Neural Networks and Computational Intelligence

Assignment 1: Perceptron Training

Jeroen Overschie, Erik Zhivkoplias

Group 01

January 31, 2020

1 Introduction

In learning about multi-layer Neural Networks, a good way to start is to learn first about a single-layer neural network, the Perceptron. The Perceptron algorithm was invented by Frank Rosenblatt, therefore coining the full term for the procedure Rosenblatt perceptron algorithm. In this assignment, students are asked to implement a version of this algorithm themselves, and observe its performance and behavior. Then, a comparison should be made to the theoretical behavior of the algorithm under circumstances of various parameter values as defined in the assignment. Hereby, the storage capacity of the perceptron is tested, using computer simulations.

1.1 Problem

Given dataset D_N^P with input vector $\xi^\mu \in R^N$ and binary labels $S^\mu \in \{-1, +1\}$, where $\mu = 1, 2 \ldots, P$ we want to find a vector $w \in R^N$ such that $sign(w * \xi^\mu) = S^\mu$ for all μ.

We consider dataset to be linearly separable (l.s.) if in N-dimensional space such a N-1 hyperplane exists that can separate all examples belonging to class +1 from all examples belonging to the class -1.

In the present work we want to implement the Rosenblatt perceptron algorithm and find out how good the result of computer simulations while training perceptron on D_N^P with different α values will fit theoretical learning capacity of the perceptron.

2 Method

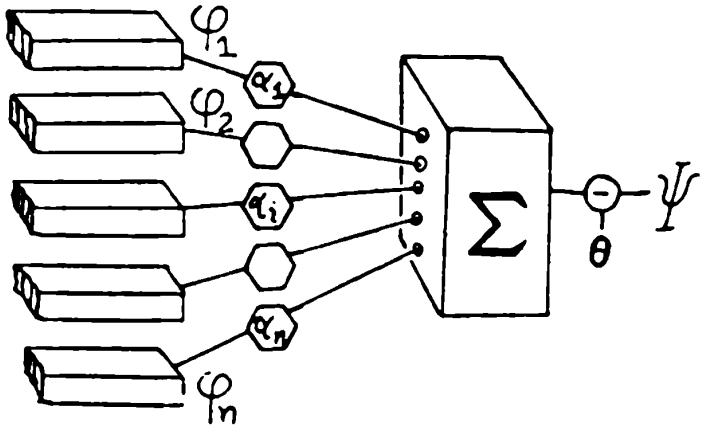

Figure 1: The computational graph of the Perceptron (Minsky-Papert, 1969).

2.1 The concept of algorithm

The Rosenblatt perceptron algorithm is a feed-forward network that given the dichotomy of data can decide whether or not ξ^{μ} belongs to class S (e.g, class +1 or class -1). Data is defined as D_N^P; where N is a number of features and P is a length of the input vector $\xi_P = [x_1 \dots x_P]$. Input values are multiplied by their respective weights $w = [w_1 \dots w_n]$. If the weighted sum

$$\frac{1}{N}\sum_{\imath=1}^{N} w * \xi \tag{1}$$

is greater than the specified threshold θ (here $\theta = 0$) then the example belongs to class +1, if less or equal to θ the example belongs to class -1. Therefore the algorithm converges and yields the sought weight vector w if:

$$S^{\mu} = sign((w * \xi) - \theta) = \pm 1; \tag{2}$$

The weight vector w is by definition always orthogonal to the hyperplane that separates two classes.

2.2 Theoretical learning capacity of the Perceptron

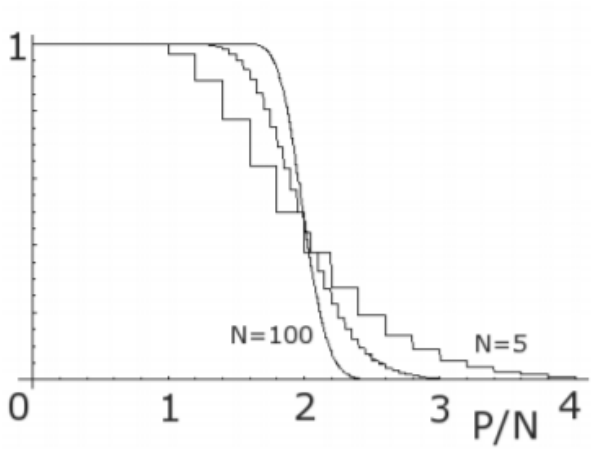

Figure 2: Learning capacity of Perceptron w.r.t. N (Minsky-Papert, 1969).

