MNIST Neural Network from Scratch

Luis Enrique Meza. Tecnológico de Monterrey Campus Querétaro. ISDR. A01114143.

Abstract. – In this document we can observe the process of the development of a basic neural network from scratch implementing its functionalities such as forward and backward propagation.

I. INTRODUCTION:

Neural networks were created a long time ago. Since then, a lot of frameworks and facilities for developers have emerged to help everyone create what they want. But it is interesting to take a jump back and try to recreate the basics of this architectures.

In this project, I coded a neural network from scratch, based on the principles of forward propagation, backward propagation, and some others.

II. DATASET:

The dataset I used it is probably the most famous deep learning dataset out there. It is the MNIST dataset, which consists of people's handwriting digits from 0 to 9.

Its images have a size of 28 by 28 pixels and are on a gray scale.

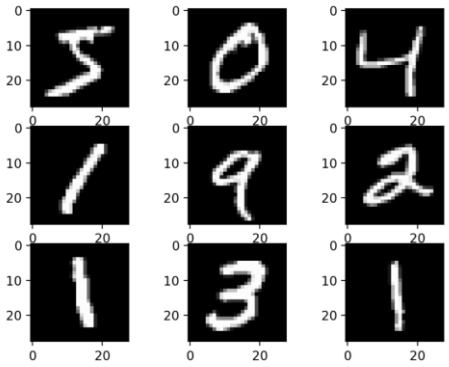

Figure 1. Examples of some samples from the MNIST dataset.

III. WORKING WITH THE DATA:

This dataset is huge. It consists of approximately 60,000 images, but since it is too simple, 2,000 images were enough to reach a good level of accuracy.

The only processes needed to made use of the dataset was to normalize its values (dividing each pixel into the greatest value it could have, being this 255), flattening them (make them shape 1, 784) and changing its type to 'float32'.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(x_train.shape[0], 1, 28*28)
x_train = x_train.astype('float32')
x_train /= 255

y_train = np_utils.to_categorical(y_train)

x_test = x_test.reshape(x_test.shape[0], 1, 28*28)
x_test = x_test.astype('float32')
x_test /= 255

y_test = np_utils.to_categorical(y_test)
```

Figure 2. Rearranging the data.

IV. LITTLE BIT OF THEORY:

All the theory I used was given by both teachers during this subject and intelligent systems. But in addition to that, I used an amazing article written by Omar Aflak [1].

First, I created a base layer, meaning that, a class with the minimum characteristics of a layer was stablished. These minimum characteristics are forward propagation and backward propagation. The first one to give an output to the following layer, and the last one to help propagate an error to create the process of learning.

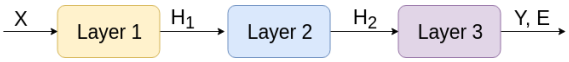

Figure 3. Forward Propagation. [1].

$$\frac{\partial E}{\partial X} \leftarrow \boxed{\text{layer}} \leftarrow \frac{\partial E}{\partial Y}$$

Figure 4. Backward Propagation [1].

After that I need to mention that a lot of layers exist. For example, on a CNN or convolutional neural network, we use Conv2D layers, flatten layers, etc. But in this case, it was only necessary to create two types of them, the fully connected layer (like Dense from Keras) and an activation layer.

a) Fully Connected Layer:

The FCLayer (As I called it on the program) just stands that all the neurons are connected with each other, the most important to know about this is the forward propagation that it requires to work. Which is simple a dot product between the input, the weights and adding the bias:

$$y_j = b_j + \sum_i x_i w_{ij}$$

Figure 5. Forward Propagation for a Fully Connected Layer.

The backward propagation on this particular layer is a more interesting task. Following a set of formulas involving derivatives to calculate the gradient at which should we be heading to reduce our error; they sum up on three important formulas:

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} W^t$$
$$\frac{\partial E}{\partial W} = X^t \frac{\partial E}{\partial Y}$$
$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial Y}$$

Figure 6. Backward propagation formulas.

The first one is used to calculate the error with respect to the input, the second one to calculate de error with respect to the weights and last, the error with respect to the bias. These last two values are multiplied for the learning rate and this creates the process of learning.

The error with respect to the input is a dot product between the error with respect to the output and the weights transposed matrix.

The error with respect to the weights is a dot product between the input transposed matrix and the error with respect to the output.

Finally, the error with respect to the bias is equal to the error with respect to the output.

b) Activation Layer:

As we have seen, a completely linear model does not tend to work okay, that is mainly the reason why we include an activation function, which gives our system the non-linearity characteristic we are looking for.

On forward propagation we just pass the input through an activation function and give that output to the following layer. In this case I implemented both activation functions, which are the sigmoid and the hyperbolic tangent.

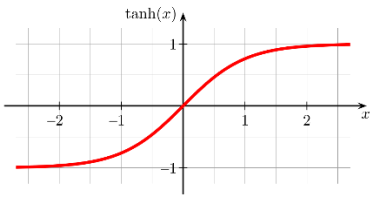

Figure 7. tanh activation function.

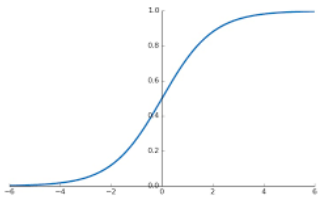

Figure 8. Sigmoid Function.

The backward propagation on this layer is an element-wise multiplication between the error with respect to the output and the input passed through the derivative of the activation decided on the first place.

