Handwritten digits recognition Neural Network

(Submission to 2022 Qiskit fall festival)

Abdelghani EL OUARDI

November 1st,2022

1. Building a handwritten digits recognition neural network on classical computer

1.1 The structure of neural network

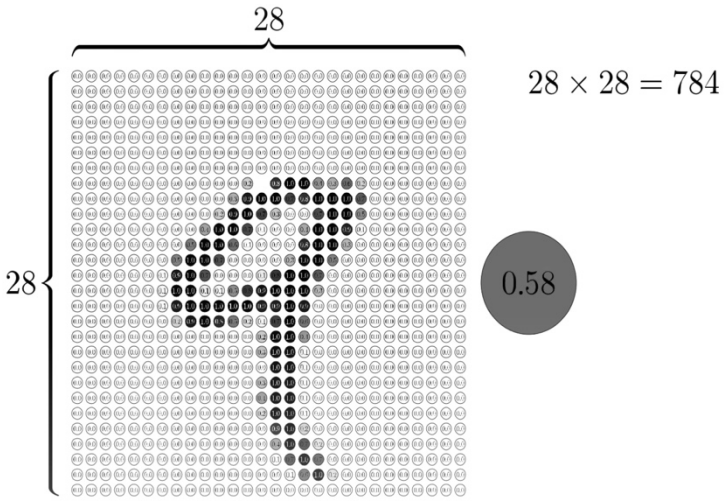
The handwritten digits recognition is somewhat of a classical example for introducing the topic of machine learning, or what some may call it the **"hello world"** of machine learning.

As the name suggest, Neural networks are inspired by the brain, but in what sense? what exactly do we mean by neurons? And in what way they are connected?

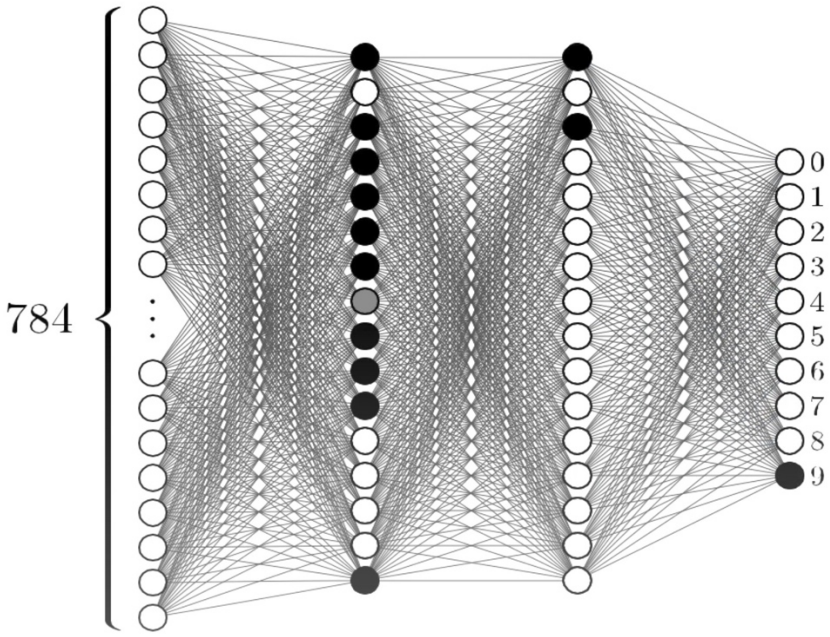
When we talk about neuron, what we should imagine is an object that hold a number, or what's called an activation number, that is between 0 and 1.

For example, the network starts with a bunch of neurons corresponding to each of the 28 X 28 pixels of the input image, each one of these holds a number, ranging from 0 for white pixels, up to one for black pixels.

These 784 neurons make up our first layer of our network.



And what we hope for is that the last layer (which will have 10 neurons corresponding to the 10 digits) will be activated in a way that correspond to the right digit. In our example that will be 9.



There are also a couple layers in between, called the hidden layers.

The way this network works is that the activation in one layer determine the activation in the next layer. And of course, the network as an information processing mechanism, comes down to exactly how those activation from one layer bring about activations in the next layer.

