
Neural Network for MNIST database

Haegel Adrien - 20154913

December 17, 2015

1 INTRODUCTION

For this project, I wanted to deeply understand how neural networks worked. I therefore decided to implement my own neural network.

The main issue with neural networks being the computation time, I used the MNIST database, which is a database for handwritten numbers recognition. Since there are only 10 characters to recognize, the computation time is achievable on a personal computer.

I wrote a code in C#, and an executable version of this code is given with this report. It is possible to launch and optimize a neural network with wanted parameters. A detailed explanation of this program will be given later.

I followed the online book from Michael Nielsen, which is a very clear and detailed introduction to neural networks. (This online course can be found at this adress: <http://neuralnetworksanddeeplearning.com/>)

However, I have written all the code entirely by myself and from scratch.

In this report, I first explain the theory behind neural networks, then I introduce technical and computational issues I encountered, and I will present the results of my algorithm. In the last part there are instructions to run my code if needed.

2 NEURAL NETWORKS : HOW DOES IT WORK?

The idea behind neural networks is to simulate the biological brain by imitating neurons. A neuron has a quite simple mechanism: Several axons send a "potential" to a neuron, and the neuron "fires" depending on this entry potential. By connecting a lot of neurons, it is possible to obtain very complex functions. The difficulty is to "teach" a neural network on a dataset, since the number of variables makes it complex and computationally expensive.

2.1 THE SIGMOID FUNCTION

A neuron is modelled as shown in the figure 2.1.

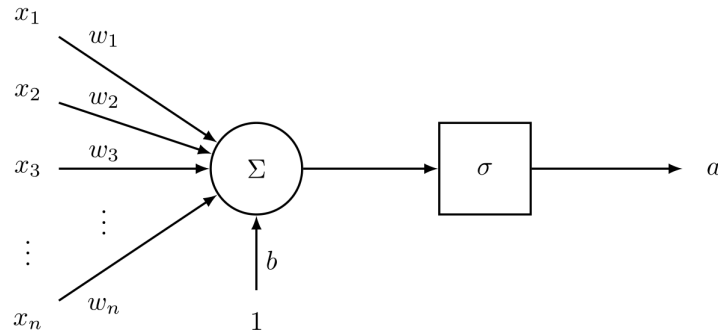


Figure 2.1: A neuron using the sigmoid activation function

- x_1, x_2, \dots, x_n are the output of the previous neurons
- w_1, w_2, \dots, w_n are the weights corresponding to the previous neurons
- b is the bias of this neuron. It represents if the neuron "fires" easily.
- sigma is the sigmoid function, $\sigma(x) = \frac{1}{1+e^{-x}}$
- The activation a for this neuron is given by:

$$a = \sigma(b + \sum_{i=1}^n x_i w_i) \quad (2.1)$$

The sigmoid function is plotted in figure 2.2

2.2 THE NEURON LAYERS

Our network consist of several neuron layers. The first layer is the input layer, where each neuron has the activation corresponding to a pixel input. (Since MNIST images are 28*28 images, our first layer has 784 neurons.) The last layer is the output layer, and the "expected" output if the image classification is 2 for example, is $[0,1,0,\dots,0]$.

Figure 2.3 shows the notations used in our network.

