SI221 Assignment #4:
Perceptron - 06/10/2020

Instructions

Read all the instructions below carefully before you start working on the assignment.

- Please submit the source code and a pdf report of your work. You can use the Jupyter notebook instead, by submitting the ipynb file. Each file must have as title: *TP_Perceptron_Student1_Student2*.
- Late assignments will not be corrected.
- You must do this assignment in groups of 2. Please submit no more than one submission per group.
- You must implement a perceptron algorithm from scratch.
- Code that does not work will not be considered.
- Send your work to: `nazareth@telecom-paris.fr` and `colombo.pierre@gmail.com`

Practical assignment objective

This assignment is aimed at coding a perceptron from scratch in order to learn how this simple but powerful linear binary classifier works. In the following sections you can find a brief summary on what a perceptron is and how it works, and finally the text of the assignment.

Reminders about perceptron[1]

Historical introduction

The perceptron occupies a special place in the historical development of neural networks: It was the first algorithmically described neural network. Its invention by Rosenblatt, a psychologist, inspired engineers, physicists, and mathematicians alike to devote their research effort to different aspects of neural networks in the 1960s and the 1970s. The perceptron is the simplest form of a neural network used for the classification of data said to be linearly separable (i.e., data that lie on opposite sides of a hyperplane). Basically, it consists of a single neuron with adjustable synaptic weights and bias. The algorithm

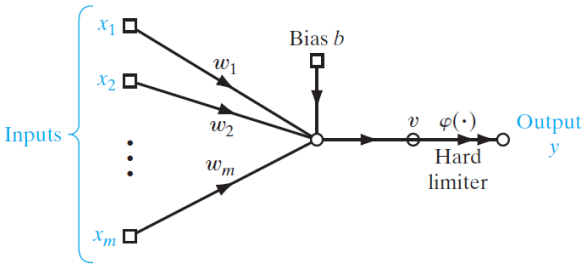
[1]The text and the pictures of this section were taken from *Haykin, O., "Neural Networks and Learning Machines", Pearson, 2009*

used to adjust the free parameters of this neural network first appeared in a learning procedure developed by Rosenblatt for his perceptron brain model. Indeed, Rosenblatt proved that if the data used to train the perceptron are drawn from two linearly separable classes, then the perceptron algorithm converges and positions the decision surface in the form of a hyperplane between the two classes. A perceptron is limited to performing pattern classification with only two classes. By expanding the output (computation) layer of the perceptron to include more than one neuron, we may correspondingly perform classification with more than two classes.

Working with a perceptron

A perceptron consists of a linear combiner followed by a hard limiter (e.g. the sign function, the Heaviside step function or a simple function like these) (see Figure 1).

Figure 1: Perceptron's structure.



The summing node of the neural model computes a linear combination of the inputs applied to its synapses, as well as incorporates an externally applied bias. The resulting sum is applied to a hard limiter. Accordingly, for example if the hard limiter is the Heaviside step function, the neuron produces an output equal to 1 if the hard limiter input is positive, and 0 if it is negative. The synaptic weights of the perceptron are denoted by w_1, w_2, ...,w_m. Correspondingly, the inputs applied to the perceptron are denoted by x_1, x_2, ..., x_m. The externally applied bias is denoted by b. From the model, we find that the hard limiter input of the neuron is:

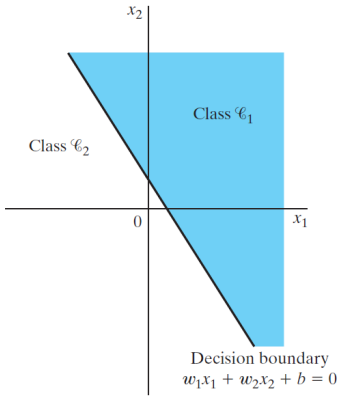
$$v = \sum_{i=1}^{m} w_i x_i + b.$$

The goal of the perceptron is to correctly classify the set of externally applied stimuli x_1, x_2, ..., x_m into one of two classes, $\mathcal{C}_1$ or $\mathcal{C}_2$. The decision rule for the classification is to assign the sample represented by the inputs x_1, x_2, ..., x_m to class $\mathcal{C}_1$ if the perceptron output y is 1 and to class $\mathcal{C}_2$ if it is 0. To develop insight into the behavior of a classifier, it is customary to plot a map of the decision regions in the m-dimensional signal space spanned by the m input variables x_1, x_2, ..., x_m. In the simplest form of the perceptron, there are two decision regions separated by a hyperplane, which is defined by:

$$\sum_{i=1}^{m} w_i x_i + b = 0.$$

This is illustrated in Figure 2 for the case of two input variables x_1 and x_2, for which the decision boundary takes the form of a straight line.