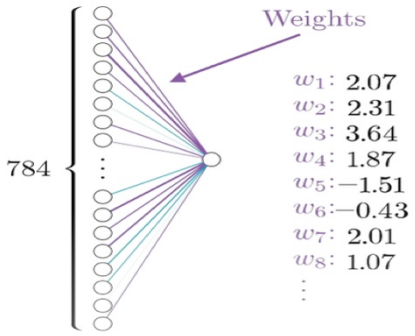
The sole purpose of this network is that when we feed it an image, the pattern at which those 784 neurons are activated, will determine how the neurons in the second layer are activated, and the pattern formed in the second layer will determen how the neurons in the third layer are activated, and finally how the neurons in the output layer will be activated. The neuron activated in the output layer, represent the network choice for what digit this image represent.

But how exactly the activation in one layer determines the activation in the next layer. To explain this mechanism, we going to focus on one specific example at how the activation in the first layer influence the activation in one neuron in the second layer.

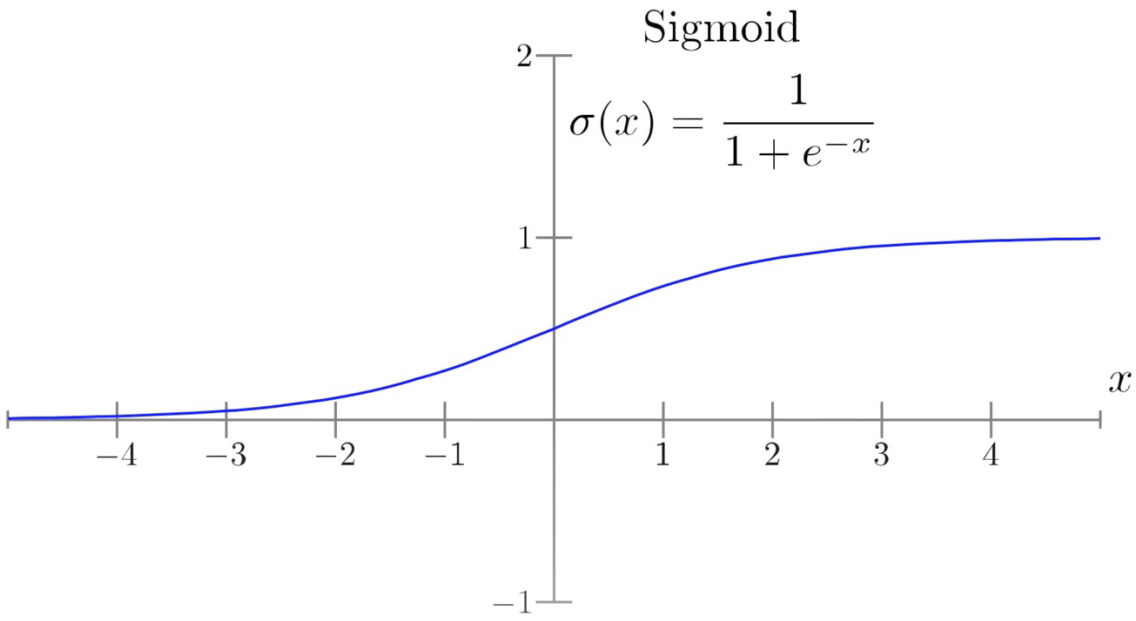
The way we do this is assigning a “weight” to the connection between the neurons of the first layer and the specific neuron in the second layer, these weights are just numbers.

And the activation in this neuron is just the weighted sum of the activation of the neurons in the first layer

$$w_1a_1 + w_2a_2 + w_3a_3 + \cdots + w_na_n$$



But since the activation in a neuron Is restricted to the values between 0 and 1, what we will do is to pump this weighted sum into the sigmoid function, also known as the logistic curve



Which basically take the value 1 for verry positive input, and 0 for very negative input.