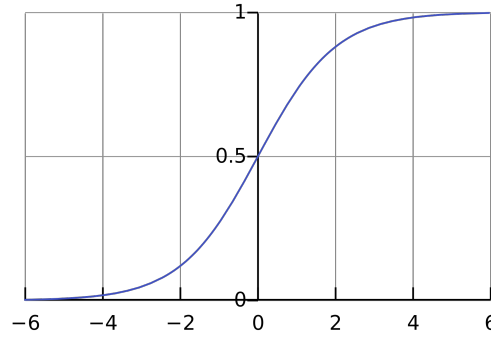


Figure 2.2: Sigmoid function σ

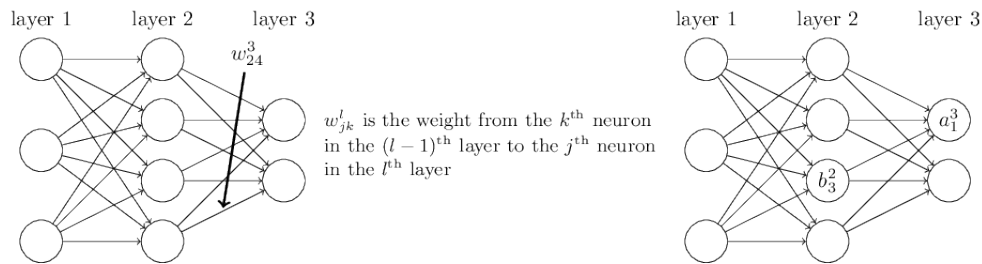


Figure 2.3: Notation for our neural network

2.3 HOW THE LEARNING ALGORITHM WORKS

To optimize our network, we use a stochastic gradient descent algorithm:

- We take a subsample of our training data set. For every subsample in the dataset:
- We propagate the samples in the network, and look at the result on the last layer.
- We compute the error on these samples, and backpropagate to compute the corrections that need to be done on all the weights and biases.

Everytime we go through the entire dataset, we call that an epoch.

The different parameters are the following:

- The neural network structure: number of layers, and number of neurons for each layer.
- η : The learning rate. Regulates the amount of correction performed on the network
- λ : a regularization parameter. If 0: no regularization, if high: all the weights have a tendency to go to 0.
- The batchsize m: the number of items in each subsample.

2.4 THE PROPAGATION

We start by giving to the input layer the activation corresponding to the pixel value of an input image. We then propagate this activation through the network by using the formula 2.1, which gives the corresponding matrix equation:

$$a^l = \sigma(w^l a^{l-1} + b^l) = \sigma(z^l) \quad (2.2)$$

This formula is therefore applied for $l = 2, 3, \dots, L$

The predicted output is the neuron in the last layer with the highest activation.

2.5 THE BACKPROPAGATION

The idea behind the back-propagation is to minimize a cost function, like:

- Quadratic cost : $\sum_x \|y(x) - a^L(x)\|^2$ where x is an image, $y(x)$ the correct prediction, and $a^L(x)$ the activation output of the last layer
- The cross-entropy function: $-\sum_x \sum_j y_j \ln(a_j^L) + (1 - y_j)(1 - \ln(a_j^L))$

We implemented both cost function, but we will only consider the cross-entropy in this report.

The equations for the backpropagation derive from the following objective:

We want to compute $\frac{\partial C}{\partial w_{i,j}^l}$ and $\frac{\partial C}{\partial b_i^l}$. We won't be detailing the calculations to obtain the back-propagation formula, but these formulas are quite easily obtained by using the chain rule and using recurrence from the last layer to the first one.

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad (2.3)$$

where \odot is the term by term product.

For the cross-entropy function, this leads to:

$$\delta^L = a^L - y \quad (2.4)$$

The propagation equation for delta is the following:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad (2.5)$$

And the derivatives of the cost function regarding the weights are:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad (2.6)$$

$$\frac{\partial C}{\partial w_{i,j}^l} = a_j^{l-1} \delta_i^l \quad (2.7)$$

This leads us to the following formula to numerically update the weights and biases:

$$w^l = w^l (1 - \frac{\eta \lambda}{n}) - \frac{\eta}{m} \sum_x \delta_x^l (a_x^{l-1})^T \quad (2.8)$$

$$b^l = b^l - \frac{\eta}{m} \sum_x \delta_x^l \quad (2.9)$$

A regularization parameter, $\frac{\eta \lambda}{n}$ has been added. This is useful to avoid that weights get stuck with a too high value.

2.6 FROM THEORY TO PRACTICE

We now have everything we need to write the code for our neural network.

To sum up: We take a picture from the data set, propagate it, measure the error, backpropagate the error, and update the variables of our neural network.

In the next part, I will describe the data sets and the method to avoid overfitting. Then, I will discuss some computational issues, and lastly I will cover the automated learning method.

3 IMPLEMENTATION

3.1 THE MNIST DATABASE

The MNIST database consist of 70.000 black and white pictures, 28*28 pixels, associated with the image label. A sample of this database is shown in figure 3.1.



Figure 3.1: Samples from MNIST database

In order to measure accurately the performance of our neural network, we want to avoid using the same samples to learn the weights and to measure the performance.