From the lecture material we know that the storage capacity of perceptron (the number C (P, N) of homogeneously l.s. dichotomies) is the function of α; $\alpha = \frac{P}{N}$. We also learned that the algorithm demonstrates certain behaviour when $N \to \infty$

$$Pl.s.(\alpha) = \begin{cases} 1, & \text{if } \alpha \leq 2. \\ 0, & \text{if } \alpha \geq 2. \end{cases} \tag{3}$$

However, it is required that the dataset $D_N^P = \{\xi^\mu \in R^N\}$ of P vectors in N dimensions must be in general position which means that all subsets of dataset with $\tilde{P} \leq N$ must be linearly independent. Here we meet that requirement applying the Rosenblatt perceptron algorithm onto a randomized dataset where all of the subsets of vector ξ are always in general position: a randomized dataset of Gaussian components $\xi_j^\mu \sim \mathcal{N}(0, 1)$.

2.3 Implementation

The implementation of the Rosenblatt perceptron was written in Python.

Perceptron computations are wrapped inside a *perceptron* function, with the following input parameters: parameter α that relates N and P, number of features N and maximum number of epochs $nmax$. The *perceptron* function operates as follows:

1. Generate a random dataset with N features using *np.random.normal* with $\xi_j^\mu \sim \mathcal{N}(0, 1)$ to draw from a standard Gaussian distribution. Also, generate a label accompanying the feature vector; a random number of either -1 or 1. In total P such samples are generated.
2. Initialize weights vector with length N all at 0. Then, enter the epoch loop; loop until the Perceptron either converged or we looped a maximum number of $nmax$ times.
3. For every epoch: initialize a *scores* vector of length P all at 0, one for every example. Then, loop all data samples (loop of length P).
4. For every example: take the dot product of the designated feature vector $\xi^{\mu(t)}$ with the weights vector, multiply this by the accompanying label $S^{\mu(t)}$. Set the scores vector at position $\mu(t)$ to 1 when the previous product > 0, else the score is set to 0. If the computed product is less or equal than 0, however, the product of *alpha*, the feature vector of current data sample, and the current data sample label are added to the weights vector.
5. Update weight vector; $w(t+1) = w(t) + \frac{1}{N} * \xi^{\mu(t)} * S^{\mu(t)}$
6. Exiting the data example loop, the sum of scores of all samples is computed. If the algorithm got all samples correct, the Epoch loop is exited early- and the algorithm returns a successful convergence using the given input parameters.
7. In case no convergence was accomplished within the maximally allowed amount of epochs $nmax$, the algorithm returns unsuccessful convergence.

Then, in order to execute this sequence of steps multiple times with various different parameters, a *starmap* function is used. Such a function is available in the *itertools* module, but also- in the *multiprocessing* module. Since we make use of multiprocessing, a specialized *starmap* function is used for this. More about this in the next section.

2.3.1 Performance concerns

In running the perceptron with higher parameter values, the CPU-workload gets higher quickly. This in part due to the Curse of dimensionality (Keogh and Mueen, 2010). Therefore, it is desirable to optimize perceptron execution, in order to be able to execute the perceptron with higher parameter values.

To optimize performance, a library that enables utilizing more CPU cores is used. This is the *multiprocessing* python package. This packages enables users to create a "pool", with a certain amount of CPU cores to utilize, n. By default, n is the maximally available CPU cores on the designated machine. In this experiment, default parameters are used.

In order to switch between single-core and multi-core execution, a command line flag is revised *multicore*. Only when executing the program with this flag present, will the perceptron be executed in parallel. This also enables one to keep the ability to debug in VSCode- the multiprocessing python package is not compatible with the VSCode debugger- and one must thus execute on a single-core when debugging. The implementation for the switching procedure is as follows.

```
multicore = np.size(sys.argv) > 1 and (
    sys.argv[1] == '--multicore')
provider = Pool() if multicore else itertools
```

Listing 1: Code used deciding between single- or multicore execution.