So, the activation in the neuron in the second layer is therefor

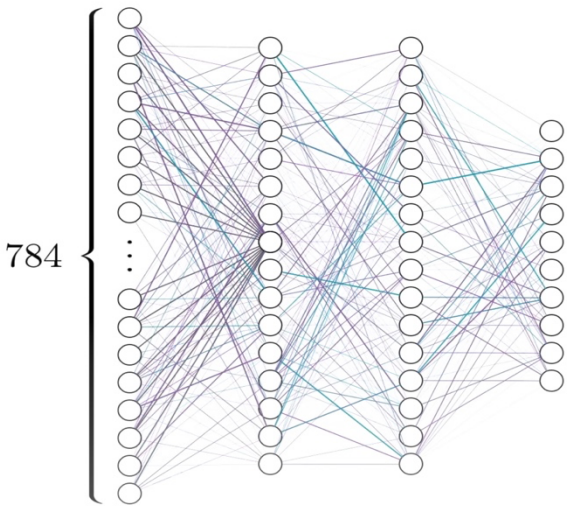
$$\sigma(w_1a_1 + w_2a_2 + w_3a_3 + \cdots + w_na_n)$$

But we don't want the activation to equal to one just because the weighted sum is bigger than 0, but rather when it's bigger than a specific number b called the bias for inactivity. And so, we will plug this number into the weighted sum before pumping it into the sigmoid function

$$\sigma(w_1a_1 + w_2a_2 + w_3a_3 + \cdots + w_na_n + b)$$

And this is just one neuron. All the other neurons will be activated according to their own weighted sum and their own bias.

And that's just the connection between the first layer and second layer, each of the connection between the other layers have their own weights and biases

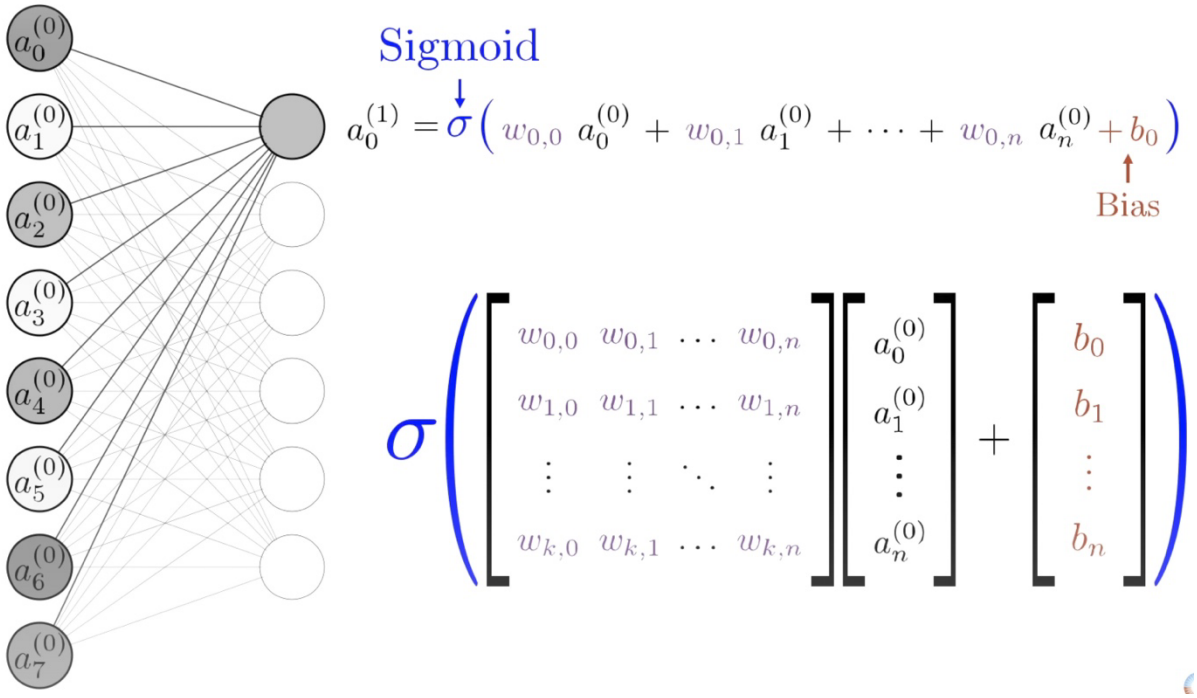


$784\times16+16\times16+16\times10$
weights

$16\ +16+10$
biases

13,002

In general, this weighted sum can be thought as a matrix transformation, where the set of activation in each layer is a vector that results from multiplying the previous layer by some matrix, then adding the bias and then applying the sigmoid function



