Handwritten Digit Recognition with Neural Network Architectures

Guillermo Creus*, Victor Picón†
Gitlab: *@guillermocreus98, †@vpicon

***Abstract*—Using neural neural networks with the objective of digit recognition, consists in one of the most basic and didactic problems in machine learning. Our study consists in an implementation of different neural network architectures and then comparing results. This article disposes the mathematical foundations behind the programming implementation published at our git repository [1].**

***Index Terms*—Neural Network, digit recognition, MNIST.**

I. Introduction

A. Background and motivation

In the preceding years, machine learning has taken part in every single field it stood a chance. Inspired by biological neural networks, their artificial counterparts have been widely used to perform image recognition tasks. One of its main traits is their ability to perform tasks they were not programmed to do. I.e. by trial and error they adjust their output until it is close enough to the true result.

The project's aims are mainly didactic in the practical aspect. That is, in order to deepen our knowledge in this computer science branch we decided to develop this technology, and document our findings in this article. However, we did not want to follow the easy path by using conventional Python libraries where almost everything is built-in and you only need to fine tune your model. What we want to achieve is a fully operational neural network developed in Python3 and **only** with the help of mathematical libraries, as it is NumPy [2], which adds support for large multi-dimensional arrays and matrices.

To do the mentioned above, the data set used was the widely known MNIST dataset [3]. It consists of 60,000 labeled handwritten images 28 pixels wide and high. As it is black and white, each pixel contains a value ranging from 0 to 783, which will be normalized.

The project does not intend to discuss the theoretical aspect and "learning" capabilities of neural networks, nor comparison between other existent learning methods. The work is done under the assumption of these methods being capable of producing a sufficiently accurate classification tool [4].

The document is structured in four main parts: the development of the architecture and training of an L-layered neural network, then the obtained results and finally a short conclusion. The first one will dive into the mathematics/inner workings of the neural network. The second part will show the reader what the Python code is capable of and how it was done. The last part will attempt to summarize our work, its strengths and mistakes made during the development of the project.

II. L-layered Neural Network

A. Network Architecture and Notation

As a neural network, we are working with L different layers, each containing N_l values, $x_i^{(l)}$ with $1 \leq i \leq N_l$, where we indexed each layer with $1 \leq l \leq L$. And each of these values is obtained as a weighted sum of all the values in the preceding layer plus a bias, passed through an activation function, $h^{(l)}$, which may differ between layers. That is, $\forall j$ such that $1 \leq j \leq N_l$ we have:

$$x_j^{(l)} = h_l\Big(\sum_{i=1}^{N_{l-1}} w_{i,j}^{(l)} x_i^{(l-1)}\Big), \tag{1}$$

where $w_{i,j}^{(l)}$ is the weight of the lth layer, going from $x_i^{(l-1)}$ to $x_j^{(l)}$. Where the bias term is made by forcing the last value of each layer (in exception with the last one) to be always 1, $x_{N_l}^{(l)} = 1$ ($\forall l < L$), and the bias will be held in the corresponding weight $w_{N_l,j}^{(l)}$.

Preceding all these layers, we have an input layer, which we will label with $l = 0$, to keep with the notation above.[1] In our case, our input will be each of the pictures from the data set. Thus, $N_0 = 28 \times 28 + 1$, where the 28×28 term comes from the pixels of the picture, and the $+1$ is for the bias term. The last layer, will be the ouputs layer, that is we want it to classify the input given, into one of the 10 digits available, desirably the correct one! For so, we will have an otput layer with a total of $N_L = 10$ values.

For our data set, suppose now we have M different pictures for our train set [2]. For our given neural network, we will have different values in each layer, for every picture, which we will label as $x_{k,i}^{(l)}$ with $1 \leq k \leq M$. Note though, that the weights of our network, are the same for all M pictures.

For if we are dealing with a train set or a test set, we will have a label associated to each picture in the set, z_k, which will be a vector of nine zeros and a one in the position of the number we want to mark. For example a picture labeled with a 0 will have a label given by $(1, 0, \ldots, 0)$. Also, the output

[1] With this notation for $l = 0$, the formula above for computing the values for the first layer ($l = 1$) remains true, taking values for the weighted sum, from the input layer.

[2] Or take the test-set or any set of pictures, no distinction is important now.

of the neural network for the kth picture, will be notated as $\hat{z}_k$, or $\hat{z}_{k,j}$, with $1 \leq j \leq 10$, for each different digit.

