# Practical Assignment I: Perceptron Training

1nd Niels Rocholl
*S3501108*

2nd Merlijn Frikken
*S3180417*

## I. Introduction

The Rosenblatt Perceptron Algorithm is a groundbreaking contribution to the field of machine learning, particularly in the realm of binary classification. It was first presented by Frank Rosenblatt in 1958 and marked a pivotal moment in the development of artificial neural networks. As one of the earliest algorithms to demonstrate the capability of learning from data, the Rosenblatt Perceptron Algorithm laid the foundation for many contemporary neural network architectures. Additionally, the Perceptron Convergence Theorem, a fundamental result in the field, guaranteed convergence for linearly separable problems and was instrumental in popularizing the use of the algorithm.

The rosenblat perceptron algorithm, basically ask the question, given a linearly separable data set, (how) can we find a perceptron weight vector $w$ that satisfies equation 3.6 from the lecture notes:

$$S_w^\mu = sign(w \cdot \xi^\mu) = S_T^\mu \text{ for all } \mu = 1, 2, ..., P, \quad (1)$$

Where $w$ is the weight vector, $\xi^\mu$ is the input pattern and $S_T^\mu$ is the desired output label. This question is elegantly addressed in the Perceptron Storage Problem (PSP, Figure 3.7 & 3.8 of the lecture notes), which states:

Perceptron Storage Problem (I) (3.7)

For a given $\mathbb{D} = \{\boldsymbol{\xi}^\mu, S_T^\mu\}_{\mu=1}^P$ with $\boldsymbol{\xi}^\mu \in \mathbb{R}^N$ and $S_T^\mu \in \{-1, +1\}$,
find a vector $\mathbf{w} \in \mathbb{R}^N$ with $\text{sign}(\mathbf{w} \cdot \boldsymbol{\xi}^\mu) = \mathrm{S}_\mathrm{T}^\mu$ for all $\mu = 1, 2, \ldots, P$.

Fig. 1. PSP in terms of weights

Perceptron Storage Problem (II) (3.9)

For a given $\mathbb{D} = \{\boldsymbol{\xi}^\mu, S_T^\mu\}_{\mu=1}^P$ with $\boldsymbol{\xi}^\mu \in \mathbb{R}^N$ and $S_T^\mu \in \{-1, +1\}$,
find a vector $\mathbf{w} \in \mathbb{R}^N$ with $E^\mu \geq c > 0$ for all $\mu = 1, 2, \ldots, P$.

Fig. 2. PSP in term of a set of inequalities

The term "storage" refers to the fact that the application of the function $S_w(\xi)$ to vectors $\xi$ that are not elements of $D$ is not currently being targeted. The Rosenblatt Perceptron Algorithm aims to solve this issue by pursuing the steps depicted in Figure 3.16 of the lecture notes:

In this paper we will apply the Rosenblatt Perceptron Algorithm to randomized data and compare the results with theoretical findings (capacity of a hyperplane), as discussed in section 3.4.2 of the lecture notes. The paper provides a comprehensive analysis of the algorithm's convergence and stability. Through this study, we aim to advance our understanding of the Rosenblatt Perceptron Algorithm and its potential applications in machine learning.

Rosenblatt perceptron algorithm (3.16)

at discrete time step $t$
- determine the index $\nu(t)$ of the current example according to (3.11)
- compute the local potential $E^{\nu(t)} = \mathbf{w}(t) \cdot \boldsymbol{\xi}^{\nu(t)} S_T^{\nu(t)}$
- update the weight vector according to

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \frac{1}{N} \Theta[c - E^{\nu(t)}] \boldsymbol{\xi}^{\nu(t)} S_T^{\nu(t)} \quad (3.17)$$

$$\left[ \begin{array}{c} \text{or, equivalently, increment the corresponding embedding strength} \\ x^{\nu(t)}(t+1) = x^{\nu(t)}(t) + \Theta[c - E^{\nu(t)}] \\ \text{(all other embedding strengths remain unchanged at time } t) \end{array} \right] \quad (3.18)$$

Fig. 3. Rosenblatt Perceptron Algorithm

## II. Method

We implemented the Rosenblatt perceptron training algorithm using the Python and the Numpy library for mathematical operations. To improve the efficiency of the code, we employ the Numba package to dynamically translate Python functions into optimized machine code, resulting in a significant increase in speed. Our approach involves dividing the problem into three key components: artificial data generation, perceptron training, and calculation of the fraction of successful runs $Q_{l.s.}$. We provide a detailed overview of each component in the following sections.

### A. The number of linearly separable dichotomies

The main results we have generated for this experiment, is the fraction of successful runs $Q_{l.s.}$. A successful run is defined as a a run in which the algorithm was able to converse ($E^v > 0$ for all $v$) within the designated number of epochs (100). This value relates to $P_{ls}$ from the lecture notes section 3.4.1 (equation 3.42). $P_{ls}$ is defined as:

$$P_{ls}(P, N) = \frac{C(P, N)}{2^P} = \begin{cases} 1 & \text{for } P \leq N \\ 2^{1-P} \sum_{i=0}^{N-1} \binom{P-1}{i} & \text{for } P > N, \end{cases} \quad (2)$$

Which represents the fraction of linearly separable dichotomies. These values can be calculated theoretically, but what we aim to do, is find them experimentally, as the fraction of successful runs $Q_{l.s.}$.