Figure 2: Example of hyperplane for a bi-dimensional, 2-classes classification problem.



A sample (x_1, x_2) that lies above the boundary line is assigned to class $\mathcal{C}_1$, and a sample (x_1, x_2) that lies below the boundary line is assigned to class $\mathcal{C}_2$. Note also that the effect of the bias b is merely to shift the decision boundary away from the origin. The synaptic weights w_1, w_2, ..., w_m of the perceptron can be adapted on an iteration by iteration basis called *error-correction rule*. To derive this rule, it is more convenient to work with the modified model in Figure 3. In this model, equivalent to the one of Figure 1, the bias b is treated as a synaptic weight driven by a fixed input equal to $+1$. We may thus define the $(m+1)$-by-1 input vector:

$$\boldsymbol{x}(n) = \begin{bmatrix} 1 & x_1(n) & x_2(n) & \cdots & x_m(n) \end{bmatrix}^{\mathrm{T}},$$

where n denotes the time-step in applying the algorithm, $x_0(n) = 1$ and $\boldsymbol{x}(n)$ denotes the nth input vector of the data set $\{\boldsymbol{x}(1), \ldots, \boldsymbol{x}(N)\}$. Correspondingly, we define the $(m+1)$-by-1 weight vector as:

$$\boldsymbol{w}(n) = \begin{bmatrix} b(n) & w_1(n) & w_2(n) & \cdots & w_m(n) \end{bmatrix}^{\mathrm{T}},$$

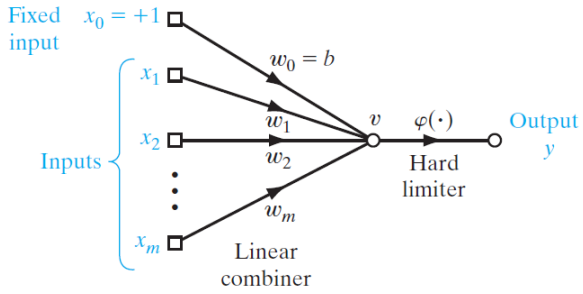
where $w_0(n) = b(n)$ and $\boldsymbol{w}(n)$ denotes the weight vector at the nth iteration of the algorithm.

Accordingly, the linear combiner output is written in the compact form as follows:

$$\begin{aligned} v &= \sum_{i=0}^{m} w_i(n)\, x_i(n) = \boldsymbol{w}^{\mathrm{T}}(n)\, \boldsymbol{x}(n), \\ y &= \varphi\left(\boldsymbol{w}^{\mathrm{T}}(n)\, \boldsymbol{x}(n)\right). \end{aligned}$$

Suppose then that the input variables of the perceptron originate from two linearly separable classes. Let $\mathcal{H}_1$ be the subspace of training vectors that belong to class $\mathcal{C}_1$, and let $\mathcal{H}_2$ be the subspace of training vectors that belong to class $\mathcal{C}_2$. The union of $\mathcal{H}_1$ and $\mathcal{H}_2$ is the complete space denoted by $\mathcal{H}$. Given the sets of vectors $\mathcal{H}_1$ and $\mathcal{H}_2$ to train the

Figure 3: Modified model of perceptron



classifier, the training process involves the adjustment of the weight vector $\boldsymbol{w}$ in such a way that the two classes $\mathcal{C}_1$ and $\mathcal{C}_2$ are linearly separable. That is, there exists a weight vector $\boldsymbol{w}$ such that we may state:

$$\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{x} > 0 \text{ for every input vector } \boldsymbol{x} \text{ belonging to class } \mathcal{C}_1$$
$$\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{x} \leq 0 \text{ for every input vector } \boldsymbol{x} \text{ belonging to class } \mathcal{C}_2$$