```
def tanh(x):
    return np.tanh(x)

def tanh_(x):
    return 1-np.tanh(x)**2

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

def sigmoid_(x):
    return np.exp(-1*x)/(1+np.exp(x))**2
```

Figure 9. Activation functions and its derivatives on the program.

Using this model, the hyperbolic tangent performed a little bit faster than the sigmoid function, but mainly because of the backward propagation and how heavier is in comparison the derivative of the sigmoid function with respect to the derivative of the hyperbolic tangent.

V. THE NETWORK

After all these classes are done, the only thing left to do is create a class that help us put all these layers in a list and performs the prediction and the training process, as well as deciding which loss function are we going to be using.

This class is called network, and it consists of a list of layers, a variable for the loss and another one for its derivative.

$$E = \frac{1}{n}\sum_{i}^{n}(y_i^* - y_i)^2$$
$$= \frac{2}{n}(Y - Y^*)$$

Figure 10. Mean-Square Error Formula and its derivative.

Also, it has one function to predict the output of the model, and another one for the training process, that receives the data, the epochs, and the learning rate.

```
network = Network()
network.add(FCLayer(28*28, 100))
network.add(ActivationLayer(tanh, tanh_))
network.add(FCLayer(100,50))
network.add(ActivationLayer(tanh, tanh_))
network.add(FCLayer(50,10))
network.add(ActivationLayer(tanh, tanh_))

network.use(mse, mse_)
network.fit(x_train[0:2000], y_train[0:2000], epochs = 50, lr=0.1)
```

Figure 11. Network used on the program.

VI. RESULTS

As expected, the results were optimal. Without the need of too much training or testing, the loss gets way too low and the model predicts as it should be.

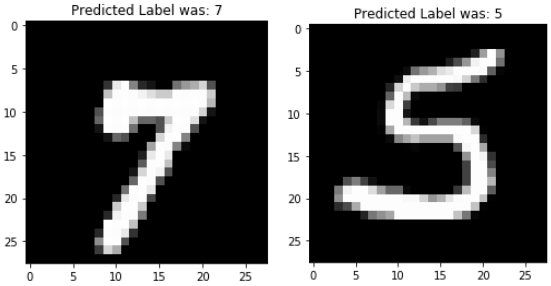

Figure 12. Some predictions of the model.

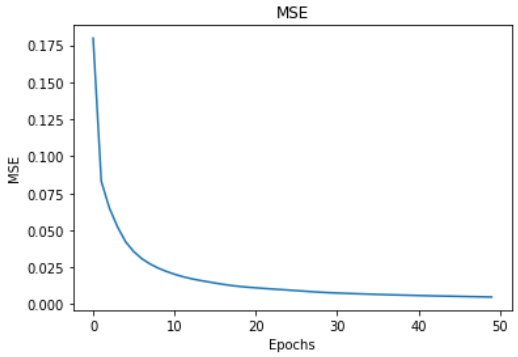

Figure 13. Graph of its MSE.

VII. OWN TESTING

The hardest part of the project was to test it with own handwritten images. The dataset is so meticulous, that an image had to have the perfect characteristics to be read as it

should be. But thanks to Ole Kröger, this was achieved. [2]

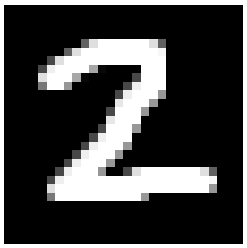

Figure 14. Example of an image drawn by me on paint.

To adapt the image to the required characteristics for a MNIST model, we need to leave the background black and center the image as most as we can, using something called "center of mass" and we want the image to be in a 20x20 pixel box that all the MNIST digits are on.

```
file = "Samples/Own2.png"
test = cv.imread(file, cv.IMREAD_GRAYSCALE)
gray = test

while np.sum(gray[0]) == 0:
    gray = gray[1:]

while np.sum(gray[:,0]) == 0:
    gray = np.delete(gray,0,1)

while np.sum(gray[-1]) == 0:
    gray = gray[:-1]

while np.sum(gray[:,-1]) == 0:
    gray = np.delete(gray,-1,1)

rows,cols = gray.shape

if rows > cols:
    factor = 20.0/rows
    rows = 20
    cols = int(round(cols*factor))
    gray = cv.resize(gray, (cols,rows))
else:
    factor = 20.0/cols
    cols = 20
    rows = int(round(rows*factor))
    gray = cv.resize(gray, (cols, rows))

colsPadding = (int(math.ceil((28-cols)/2.0)),int(math.floor((28-cols)/2.0)))
rowsPadding = (int(math.ceil((28-rows)/2.0)),int(math.floor((28-rows)/2.0)))
gray = np.lib.pad(gray,(rowsPadding,colsPadding),'constant')
```

Figure 15. Code to transform the image

After all this process, an image can be read by the model:

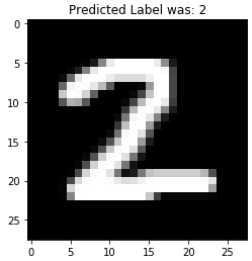

Figure 16. Output of the input in figure 14.

VIII. CONCLUSIONS:

Although the MNIST dataset is simple, building a neural network from scratch can be really entertaining and it has a lot of meaning.

Understanding where everything comes from is really important, is not only about using Keras or PyTorch and not knowing absolutely nothing on what you are doing.

I conclude that this was an amazing project, and even if it looks easy, it had its difficulties.

IX. REFERENCES:

[1]. Aflak, O. (2018). Neural Network from scratch. Towards Data Science. Recovered from: https://towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65

[2]. Kröger, O. (2016). Tensorflow, MNIST and your own handwritten digits. Medium. Recovered from: https://medium.com/@o.kroeger/tensorflow-mnist-and-your-own-handwritten-digits-4d1cd32bbab4