### B. Artificial Data Generation

A dataset $D$ is generated, containing $P$ randomly generated feature vectors $\xi$, with each vector having a dimensionality of $N$. The components of the individual feature vectors are drawn from a Gaussian distribution with a mean of zero and a variance of one. A binary label $S$ is assigned to each feature vector, with an equal probability of being either -1 or 1.

$$D = \{\xi^{\mu}, S^{\mu}\}_{\mu=1}^{p} \tag{3}$$

In python we implement this by creating a function *generate_artificial_data()*. In this function Numpy vectors are generated using the *numpy.random.normal(0, 1, N)* function, which returns a Numpy vector of size $N$ with entries drawn from a Gaussian normal distribution with mean 0 and variance 1. These vectors are then added to the *feature_vectors* list. Additionally, a list of labels is generated using the *numpy.choice([-1, 1])* function, which returns integers with an equal probability of being either -1 or 1.

### C. Perceptron Training

One of the most important components of this task is the Rosenblatt perceptron training algorithm. Prior to training, all weights are initialized to zero. The perceptron is then trained over $N_{max}$ sweeps, during which all samples $\xi^{\mu}$ in the dataset are iterated over. For each sample, the local potential $E^{\mu(t)}$ is calculated using the following equation:

$$E^{\mu(t)} = w(t) \cdot \xi^{\mu(t)} S^{\mu(t)} \tag{4}$$

Here, $E^{\mu(t)}$ represents the local potential of sample $\mu$ at time $t$, $w(t)$ is the weight vector at that time, $\xi^{\mu(t)}$ is the corresponding feature vector, and $S$ is the label presented at time $t$. Using equation 4, we can update the weights according to the formula:

$$w(t+1) = \begin{cases} w(t) + \frac{1}{N}\xi^{\mu(t)} S_{\mu(t)}, & \text{if } E^{\mu(t)} \leq 0, \\ w(t) & \text{else,} \end{cases} \tag{5}$$

The training process is terminated when the algorithm has reached convergence, such that $E^{v} > 0$ for all $v$ in the dataset, or when the maximum number of sweeps $N_{max}$ has been completed.

We implemented this in python by defining a function *train_perceptron()*. In this function a loop is initiated to iterate over all sweeps. Within this loop, a second loop is initiated to iterate over all samples in the given dataset. For each sample, the feature vector and label are extracted. The local potential is then calculated according to formula 4, which is performed using the numpy function *numpy.dot()*, taking as input the weight and feature vector and calculating the dot product. This dot product is then multiplied by the label. The weight vector is then updated according to formula 5. The python function representing this formula checks the value of the local potential $E^{\mu(t)}$. If the local potential is less than or equal to 0, a new weight vector is returned, which is the sum of the original weight vector and a scalar multiple of the element-wise product of the feature vector and label, where the scalar multiple is equal to $\frac{1}{N}$. If the local potential is larger than 0, the original weight vector is returned. Finally, the system is checked for convergence ($E^{v} > 0$ for all $v$). If convergence has been achieved, the weight vector is returned; otherwise, the training continues until a solution is found or the number of sweeps reaches $N_{max}$.

### D. Experimental Setup

The last part of our method is fairly simple. It calculates the the fraction $Q_{l.s.}$ of successful runs as a function of $\alpha = P/N$, by repeating the experiment for different values of $P$. We create a python function which takes in four parameters: alpha ($\alpha$), the dimensionality of each sample $N$, the number of data sets $N_d$, and the maximum number of sweeps $N_{max}$. The function returns a float value representing the fraction of successful runs. The function iterates over each data set, generates artificial data using the *generate_artificial_data()* function, initializes weights, and trains the perceptron using the *train_perceptron()* function. The number of successful runs is then calculated and returned as the fraction of successful runs over the total number of data sets.

## III. Results

We present two Figures. Figure 4 was obtained by running the algorithm with the minimal hyperparameters presented in the assignment instruction, we call this the "Base Experiment". Additionally, we obtained fine grained results for a range of $N$s, of which the results can be found in Figure 5. These results have been created by setting the step size for $\alpha$ very small, and the number of independent datasets very large. We call this the 'Large Experiment'. The exact details of these parameters can be found in Table I. The plot shown in the reader can be found in Figure 6, which will be used for comparison.

| **Hyperparameters** | **N** | $\alpha$ | $N_d$ | $N_{max}$ |
|---|---|---|---|---|
| Base Experiment | $\{20, 40\}$ | $\{0.5, 0.75..., 4.0\}$ | 50 | 100 |
| Large Experiment | $\{5, 20, 40, 100, 1000\}$ | $\{0.5, 0.505..., 4.0\}$ | 5000 | 100 |

TABLE I

We begin our analysis by examining the results depicted in Figure 4, the base experiment. As we can observe, the lines corresponding to $N = 20$ and $N = 40$ exhibit characteristics of an inverse sigmoidal curve. The large step size of $\alpha$ result in sharp corners at certain points along the curves. In contrast, if we examine Figure 6, we can observe that the line corresponding to $N = 5$ and the line corresponding to $N = 20$ resemble a stair-step pattern. Despite the presence of a similar trend in the Base Experiment, this stair-step characteristic is not evident.