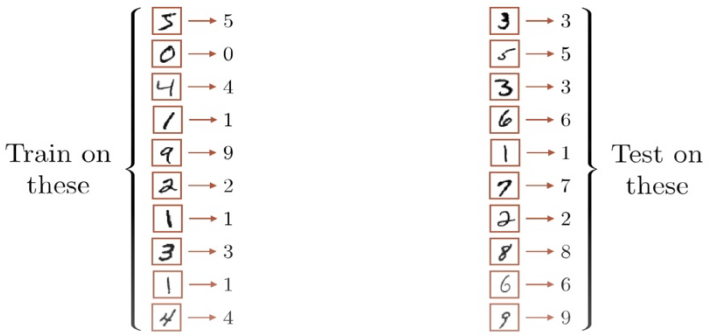
Which is summarized in the following expression

$$a_i^{(L)} = \sigma\left(\sum_j w_{ij}\, a_j^{(L-1)} + b_i\right)$$

But how is this weight values are determined? It's tempting to think that these values are pre-determined. But we won't have learning machine in that case. What actually happening is that these weights are constantly modified when exposing the machine to a training data set, the modification is achieved by a process called the **Back-propagation.**

1.2 The Gradient descent and the back-propagation

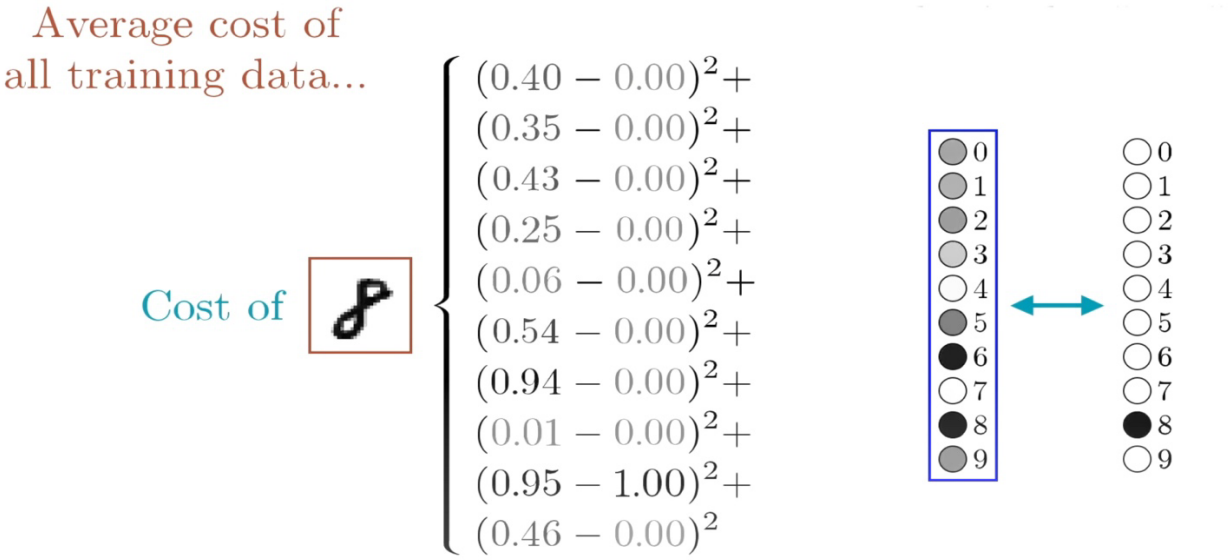
In order for the network to learn, we need an algorithm that feed this network a whole bunch of data which come in the form of a bunch of handwritten digits along with labels for what those digits supposed to be, and it will adjust those weight and biases accordingly, and hoping that the neural network will generalize what it learns to images beyond the training data.