B. Optimization Problem

In order to develop a neural network we decided to define an error function, depending on the weights of our network, which will be optimized over the training data. That is, in order to get the output of the neural network in each picture as close as possible to its label, the error function will increase in those pictures where label and output are very different. And so, in minimizing the error function, with respect to the weights, the weights will be adjusted to match as close as possible the labels of each picture. This reduces our learning problem to an optimization problem,

$$\underset{w}{\operatorname{argmin}} \, E(w), \tag{2}$$

where the error function is given by

$$E(w) = -\sum_k \sum_j z_{k,j} \cdot \log \hat{z}_{k,j}. \tag{3}$$

The error function used above is known as a cross-entropy error. It is commonly used in classification problems where you need to compare two probability distributions. In this case the distributions where $z_{k,j}$, the true label, and $\hat{z}_{k,j}$, the predicted label. By the nature of logarithms, whenever the label $z_{k,j}$ is equal to 1, the optimal predicted label will be $\hat{z}_{k,j} = 1$, bringing the logarithm to zero.

To optimize the error function we chose the gradient descent method, because of its main trait: not being computationally expensive. Considering a constant learning rate the iterative process can be summarized as:

$$w_{n+1} = w_n - \eta \cdot \nabla E(w_n) \tag{4}$$

Lastly, in order to create a probability distribution to predict the picture k_0, condition (5) should always be met. Also, the highest probability will be our way to predict the true label.

$$\sum_j \hat{z}_{k,j} = 1 \qquad \forall k \tag{5}$$

C. Activation functions

The use of activation functions in each layer is one way to achieve non-linearity of neural networks. This feature, which is key to solve the image recognition problem, shouldn't be achieved through carelessness choice of activation functions. The candidates used for this neural networks are:

1) Leaky ReLU: This function is a modification of the classical Rectified Linear Unit. One of the main benefits of this family of functions is the prevention of the vanishing gradient. By being a piece-wise linear function, its derivative is constant, regardless of the value of the function. However, whenever the values are negative the derivative is 0, preventing the update of the network's weights.

To solve this, the leaky ReLU comes in place:

$$ReLU_L(x) = \begin{cases} x & x \geq 0 \\ \frac{x}{100} & x < 0 \end{cases}$$

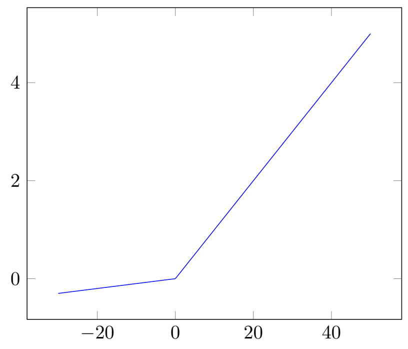

Fig. 1: Leaky ReLU

It inherits all the benefits of the ReLU function and tries to solve the 0 derivative by introducing a milder slope (when $x < 0$). Therefore, this will be the function used through the network layer excluding the Lth layer.

2) Softmax: This was the function used in the output layer. It can be defined as:

$$\sigma : \mathbb{R}^{10} \to \mathbb{R}^{10}$$
$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_n e^{z_n}}$$

It provides the last layer with an output that ranges between 0 and 1 and the sum adds up to 1. As stated before, the main goal of the neural network is to shift the weights so the $\sigma(\mathbf{z})_{label[k]}$ is as close to 1 as possible.

D. Gradient calculation and Back-propagation

As seen in (4), we need to compute the gradient of the error function, with respect to the weights, in each iteration. To do so, a famous algorithm called back-propagation, is used here to compute each derivative. [3]

We see by our definition of the error function (3), that we can compute the error, as a sum of errors of each picture. That

[3]A deep study can be made to choose a proper, or even optimal. However, as it is typically done in the field, a sufficiently small learning rate is taken assuring the non-divergence of the function.

is:

$$E(w) = \sum_k E_k(w), \text{ with } E_k(w) = -\sum_j z_{k,j} \cdot \log \hat{z}_{k,j}.$$

First we define the output of the weighted sum of each value in the layer as:

$$a_{k,j}^{(l)} := \sum_i x_{k,i}^{(l-1)} \cdot w_{i,j}^{(l)} \tag{6}$$

Hence, we trivially have that $x_{k,j}^{(l)} = h_l(a_{k,j}^{(l)})$. Now, using the chain rule we may write the partial derivative of $E_k(w)$ with respect to the weight $w_{i,j}^{(l)}$ as

$$\frac{\partial E_k}{\partial w_{i,j}^{(l)}} = \sum_k \frac{\partial E_k}{\partial a_{k,j}^{(l)}} \cdot \frac{\partial a_{k,j}^{(l)}}{\partial w_{i,j}^{(l)}} \tag{7}$$

