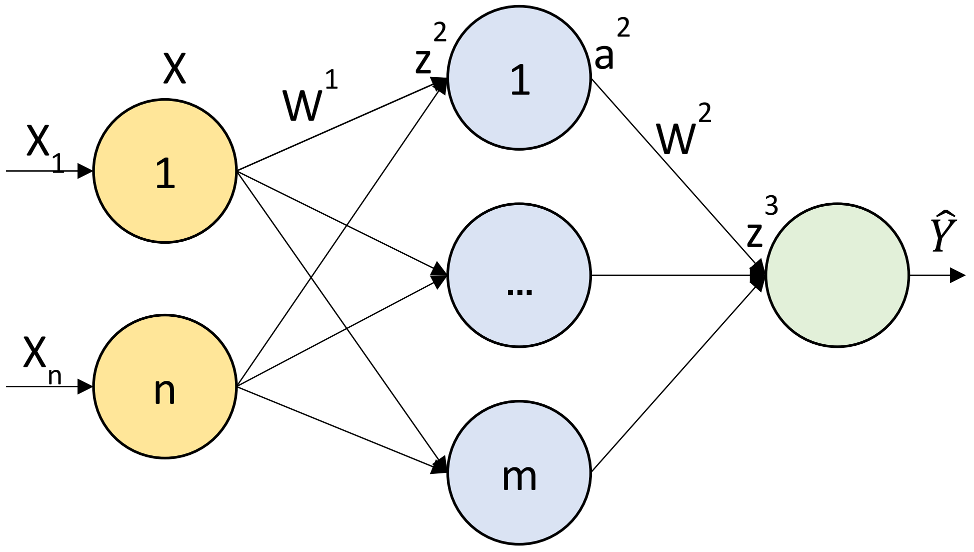
Neural Networks – The Graph Approach

1. Introduction

As in the previous parts, we will be working with the same neural network structure.



As we have seen in the learning paragraph of the first chapter, the neural networks update their weights based on the error committed by forwarding the inputs and comparing these values with the 'actual values', or what these outputs should have been.

Mathematically, we compute the gradient of these errors, and update the weights as follows:

$$W^i(k+1) = W^i(k) - \lambda_i \cdot \frac{dJ}{dW^i}(k) \quad \text{(Eq. 1)}$$

To train neural networks we need to provide them with large amounts of data, so the process of try, check the error and learn from it can be iterated as much as possible (without overfitting!)

To make this computationally feasible, the calculations rely on a graph approach as we will see in this chapter.

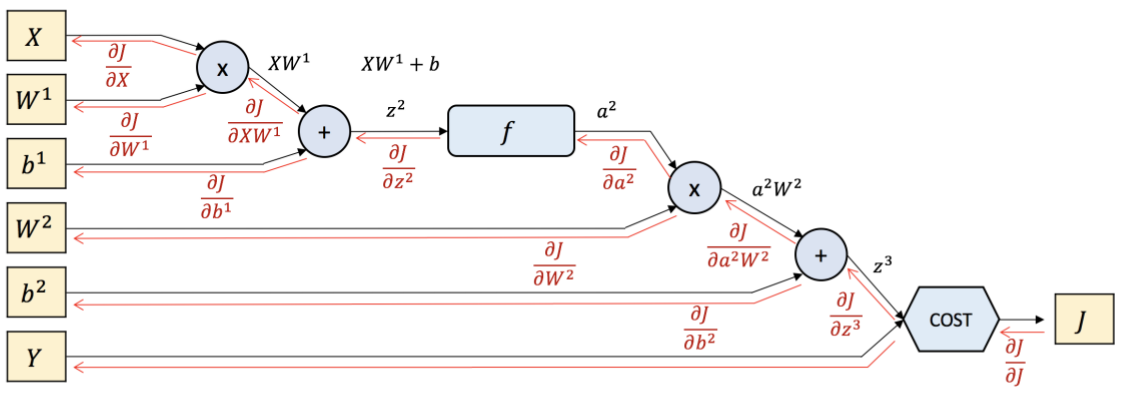
To make things easy, we will work on the same networks. Let's decompose it into a graph!

2. Graph Representation of Neural Networks

The time has come to discover how our neural networks are behaving. If you have ever used a machine learning framework like TensorFlow or Keras (links), what we are going to do is trying to identify what is behind that few lines of code that simplifies your life.