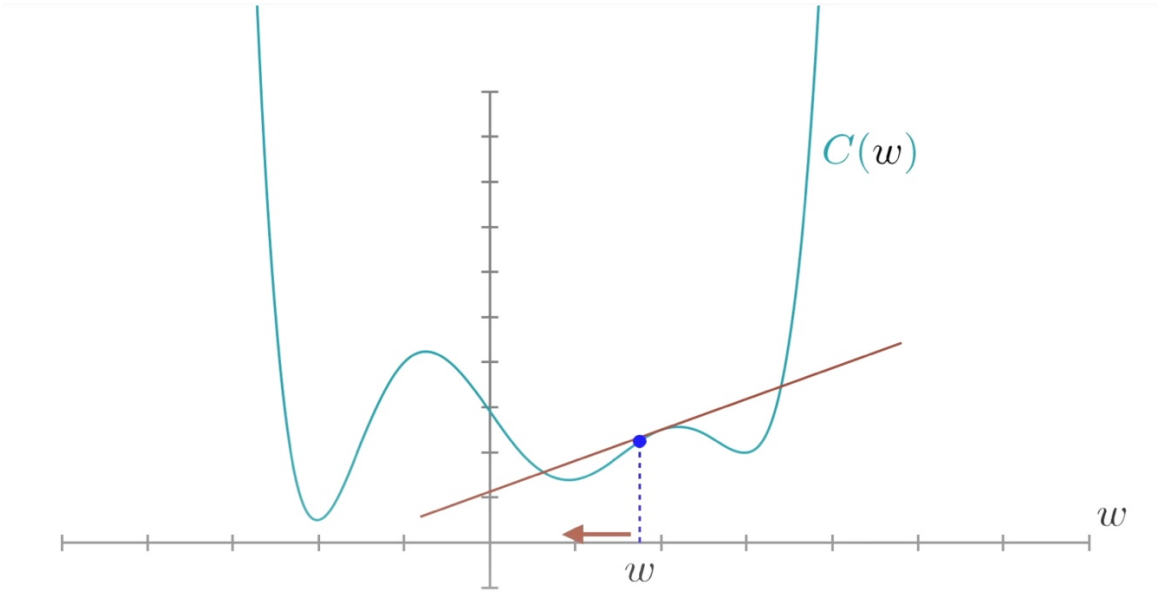
And then see how accurately it classifies those images, luckily for us, the **MNIST** data base contain a collection of 10,000s of hand written digits images along with their label

What we mean by learning, is a process at which the system tries to find the minimum of a certain function called the **cost function,** and the weights and biases an adjusted in a way that minimize this cost function. The cost function is a measure of the error i.e., how bad Is the machine at recognizing digits.

The cost function is defined as the sum of the difference between the activation value of the output layer and the corresponding label squared, of the training data. The following is an example of the cost function of the digit 8



And then the problem of training the machine is reduced to finding the weights that minimize the cost function, we can do this by the process of the gradient descent, which Is just modifying the weight in a matter that lowers the cost function.



This graph is an over simplification for one weight value, while in our case, the handwritten digits recognition neural network contains more than 13 000 connections, each associated with its own weight value.

The algorithm of minimizing this function is to compute the gradient ∇C and then taking a step in the $-\nabla C$ direction, and repeat that over and over.

$$\vec{\mathbf{W}} = \begin{bmatrix} 2.43 \\ -1.12 \\ 1.47 \\ \vdots \\ -0.76 \\ 3.50 \\ 2.03 \end{bmatrix} \qquad -\nabla C(\vec{\mathbf{W}}) = \begin{bmatrix} 0.18 \\ 0.45 \\ -0.51 \\ \vdots \\ 0.40 \\ -0.32 \\ 0.82 \end{bmatrix}$$

Where each element of $-\nabla C(\vec{w})$ tells each element of $\vec{w}$ if it should increase or decrease and by how much, in order to minimize the cost function.

we can describe this procedure mathematically by

$$w_k \longmapsto w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \longmapsto b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$$

Where η is a small positive parameter known as the learning rate.