Looking at (6), we get that the second term inside the summation is given by:

$$\frac{\partial a_{k,j}^{(l)}}{\partial w_{i,j}^{(l)}} = x_{k,i}^{(l-1)}$$

And defining the first term as:

$$\delta_{k,j}^{(l)} := \frac{\partial E_k}{\partial a_{k,j}^{(l)}}$$

Putting it all together, we can rewrite equation (7) as:

$$\frac{\partial E_k}{\partial w_{i,j}^{(l)}} = \sum_k \delta_{k,j}^{(l)} x_{k,i}^{(l-1)} \tag{8}$$

For the last layer L, and using our error function (3), we can compute the value of $\delta_{k,j}^{(L)}$ as

$$\begin{aligned}\delta_{k,j}^{(L)} &= \frac{\partial}{\partial a_{k,j}^{(L)}} \Big\{ \sum_t -z_{k,j} \cdot \log\left(\frac{e^{a_{k,t}^{(L)}}}{\sum_n e^{a_{k,n}^{(L)}}}\right) \Big\} \\ &= \sum_{t \neq j} z_{k,t} \cdot \frac{e^{a_{k,j}^{(L)}}}{\sum_n e^{a_{k,n}^{(L)}}} - z_{k,j} \cdot \left(1 - \frac{e^{a_{k,j}^{(L)}}}{\sum_n e^{a_{k,n}}}\right) \\ &= \frac{e^{a_{k,j}^{(L)}}}{\sum_n e^{a_{k,n}^{(L)}}} \cdot \underbrace{\sum_t z_{k,t}}_{1} - z_{k,j} = \widehat{z}_{k,j} - z_{k,j}.\end{aligned} \tag{9}$$

Where we have used the fact that for any picture k, the label vector is such that it has only one element different from zero, and it is a 1, thus, $\sum_j z_{k,j} = 1$.

From the calculation of above for the last layer, the back-propagation algorithm allows us to compute the remaining $\delta_{k,j}^{(l)}$ for any layer $l < L$; it is done as follows. In the definition of $\delta_{k,j}^{(l)}$, using the chain rule again, we have that:

$$\delta_{k,j}^{(l)} = \sum_t \frac{\partial E_k}{\partial a_{k,t}^{(l+1)}} \cdot \frac{\partial a_{k,t}^{(l+1)}}{\partial a_{k,j}^{(l)}} = \sum_t \delta_{k,t}^{(l+1)} \frac{\partial a_{k,t}^{(l+1)}}{\partial a_{k,j}^{(l)}},$$

and for the last term we have:

$$\frac{\partial a_{k,t}^{(l+1)}}{\partial a_{k,j}^{(l)}} = w_{j,t}^{(l+1)} \cdot h_l'(a_{k,j}^{(l)}),$$

which putting it in the above formula, leads us to the following result:

$$\delta_{k,j}^{(l)} = h_l'(a_{k,j}^{(l)}) \cdot \sum_t \delta_{k,t}^{(l+1)} \cdot w_{j,t}^{(l+1)} \tag{10}$$

which is known as the backpropagation formula.

E. Implementation

This section is a brief review on the structure of the presented code in the repository. The `main.py` file contains the main contents of the project. The neural network itself has been structured into a class called `NeuralNetwork` which itself contains an array with classes called `Layers` and methods that range from updating the parameters of its layers to print errors and precision. Doing that helps to keep the code centralized.

To improve efficiency and run-time, a matrix structure was adopted in most of the code, where huge sums were performed, and keeping everything in these arrays was helpful and efficient, due to BLA BLA Numpy.

III. Results and Conclusion

In this paper we have shown the ability to recognize handwritten digits with neural networks. Within a two hour time span a whole network can be trained to recognize handwritten digits with 95% precision. Diving deep into the architecture has shown us the huge role played by hyper-parameters and activation functions.

The hyper-parameters have been key to develop a working neural network. By changing them, we were able to avoid amplification of the output layer. In other words, we had a factors' vector which corrected the output of the layers by multiplying the input of the activation function by a factor that varies in every layer.

Another area that should be followed closely is the choice of activation functions. In the first stages of the project we used the sigmoid function in the input and hidden layers. At first glance, this seems like a great idea, but the vanishing gradient is a huge problem that hinders the network's learning process. We carefully chose a function where in the extremes, the derivative was not close to zero. That is how the leaky ReLU was found and it improved the precision by double digits.

It is of great concern of the simplicity of these type of neural networks. In the past years, more sophisticated models and even applied to image recognition, have proved to have a better performance. Having fewer weights and avoiding the fully connected network would help to extract features in more adverse conditions (images not centered, etc). More advanced methods are available nowadays, and even better methods will be available in the coming years.

References

[1] *Gitlab Repository*,
https://gitlab.com/guillermocreus98/digit-recognition

[2] *NumPy official webpage*,
https://numpy.org/

[3] *MNIST Dataset*,
http://yann.lecun.com/exdb/mnist/

[4] Shai Shalev-Shwartz, Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*, Cambridge University Press, 2014.