Well, believe it or not, our friendly neural network we have been working with is exactly like in Figure 1 is represented. But don't worry, we will split in into smaller pieces to make sure we understand every component and, at the end, we will code it ourselves!

Figure 1. Graph representation of out [2, 3, 1] neural network.



X
W^1
b^1
W^2
b^2
Y
$\frac{\partial J}{\partial X}$
$\frac{\partial J}{\partial W^1}$
$\frac{\partial J}{\partial b^1}$
x
XW^1
$\frac{\partial J}{\partial XW^1}$
+
$XW^1 + b$
z^2
$\frac{\partial J}{\partial z^2}$
f
a^2
$\frac{\partial J}{\partial a^2}$
x
a^2W^2
$\frac{\partial J}{\partial a^2W^2}$
$\frac{\partial J}{\partial W^2}$
+
$\frac{\partial J}{\partial b^2}$
z^3
$\frac{\partial J}{\partial z^3}$
COST
J
$\frac{\partial J}{\partial J}$

Let's have a first look an identify what is every shape and colour:

Yellow boxes are variables. X is the input, Y is ~~the output~~ what the output of the forward process should be; and J is the error between the real output and Y.

Blue shapes correspond to mathematical operations between the layers. They are basically functions, but some of them are just an addition or a multiplication, so in those cases we just see the sign.

Now we can go to last chapter and refresh how we were doing the forward process by the several steps of multiplying the different matrices. Let's write down the equations we can infer for the graph and check if they agree with our previous knowledge.

$$XW^1 = X * W^1 \qquad \text{(Eq. F1)}$$

$$z^2 = X * W^1 + b^1 \qquad \text{(Eq. F2)}$$

$$a^2 = f(z^2) = f(X * W^1 + b^1) \qquad \text{(Eq. F3)}$$

$$z^3 = a^2W^2 = a^2 * f(X * W^1 + b^1) \qquad \text{(Eq. F4)}$$

$$z^3 = a^2W^2 + b^2 = a^2 * f(X * W^1 + b^1) + b^2 \qquad \text{(Eq. F5)}$$

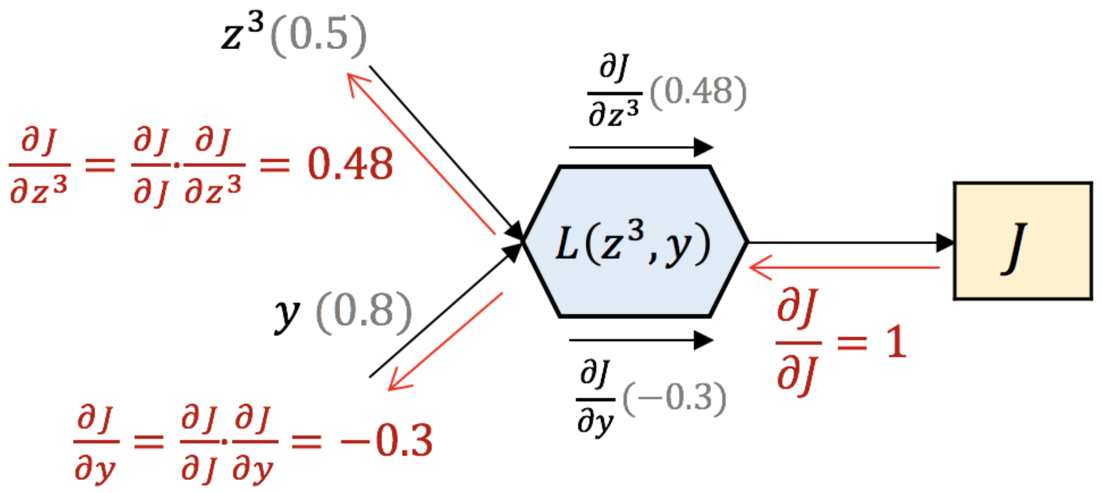
$$\hat{Y} = f(z^3) = (f(a^2 * f(X * W^1 + b^1) + b^2) = z^3) \qquad \text{(Eq. F6)}$$

$$J = \frac{1}{n}(\hat{Y} - Y)^2 = \frac{1}{n}(z^3 - Y)^2 \qquad \text{(Eq. F7)}$$

That actually makes sense, right? We have the same equation for the forward process that we had in the second chapter of the first document (LINK).