Therefore, we keep the last 10.000 images as test data, and 10.000 others as validation data. Validation data will be used to calibrate the parameters of our network and also to monitor the convergence. By doing that, we can reduce overfitting. Overfitting happens when the algorithm becomes specialized on a particular data set, but loses general accuracy for other (unknown) data sets.

We therefore have 50.000 images for training.

3.2 COMPUTATIONAL ISSUES

I tested a first implementation, using the very high level properties of the C# language. Every neuron and every axon (link between neurons) was represented by an independant object. I had the hope that this implementation would be really easy to modify, since it is possible to handle the objects one by one. However, the main flaw of this architecture was the computational time. Indeed, accessing independant objects is quite slow, compared to a much more compact "matricial" implementation. Especially in this problem, where it is possible to express the entire network under matrices, and to only have matricial operations.

Therefore, I have rewritten my entire code, using matrices. As expected, the algorithm went around 10 times faster. I found it interesting to see the performance that can be obtained by using a better algorithm architecture.

However, I was still not satisfied with the speed, and it seemed to me that the matrix class I was using (Mathnet Numerics) was not quick enough. Since it is not possible to access the functions, it is hard to do optimizations on it. But the longest operation in my network was the computation of a rank 1 matrix (multiplication of 2 vectors that give a matrix). I could only use the general matrix multiplication, that must not be optimized for this kind of computation.

Therefore, I rewrote my own linear algebra class. Therefore, I could implement my own method for the backpropagation, where I could write directly into the matrix instead of using heavy methods that were creating new matrices on the way.

With that, I reduced the time for one learning cycle from 9'15" to 3'15".

Then, I added some parallelization in this matrix class (one thread to compute each row). That way, I was using all my CPU power, and that brought the time down to 1'50".

The conclusion is that it is possible to divide the computation time by 5 by doing more clever basic operations.

I tried to use CUDA to reduce this time even more, but I got stuck with technical issues. I would be really interested to know the performance gap achievable using the GPU power.

3.3 THE AUTOMATED LEARNING METHOD

How to tune the learning rate parameter? Ideally, it is better to have a large value at the beginning and a lower one at the end, to do more precise optimisations. Furthermore, we want to reduce the number of parameters to tune, so doing an automated learning rate would be a nice way to do so.

In order to do that, I use the validation set, and measure the accuracy at each learning period. If the accuracy is not improving, we reduce the learning rate. If there is no more improvement after having reduced the learning rate several times, we end the algorithm.

In the next part, I will show the results of the parameter optimization, and then the results of my algorithm.

4 RESULTS

4.1 PARAMETER OPTIMIZATION

Until now, my parameters were chosen arbitrarily. We need to optimize them. For that purpose, I computed the accuracy on the validation data when the parameters are in a given range. The results are given in the following tabulars. The two parameters to optimize are the regularization parameter λ and the batchsize.

I did that on a simple network (784 - 100 - 10 neurons) and a bigger one (784,250,100,50,10).

	batchsize			
lambda	5	10	25	50
1	2,08	1,88	2,36	2,98
2,5	2,12	1,88	2,17	2,45
5	2,07	1,9	2,01	2,41
10	2,54	2,09	2,28	2,43

Figure 4.1: Error percentage on the validation data - 784/100/10 neurons

	batchsize			
lambda	5	10	25	50
1	1,89	1,68	1,54	1,77
2,5	1,53	1,48	1,66	1,53
5	1,52	1,53	1,72	1,65
10	1,6	1,73	1,7	1,77

Figure 4.2: Error percentage on the validation data - 784/250/100/50/10 neurons

From these results, I concluded that the optimal parameters were $\lambda = 2.5$ with a batchsize of size 10.

4.2 CONVERGENCE IN TIME

How good is our learning algorithm? The following figure shows the accuracy of our neural networks with the learning epochs. The learning rate is also displayed, and it is possible to see that the learning goes down and allows the neural network to converge to its final solution. Since the accuracy stays flat for different learning rate, we can assume our network reached his optimal accuracy.