In Figure 5, the Large Experiment, five lines are depicted, ranging from $N = 5$ to $N = 1000$. The results of this experiment, obtained through the use of a new hyperparameter setting, exhibit high quality and bear a strong resemblance to those shown in Figure 6. For $N = 5$ and $N = 20$, the stair-step characteristics are clearly visible, though some noise, primarily

along the y-axis, is still present. This noise could potentially be reduced by increasing $N_d$ and $N_max$, in conjunction with a smaller step size of $\alpha$. For higher values of N, the line appears to approach a step function. It is worth mentioning that, due to computational constraints, the $N = 1000$ line was generated with a larger $\alpha$ step size (0.1) and a smaller $N_d$ (500).

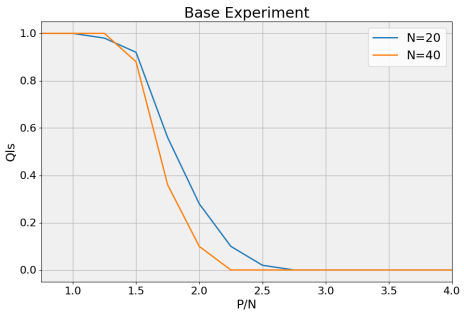


Fig. 4. Small $N$, $N_d$, large $\alpha$ values

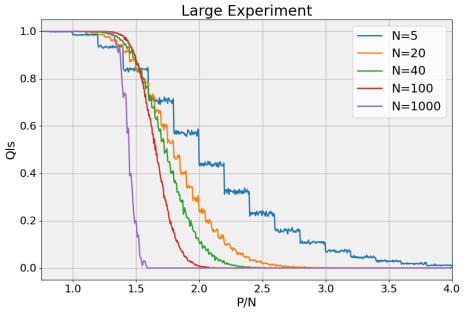


Fig. 5. Large $N$, $N_d$, small $\alpha$ values

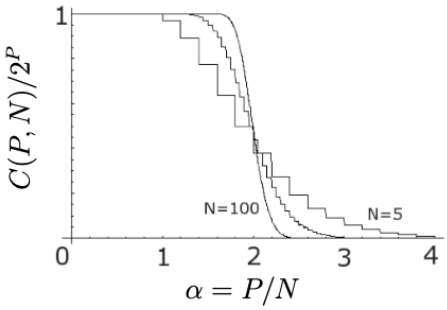


Fig. 6. Results from the reader: $N = \{5, 20, 100\}$

## IV. Discussion

One of the main point of discussion is the obvious difference between the results from the reader (Figure 6) and our own results (Figure 5). The notable difference is that the results from the reader show no noise, and ours do. We suspect that the results from the reader are a plot of the mathematical definitions, which represent the $Q_{l.s.}$ in the limit. This is also why we expect that our results would smooth out as we increase $N_d$ and $N_{max}$, and decrease the step size of $\alpha$. Therefore, we predict that in the limit, our results approach the results shown in 6.

Another notable difference is that the lines in Figure 6 intersect at $\alpha = P/N = 2$. However in our results this is not the case. This can most likely be attributed to the fact that our implementation executes the Perceptron algorithm for a limited duration of 100 epochs. Within these 100 epochs, the algorithm can efficiently update the weights and converge for small datasets. Conversely, larger datasets necessitate more updates for each data point and therefore, 100 epochs may not be sufficient for convergence to occur. As a result, an increase in the number of examples shifts the line to the left with a higher frequency of unsuccessful runs. This phenomenon can be interpreted as the algorithm not having adequate time to arrive at a solution. An increase in the number of epochs may shift these lines towards the right, bringing them closer to the theoretical functions.

A critical theoretical component of the experiment concerns the capacity of the hyperplane. The capacity of a hyperplane denotes its ability to linearly partition a set of feature vectors within an N-dimensional input space into binary classifications or dichotomies. This concept is represented in Equation 3.42 in the lecture notes, or Equation 2 in Section II of this paper. The equation posits that, given a data set is in general position, if the quantity of feature vectors, represented by $P$, is less than or equal to the dimensionality of the feature vectors, denoted by $N$, the data is guaranteed to be linearly separable and, as a result, the fraction equals 1. This is due to the presence of more data points than features, allowing for sufficient degrees of freedom to segregate the data through a hyperplane. Analysis of the simulation indicates that the formula predicts a decline in the fraction below 1 when $P/N$ exceeds 1, signifying that $P$ exceeds $N$. This conjecture is indeed substantiated in the results depicted in Figure 5.

## V. Conclusion