We are ready to enter in the BackProp direction of our neural network. This is getting interesting, so let's explain once the details with the example of the first back-propagated error, and see how this process is basically repeated equally until we reach the beginning of our network.

Figure 2.First Backward Propagation Step



$z^3 (0.5)$
$\frac{\partial J}{\partial z^3}(0.48)$
$\frac{\partial J}{\partial z^3} = \frac{\partial J}{\partial J} \cdot \frac{\partial J}{\partial z^3} = 0.48$
$L(z^3, y)$
J
$y\ (0.8)$
$\frac{\partial J}{\partial J} = 1$
$\frac{\partial J}{\partial y}(-0.3)$
$\frac{\partial J}{\partial y} = \frac{\partial J}{\partial J} \cdot \frac{\partial J}{\partial y} = -0.3$

The idea we need to get from this comes as follows:

First, we input our inputs z^3 and y as the inputs to a function (COST) which is described in (Eq. F7. Their values are given in grey. Therefore, right at that moment we are able to calculate the ***local gradients***. These are the partial derivatives of the current equation (we see them in black under and above the box). If we take (Eq. F7 and we derivate it, the result is (given that we have 1 time step only so n = 1):

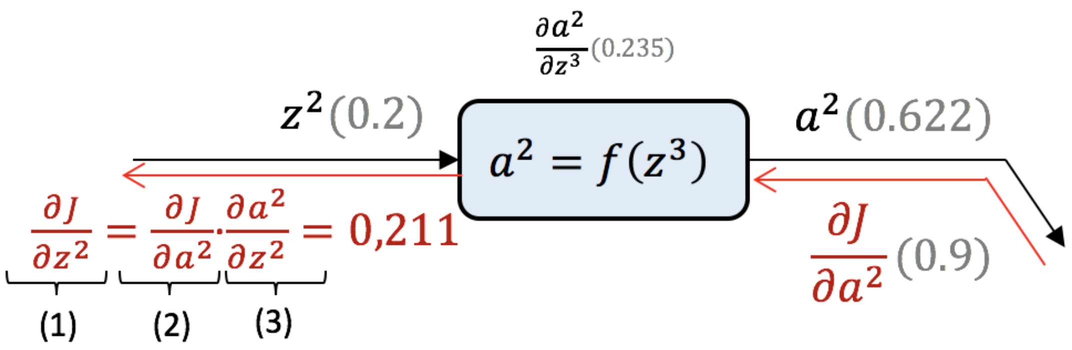
$$\frac{dJ}{dz^3} = -\frac{2}{n} \cdot y \cdot (z^3 - y) = -\frac{2}{1} \cdot 0.8 \cdot (0.5 - 0.8) = \ 0.48$$

$$\frac{dJ}{dy} = \frac{2}{n} \cdot z^3 \cdot (z^3 - y) = \frac{2}{1} \cdot 0.5 \cdot (0.5 - 0.8) = \ -0.3$$

Next, we always start a BackProp process with the local gradient of the function itself, which is obviously 1. After that, we want to propagate backwards that error to the responsible of the error committed. What we do is apply the chain rule (LINK) to decompose one derivative into simpler ones. Now if we take a look, we have solved a derivative we didn't know what its value was, by splitting in into two derivatives that we actually now their values already!

You must have realized that this first step is maybe not the most illustrative, as the derivative of the cost function by itself is 1, and therefore the chain rule seems like it's not doing anything. However, it is going to help us when looking at any other step at the backward propagation process, like in Figure 3. There, we have taken as an illustration the process where we will apply an activation function to introduce the non-linearity in the system.

Figure 3. Any Backward Propagation Step



$\frac{\partial a^2}{\partial z^3}(0.235)$
$z^2(0.2)$
$a^2 = f(z^3)$
$a^2(0.622)$
$\frac{\partial J}{\partial z^2} = \frac{\partial J}{\partial a^2} \cdot \frac{\partial a^2}{\partial z^2} = 0{,}211$
$\frac{\partial J}{\partial a^2}(0.9)$
(1)
(2)
(3)

The activation function used in this illustration is the sigmoid function.