And by repeatedly applying this update, we will be able to roll down to the nearest local minimal, this is the process at which the neural network learns.

the above cost function used in the updating process is not that of one training example but rather the average cost function of all the training data

$$\nabla C = \frac{1}{n}\sum_{x} \nabla C_x$$

Where. $C_x \equiv \frac{\|y(x)-a\|^2}{2}$ for individual training examples, the problem is the number of training examples is very large, so this will take a large time and the learning will occur very slowly.

The way we solve this is by using an idea called the stochastic gradient, the idea is to estimate the gradient ∇C by computing ∇C_x for a small sample of randomly chosen training inputs.

We will label this training samples by $X_1, X_2, X_3, \ldots, X_m$, we will refer to these as mini batch. We expect these sample sizes to be large enough for the average value ∇C_{x_j} to be roughly equal to the average over all ∇C_x

$$\frac{1}{m}\sum_{j=1}^{m} \nabla C_{x_j} \approx \frac{1}{n}\sum_{x} \nabla C_x = \nabla C$$

And then we plug this into the updating process

$$w_k \longmapsto w'_k = w_k - \frac{\eta}{m}\sum_{j=1}^{m} \frac{\partial C_{x_j}}{\partial w_k}$$

$$b_l \longmapsto b'_l = b_l - \frac{\eta}{m}\sum_{j=1}^{m} \frac{\partial C_{x_j}}{\partial b_l}$$

1.3 Implementing the handwritten Digits recognition neural network

Apart from the **MNIST** data, we also going to need a python library called **NumPy**, for doing fast linear algebra.

First of all, we have to initiate the Network object

```
class Network(object):

    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:])]
```

Where the size contains the number of neurons in the respective layers. So, if we want to create a Network with 784 neurons in the first layer, 16 neurons in the second and third layer and 10 neurons in the last layer,

we just have to write

```
net = Network([784, 16, 16, 10])
```

While all the weights and biases are initiated randomly using `np. random.randn.`

The set of weights and biases that connect the first to the second layer are stored as a list of NumPy matrices. As mention before the activation in the one layer is a function of the activation in the previous layer

$$a' = \sigma(wa + b)$$

Where w is the matrix representation of the weights connect one layer to the other, and a' a and b are vectors.

The next step is to define the sigmoid function

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

We then add a feed forward method to the network class, which apply the equation $a' = \sigma(wa + b$ to each layer

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

Now we need to implement the stochastic gradient descent in order for our neural network to learn

```
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)
```

The training data is a list of tuples (x,y) representing the training input and the corresponding desired outputs.

The code starts by randomly shuffling the training data, and then partitions it into mini batches, and then for each mini batch we apply a single step of gradient descent which is done by

`self.update_mini_batch(mini_batch, eta)`

which update the weights and biases according to a single iteration, the update_mini_bach is defined as follows

```
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)]
```

most of the work is done by

```
delta_nabla_b, delta_nabla_w = self.backprop(x, y)
```

which for every example in the mini_bach it updates self.weights and self.biases appropriately.

The back propagation function is defined

```
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)
```

This code is just an implementation of all mathematics we have seen before

We also going to have to define a couple of other functions that we will need

```
def evaluate(self, test_data):
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

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

The evaluating function Returns the number of test inputs for which the neural network outputs the correct result. Where the cost_derivative Returns the vector of partial derivatives $\frac{\partial C_x}{\partial a}$ for the output activations.

And of course, the derivative of the sigmoid function

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

No to test the program we going to start by loading the MNIST data

```
import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
```

After loading the data, we will set up a network with 30 hidden layers

```
>>> import network
>>> net = network.Network([784, 30, 10])
```

And finally, we will use the stochastic gradient descent to learn from the MNIST training data over 30 epochs, with a mini -batch size of 10, and a learning rate of $\eta = 3.0$,

```
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

here is a partial transcript of the output of one training run of the neural network. The transcript shows the number of test images correctly recognized by the neural network after each epoch of training. As you can see, after just a single epoch this has reached 9,129 out of 10,000, and the number continues to grow,

```
Epoch 0: 9129 / 10000
Epoch 1: 9295 / 10000
Epoch 2: 9348 / 10000
...
Epoch 27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

That is, the trained network gives us a classification rate of about 95 percent.

Now our next task is to implement this neural network using a quantum machine. Which is easier said than done.