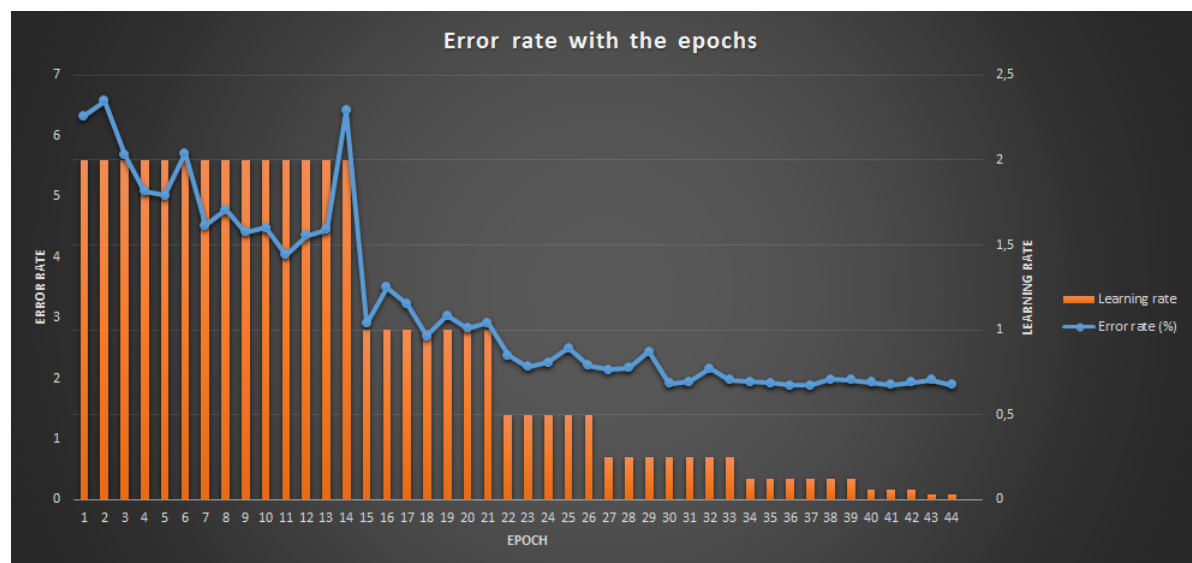


Figure 4.3: Error and learning rate along the epochs

The next tabular shows the time needed to reach final accuracy:

Network type	Average time
784-100-10	17 minutes
784-250-100-50-10	50 minutes
784-600-300-150-50-10	5 hours

We can see that on deeper networks, the computation time can become prohibitive.

4.3 FINAL RESULTS

These results are obtained on the test data (which is completely fresh to my algorithm). I used the parameters estimated earlier.

Network type	Final accuracy
784-100-10	1.96
784-250-100-50-10	1.44
784-600-300-150-50-10	1.45

I therefore reached 98.56 percent accuracy on the MNIST database. We can see that on deeper networks, the result is actually not improving. This seems to be a limitation on our neural network architecture.

The following figures show images from the MNIST dataset, and the guess of my neural network:

4.4 HOW TO IMPROVE THESE RESULTS

This neural network reaches, with this 1.44 percent error rate, the performance that was obtained in 2003 (the following website shows the MNIST scores for different algorithms across time: <http://yann.lecun.com/exdb/mnist/>). The best accuracy for now with standard neural nets is 0.83 percent, and 0.23 percent with convolutional neural nets.

The following strategies could improve the results presented in this project:

- Use GPU to reduce computation time and hence be able to train on deeper networks.
- Switch to convolutional networks, which have a better accuracy than standard neural networks.































Guess	images		
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			

Figure 4.4: Examples correctly classified































Guess	images		
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			

Figure 4.5: Examples incorrectly classified

5 HOW TO USE MY CODE

5.1 EASY EXECUTABLE

I made an easy to use command line code, where you just have to go into "EasyToLaunch" folder in my report file, and then use "NeuralNetProject.exe".

You can run the default parameters or choose to use your own parameters in the command line.

At the end of the algorithm, two new folders are created, one containing the logs of the algorithm, and the other containing the images. There is a correct folder (where the prediction was accurate) and an error folder. The first number in each picture's name correspond to the prediction.

5.2 SEE THE SOURCE CODE

In the project folder, go to "Source code/NeuralNetProject" and open the solution using Microsoft Visual Studio. Please only compile with the "Debug" mode with the option "any cpu".

The logs and result images will be displayed in bin/debug.