Sigmoid function: $$\sigma(x) = \frac{1}{1+e^{-x}}$$

Derivative of sigmoid function: $$\sigma'^{(x)} = \sigma(x) \cdot \left(1 - \sigma(x)\right)$$

Repeating the same process, we have the input z^2 which is being applied the sigmoid function to obtain a^2. At that point, we can calculate the local gradient, and wait for the back-propagated error to come from the end of the network.

Then, when it reaches, we have the $\frac{dJ}{da^2}$ which basically means 'how much responsible of the value of J is a^2'. And applying chain rule, we can determine how much responsible for that J is z^2, the next on the line.

To do so, we apply the same as for the previous example. We use the chain rule to express the gradient as a product of the local gradient and the error back-propagated until this point. Those to values are known and we can easily compute our intention then:

Now ask yourself and try to answer the following question, why are we doing this? Why back-propagating the value of the J cost? We are jumping a little bit into the next chapter (4. Learning (LINK)). We want to know 'how much responsible are the weights of the neurons to the error committed', so we can adjust their values and make our network learn. So, in the end, we want to find the $\frac{dJ}{dW^1}$ and $\frac{dJ}{dW^2}$.

$$\frac{dJ}{da^2} = 0.9$$

$$\frac{dJ}{dz^2} = \frac{dJ}{da^2} \cdot \frac{da^2}{dz^2} = 0.9 * 0.235 = 0.211$$

Good, so now that we not the tools and the objectives, let's develop are backward equations for our network friend!

$$\frac{dJ}{dz^3} = -= -\frac{2}{n} \cdot y \cdot (\mathrm{z}^3 - \mathrm{y}) = \boldsymbol{top_{diff}} \qquad \text{(Eq. B1)}$$

$$\frac{dJ}{da^3W^2} = \frac{dJ}{dz^3} \cdot \frac{dz^3}{da^3W^2} = \boldsymbol{top_{diff}} \cdot \boldsymbol{local_{diff}} = top_{diff} \cdot 1 \qquad \text{(Eq. B2)}$$

$$\frac{dJ}{db^2} = \frac{dJ}{dz^3} \cdot \frac{dz^3}{db^2} = \boldsymbol{top_{diff}} \cdot \boldsymbol{local_{diff}} = top_{diff} \cdot 1 \qquad \text{(Eq. B3)}$$

$$\frac{dJ}{da^2} = \frac{dJ}{dz^3} \cdot \frac{dz^3}{da^2W^2} \cdot \frac{da^2W^2}{da^2} = \boldsymbol{top_{diff}} \cdot \boldsymbol{way_{here}} \cdot \boldsymbol{local_{diff}} \\ = topp_{diff} \cdot 1 \cdot W^2 = topp_{diff} \cdot W^2 \qquad \text{(Eq. B4)}$$

$$\boldsymbol{\frac{dJ}{dW^2}} = \frac{dJ}{dz^3} \cdot \frac{dz^3}{da^2W^2} \cdot \frac{da^2W^2}{dW^2} = \boldsymbol{top_{diff}} \cdot \boldsymbol{way_{here}} \cdot \boldsymbol{local_{diff}} \\ = topp_{diff} \cdot 1 \cdot a^2 = topp_{diff} \cdot a^2 \qquad \text{(Eq. B5)}$$

$$\frac{dJ}{dz^2} = \frac{dJ}{dz^3} \cdot \frac{dz^3}{da^2W^2} \cdot \frac{da^2W^2}{da^2} \cdot \frac{da^2}{dz^2} = \boldsymbol{top_{diff}} \cdot \boldsymbol{way_{here}} \cdot \boldsymbol{local_{diff}} \\ = topp_{diff} \cdot W^2 \cdot f'(\mathrm{z}^2) \qquad \text{(Eq. B6)}$$

$$\frac{dJ}{dXW^1} = \frac{dJ}{dz^3} \cdot \frac{dz^3}{da^2W^2} \cdot \frac{da^2W^2}{da^2} \cdot \frac{da^2}{dz^2} \cdot \frac{dz^2}{dXW^1} \\ = \boldsymbol{top_{diff}} \cdot \boldsymbol{way_{here}} \cdot \boldsymbol{local_{diff}} \\ = topp_{diff} \cdot W^2 \cdot f'(\mathrm{z}^2) \cdot 1 \qquad \text{(Eq. B7)}$$