The part of the program that is parallelized is the perceptron execution. Multiple perceptron are spawned simultaneously and are executed in parallel. This is implemented as such.

```
resvec = list(provider.starmap(
    perceptron,
    itertools.product(alphas, np.full(nd, N), [ nmax ])
))
```

Listing 2: Code used for parallelizing the perceptron execution.

3 Results and Discussion

3.1 Experiment 1

For the first experiment, the parameters were set to $N = 20; nD = 100; nmax = 100, \alpha = \{0.75, 0.772, 0.795, \ldots, 3.0\}$

As we can see the perceptron has found all correct weight vectors for all datasets when $\alpha \leq 1$, and the ratio of correctly classified datasets is between 0 and 1 when $1 < \alpha < 3$. Therefore when the number of features N (in this case $= 20$) is $\leq$ the number of examples P the algorithm will always be able to find the solution. Now let's explore what happens if we increase N.

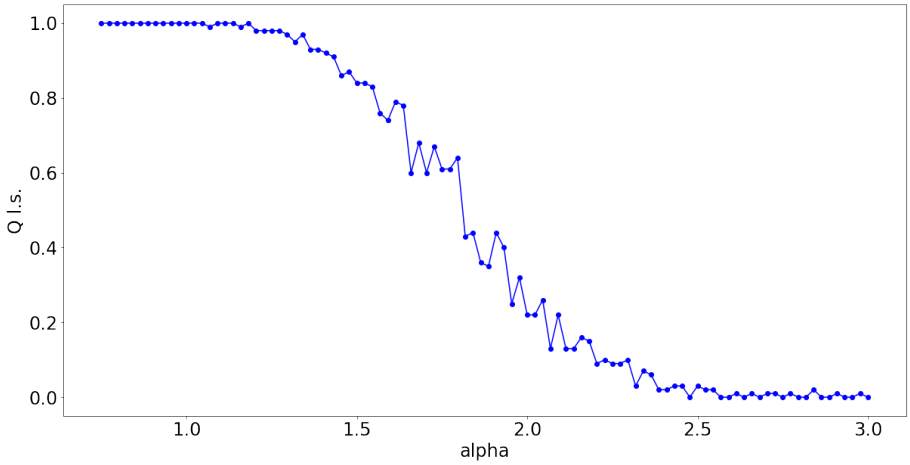

Figure 3: Q l.s.(alpha) function for N=20

3.2 Experiment 2

For the second experiment, we set $N = 200; nD = 200; nmax = \{100, 500\}$; $\alpha = \{0.75, 0.772, 0.794, \ldots, 3.0\}$;

From Figure 4 we could make a couple of interesting observations. Firstly, when N gets larger, the Q l.s.(α) function turns into the step function with more limited range of α values where the ratio of correctly classified datasets is between 0 and 1.

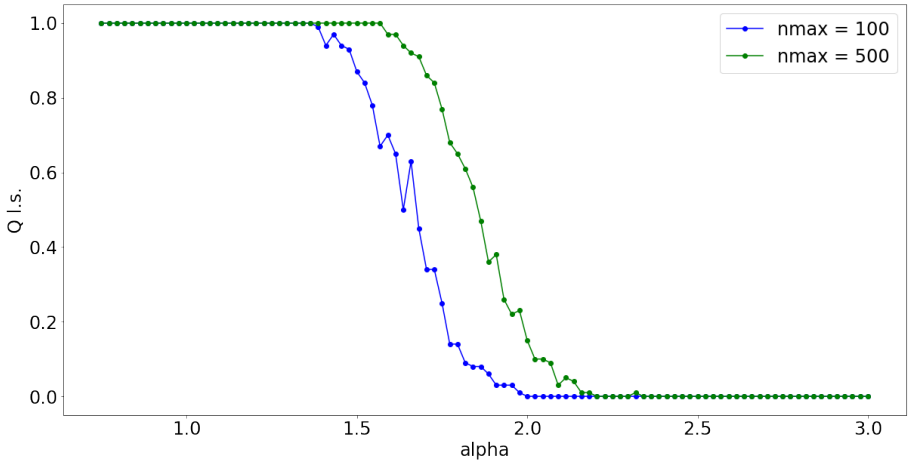

Figure 4: Q l.s.(α) function with high parameters (N = 100, nd = {100, 500})

Secondly, we know that every dataset in the set of D_N^P datasets is l.s. However, as we can see the transition region is shifted towards lower values of α in comparison the theoretical learning capacity of the perceptron as shown in Figure 2.

