

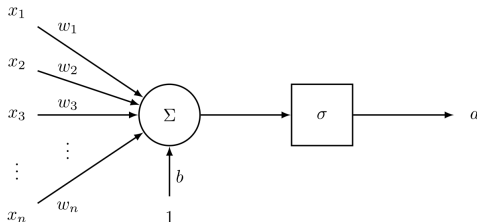
Neural Network for MNIST database

Haegel Adrien - 20154913

December 17, 2015

- My goal: I wanted to deeply understand how neural networks worked. I therefore decided to implement my own neural network.
- Main issue : The computation time. Hence, I used the MNIST database (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#, from scratch. I did not want to use external code, even if it would have yielded better accuracy.
- I followed the online book from Michael Nielsen. It is really clear and interesting, I highly recommend to read it if you are interested in this subject:
<http://neuralnetworksanddeeplearning.com>

- 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.



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

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

The sigmoid function

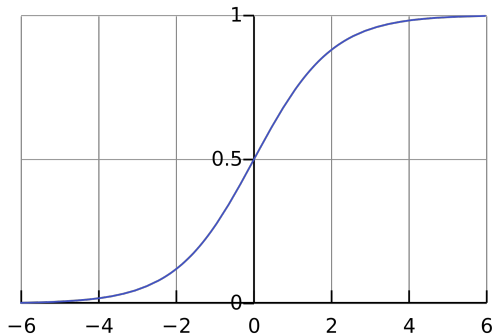


Figure: Sigmoid function σ

- Network consist of several neuron layers. No cycles and no axons going backwards.
- The first layer is the input layer : Each neuron is a pixel of the data image (784 neurons)
- The last layer is the output layer, the result is the highest-activated neuron

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

- We take a subsample of our training data set, this is called a batch. For every batch 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.

- Take an image, inject in the first layer
- Propagate this activation through the network using

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

- Do that for every layer
- The predicted output is the neuron in the last layer with the highest activation.

- Idea : minimize a cost function
- Quadratic cost between predicted and expected values
- By derivating this cost function with regards to the weights and biases of our network, we find the backpropagation formulas
- $\delta^L = a^L - y$
- $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$

- How to update the weights?
- $\frac{\partial C}{\partial b_j^l} = \delta_j^l$
- $\frac{\partial C}{\partial w_{i,j}^l} = a_j^{l-1} \delta_i^l$
- We simply use gradient descent!
- $w^l = w^l(1 - \frac{\eta\lambda}{n}) - \frac{\eta}{m} \sum_x \delta_x^l (a_x^{l-1})^T$
- $b^l = b^l - \frac{\eta}{m} \sum_x \delta_x^l$
- A regularization parameter, $\frac{\eta\lambda}{n}$ has been added

70.000 black and white pictures, 28*28 pixels with label

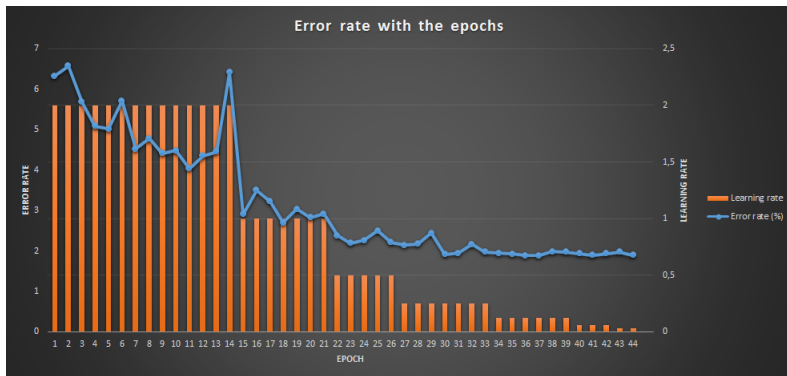


To avoid overfitting, we keep 10.000 images as validation (to calibrate the parameters of our network and also to monitor the convergence)

The last 10.000 images are used as test data to measure the performance of the algorithm.

- First implementation, very high level properties of the C# language. Was way too slow because of many memory accesses (which were not contiguous)
- Rewrote everything using more matrices, 10 times faster
- The matrix class I used (Mathnet numerics) seemed not optimized for the kind of calculations needed: I rewrote my own linear algebra class, 3 times faster.
- Used CPU parallelization, 2 times faster
- Conclusion: Trying to understand what is happening on low computational levels can increase the speed a lot! (60x)

- The learning rate should not be a constant, so I used the validation data to monitor it:
- If there is no improvement, we reduce it
- If there is no more improvements, we stop



- Until now, my parameters were chosen arbitrarily.
- To optimize them, I computed the accuracy on the validation data for different values, on different networks

	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

Network type	Final accuracy (Error %)	Time needed
784-100-10	1.96	17 minutes
784-250-100-50-10	1.44	50 minutes
784-600-300-150-50-10	1.45	5 hours

- Obtained on (fresh) test data
- The biggest network gave disappointing results. Limitation on the neural architecture?































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

Figure: Examples correctly classified































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

Figure: Examples incorrectly classified

- Best accuracy today: neural nets is 0.83 percent, and 0.23 percent with convolutional neural nets. (I have 1.44)
- Use GPU to reduce computation time and hence be able to train on deeper networks.
- Switch to convolutional networks

Thank you!