$$\frac{dJ}{db^1} = \frac{dJ}{dz^3} \cdot \frac{dz^3}{da^2W^2} \cdot \frac{da^2W^2}{da^2} \cdot \frac{da^2}{dz^2} \cdot \frac{dz^2}{db^1} = \boldsymbol{top_{diff}} \cdot \boldsymbol{way_{here}} \cdot \boldsymbol{local_{diff}} \\ = topp_{diff} \cdot W^2 \cdot f'(\mathrm{z}^2) \cdot 1 \qquad \text{(Eq. B8)}$$

$$\frac{dJ}{dX} = \frac{dJ}{dz^3} \cdot \frac{dz^3}{da^2W^2} \cdot \frac{da^2W^2}{da^2} \cdot \frac{da^2}{dz^2} \cdot \frac{dz^2}{db^1} = \boldsymbol{top_{diff}} \cdot \boldsymbol{way_{here}} \cdot \boldsymbol{local_{diff}} \\ = topp_{diff} \cdot W^2 \cdot f'(\mathrm{z}^2) \cdot W^1 \qquad \text{(Eq. B9)}$$

$$\boldsymbol{\frac{dJ}{dW^1}} = \frac{dJ}{dz^3} \cdot \frac{dz^3}{da^2W^2} \cdot \frac{da^2W^2}{da^2} \cdot \frac{da^2}{dz^2} \cdot \frac{dz^2}{dW^1} \\ = \boldsymbol{top_{diff}} \cdot \boldsymbol{way_{here}} \cdot \boldsymbol{local_{diff}} \\ = topp_{diff} \cdot W^2 \cdot f'(\mathrm{z}^2) \cdot X \qquad \text{(Eq. B10)}$$

There we go! We have done all the BackProp process and compute the gradients we are interested in so easy! Those are the values we will use to update the values for the weights and reduce the error for the next time. But now, let's take a closer look and do one thing humans still do better than machines, let's find patterns.

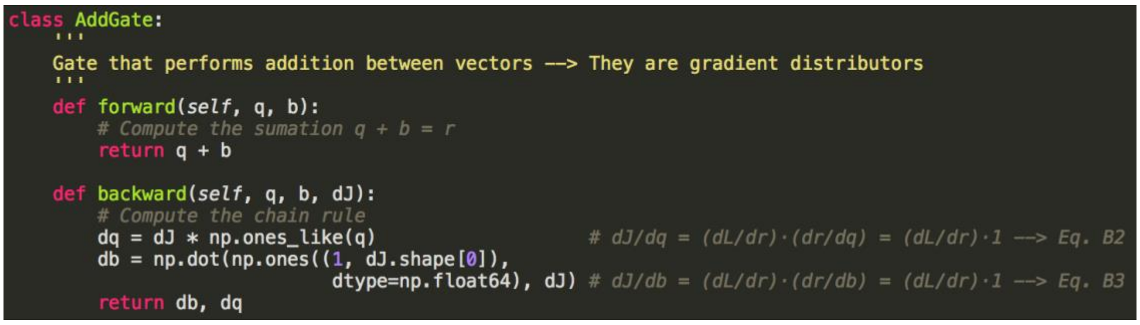
As we saw in detail in Figure 2 and Figure 3, the backpropagation for every component of the global network consist basically in multiplying the gradient that receives from the end of the network by the local gradient. That's the magic of AI and is that simple!

Another pattern we see is how the different blocks we have work. In AI world, they have their own terminology, as follows:

Add Gates:

In the forward pass, add gates gives the addition of the inputs as the output.

In the backward pass, add gates are called distributors. Take a look at one of the pair of equations B2-B2 or B7-B8. They represent the BackProp process of each of the two add gates we have in the network. The local gradient for both of them is 1, that's why we call them *distributors*. They just capture the gradient that comes to it, and pass it untouched to all of its inputs.