It can be explained in the following way: if we think about $nmax$ as of number of trials we take to correctly solve the same dataset with the same distribution of two binary classes it seems that the perceptron can find the solution only when the number of trials is large enough as for some datasets it takes more time to find the solution. Therefore, when $nmax$ is increased to 500, more randomized datasets of Gaussian components can be l.s. with the perceptron as the required number of steps is finite, but can definitely be larger than 100. Therefore, we can agree that if $nmax \to \infty$ the transition region won't be shifted towards lower values of α anymore.

Finally, let's explore the behavior of Q l.s.(α) function for different system sizes N leaving all other parameters the same.

3.3 Bonus part. Experiment 3

For the third experiment, we set $N = \{5, 20, 100\}; nD = 300; nmax = 300$; and $\alpha = \{0.75, 0.772, 0.794, \ldots, 3.0\}$;

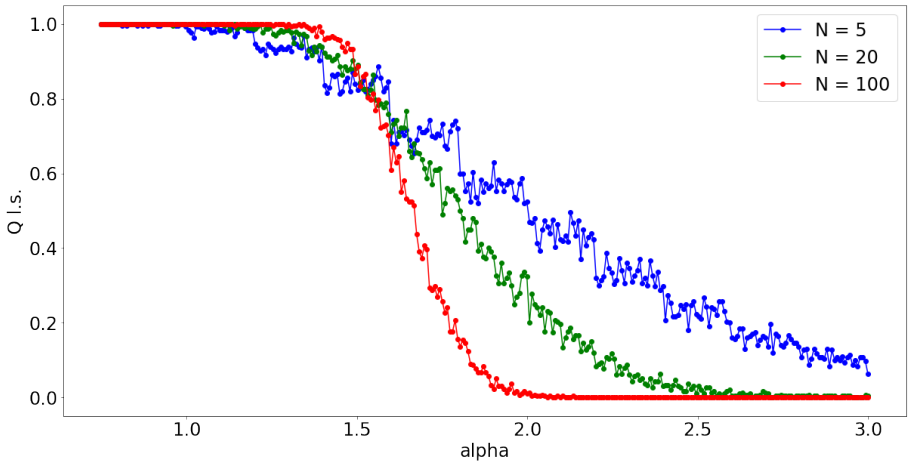

Figure 5: Q l.s.(alpha) function for different N (N=5, N=20, N=100).

Again, as we can see when N gets larger, the Pl.s.(α) function turns into the step function with the limited range of α values where the ratio of correctly classified datasets is between 0 and 1.

Our results is consistent with the theory, as we can always derive the fraction of linearly separable dichotomies for given α from:

$$Pl.s.(\alpha) = \begin{cases} 1, & \text{if } \alpha \leq 1. \\ 2^{1-P} \sum_{i=0}^{N-1} \binom{P-1}{i}, & \text{if } \alpha \geq 1. \end{cases} \tag{4}$$

Therefore, the fraction indeed is not the same for different sets of α values.

4 Conclusion

In the present work we demonstrate that the learning capacity of the perceptron while running the experiments with high parameters is in line with the theoretical capacity of the perceptron (Figure 2). We conclude that for large datasets with a sufficient number of P examples and large feature vectors N the learning capacity of the perceptron tends to decrease. When the algorithm is turned to practical use we must note that the number of P examples needs to be sufficiently larger than N features, otherwise the storage capacity is not reached, and it will be impossible for the perceptron to learn the generalized rule from the examples presented.

4.1 Individual workload

Erik Zhivkoplias:

- writing the implementation of the perceptron in Python - 35%
- writing the report - 65%:
  - methods
  - results and discussion
  - conclusion

Jeroen Overschie:

- writing the implementation of the perceptron in Python - 65%
- writing the report - 35%:
  - introduction
  - implementation
  - performance concerns

References

Eamonn Keogh and Abdullah Mueen. Curse of dimensionality. *Encyclopedia of machine learning*, pages 257–258, 2010.

Minsky-Papert. An introduction to computational geometry. *MIT Press*, 1969.