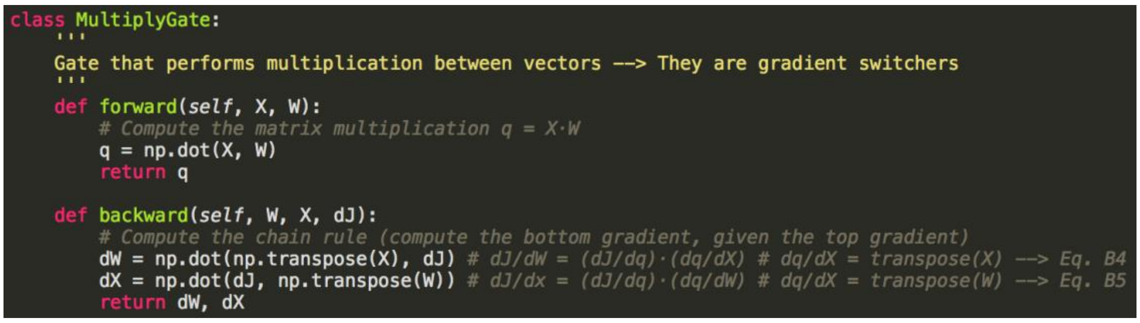


```
class AddGate:
    '''
    Gate that performs addition between vectors --> They are gradient distributors
    '''
    def forward(self, q, b):
        # Compute the sumation q + b = r
        return q + b

    def backward(self, q, b, dJ):
        # Compute the chain rule
        dq = dJ * np.ones_like(q)                     # dJ/dq = (dL/dr)·(dr/dq) = (dL/dr)·1 --> Eq. B2
        db = np.dot(np.ones((1, dJ.shape[0]),
                            dtype=np.float64), dJ) # dJ/db = (dL/dr)·(dr/db) = (dL/dr)·1 --> Eq. B3
        return db, dq
```

Multiply Gates:

In the forward pass, multiply gates gives the multiplication of the inputs as the output.

In the backward pass, multiply gates are called switchers. Take a look at one of the pair of equation B4-B5 or B9-B10. They represent the BackProp process of each of the multiply gates in the network. The local gradient for both of them is the value of the other input, that's why we call them *switchers*.

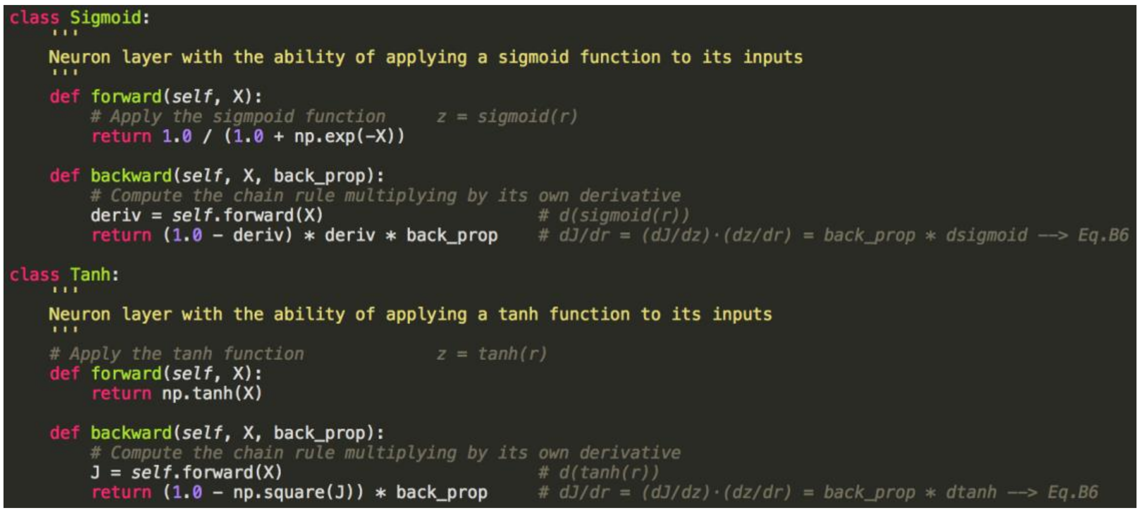


```
class MultiplyGate:
    '''
    Gate that performs multiplication between vectors --> They are gradient switchers
    '''
    def forward(self, X, W):
        # Compute the matrix multiplication q = X·W
        q = np.dot(X, W)
        return q

    def backward(self, W, X, dJ):
        # Compute the chain rule (compute the bottom gradient, given the top gradient)
        dW = np.dot(np.transpose(X), dJ) # dJ/dW = (dJ/dq)·(dq/dX) # dq/dX = transpose(X) --> Eq. B4
        dX = np.dot(dJ, np.transpose(W)) # dJ/dx = (dJ/dq)·(dq/dW) # dq/dX = transpose(W) --> Eq. B5
        return dW, dX
```

Layers:

In the forward pass, layers apply the function we have selected for them (the activation function) to the input to calculate the output.

In the backward pass, layers just multiply its own derivative value to the incoming gradient to calculate the output gradient (also called bottom gradient). Take a look at equation 6. Two typical activation functions are sigmoid or tanh.

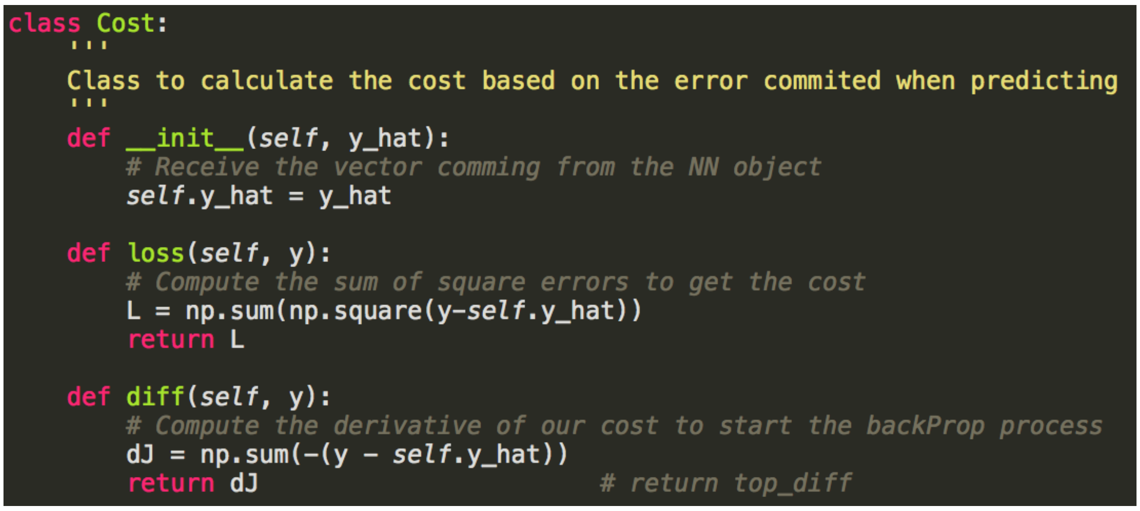


```python
class Sigmoid:
    '''
    Neuron layer with the ability of applying a sigmoid function to its inputs
    '''
    def forward(self, X):
        # Apply the sigmpoid function     z = sigmoid(r)
        return 1.0 / (1.0 + np.exp(-X))

    def backward(self, X, back_prop):
        # Compute the chain rule multiplying by its own derivative
        deriv = self.forward(X)                   # d(sigmoid(r))
        return (1.0 - deriv) * deriv * back_prop    # dJ/dr = (dJ/dz)·(dz/dr) = back_prop * dsigmoid --> Eq.B6

class Tanh:
    '''
    Neuron layer with the ability of applying a tanh function to its inputs
    '''
    # Apply the tanh function             z = tanh(r)
    def forward(self, X):
        return np.tanh(X)

    def backward(self, X, back_prop):
        # Compute the chain rule multiplying by its own derivative
        J = self.forward(X)                       # d(tanh(r))
        return (1.0 - np.square(J)) * back_prop    # dJ/dr = (dJ/dz)·(dz/dr) = back_prop * dtanh --> Eq.B6
```

Cost Calculators:

In this particular case, we have applied a cost function to minimize the mean squared error. However, the architecture of this box and the values on it are going to be different depending on the application of the neural network.

In this particular case then, it receives the output of the forward process of the inputs and compare it with what the value should be. After that, it returns the value of the derivative, what we have been calling top_diff.



```python
class Cost:
    '''
    Class to calculate the cost based on the error commited when predicting
    '''
    def __init__(self, y_hat):
        # Receive the vector comming from the NN object
        self.y_hat = y_hat

    def loss(self, y):
        # Compute the sum of square errors to get the cost
        L = np.sum(np.square(y-self.y_hat))
        return L

    def diff(self, y):
        # Compute the derivative of our cost to start the backProp process
        dJ = np.sum(-(y - self.y_hat))
        return dJ                        # return top_diff
```