We have arbitrarily chosen to say that the input vector $\boldsymbol{x}$ belongs to class $\mathcal{C}_2$ if $\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{x} = 0$. Given the subsets of training vectors $\mathcal{H}_1$ and $\mathcal{H}_2$, the training problem for the perceptron is then to find a weight vector $\boldsymbol{w}$ such that the two inequalities above are satisfied. The algorithm for adapting the weight vector of the elementary perceptron may now be formulated as follows:

1. If the nth member of the training set, $\boldsymbol{x}(n)$, is correctly classified by the weight vector $\boldsymbol{w}(n)$ computed at the nth iteration of the algorithm, no correction is made to the weight vector of the perceptron according to the rule:

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) \text{ if } \boldsymbol{w}^{\mathrm{T}}(n)\,\boldsymbol{x}(n) > 0 \text{ and } \boldsymbol{x}(n) \text{ belongs to class } \mathcal{C}_1,$$
$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) \text{ if } \boldsymbol{w}^{\mathrm{T}}(n)\,\boldsymbol{x}(n) \leq 0 \text{ and } \boldsymbol{x}(n) \text{ belongs to class } \mathcal{C}_2.$$

2. Otherwise, the weight vector of the perceptron is updated according to the rule:

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) - \eta(n)\,\boldsymbol{x}(n) \text{ if } \boldsymbol{w}^{\mathrm{T}}(n)\,\boldsymbol{x}(n) > 0 \text{ and } \boldsymbol{x}(n) \text{ belongs to class } \mathcal{C}_2,$$
$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) + \eta(n)\,\boldsymbol{x}(n) \text{ if } \boldsymbol{w}^{\mathrm{T}}(n)\,\boldsymbol{x}(n) \leq 0 \text{ and } \boldsymbol{x}(n) \text{ belongs to class } \mathcal{C}_1,$$

where the learning-rate parameter $\eta(n)$ controls the adjustment applied to the weight vector at iteration n. If $\eta(n) = \eta > 0$, where η is a constant independent of the iteration number n, then we have a fixed-increment adaptation rule for the perceptron. The adaptation of the weight vector $\boldsymbol{w}(n)$ is nicely summarized in the form:

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) + \eta\,(d(n) - y(n))\,\boldsymbol{x}(n), \tag{1}$$

where $d(n)$ is the desired output for $\boldsymbol{x}(n)$.

Getting started!

1 Synthetic data

Consider a data set $\{\mathbf{x}(n), d(n)\}_{n=1}^{200}$ consisting of 200 points $\mathbf{x}(n) = (x_1(n), x_2(n))$ and their corresponding labels $d(n)$, such that the first 100 points have label $d(n) = 1$ and are generated according to a Gaussian distribution $\mathbf{x}(n) \sim \mathcal{N}([1, 0], \sigma^2\mathbf{I})$, and such that the other 100 points have label $d(n) = 0$ and are generated according to a Gaussian distribution $\sim \mathcal{N}([-1, 0], \sigma^2\mathbf{I})$.

In order to visualize each generated data set and the corresponding solution given by the algorithm, plot all generated points in a coordinate plane and the estimated decision line, given by the estimated coefficients of the perceptron at the convergence.

1. Evaluate the computational complexity of the perceptron in terms of arithmetic operations per iteration.

2. Consider four different values $\{0.05, 0.25, 0.50, 0.75\}$ of the noise variance σ^2. For each of these values, run the perceptron over 50 randomly generated data sets, compute the average error $e(\sigma^2)$ and its standard deviation $s(\sigma^2) = \sqrt{\frac{1}{50}\sum_{i=1}^{50}(e_i - e)^2}$, where e_i denotes the fraction of misclassified points. Represent graphically $e(\sigma^2)$ and $s(\sigma^2)$ for the four values of σ^2 (use error bars). Comment.

3. Generate one data set with $\sigma^2 = 0.15$. A new random data set is now obtained by flipping each label $d(n)$ with probability p to obtain $\tilde{d}(n)$. Considering the generated data set $\{\mathbf{x}(n), \tilde{d}(n)\}_{n=1}^{200}$, repeat the previous experiments for $p \in \{0\%, 5\%, 10\%, 20\%\}$ and evaluate $e(p)$ and $\sigma^2(p)$. Comment.

2 Real data

Let's play with the real data now, called the *Iris flower data set*. This data set is widely used in machine learning and can be found in `https://archive.ics.uci.edu/ml/datasets/iris` or in the machine learning package *Scikit-learn*.

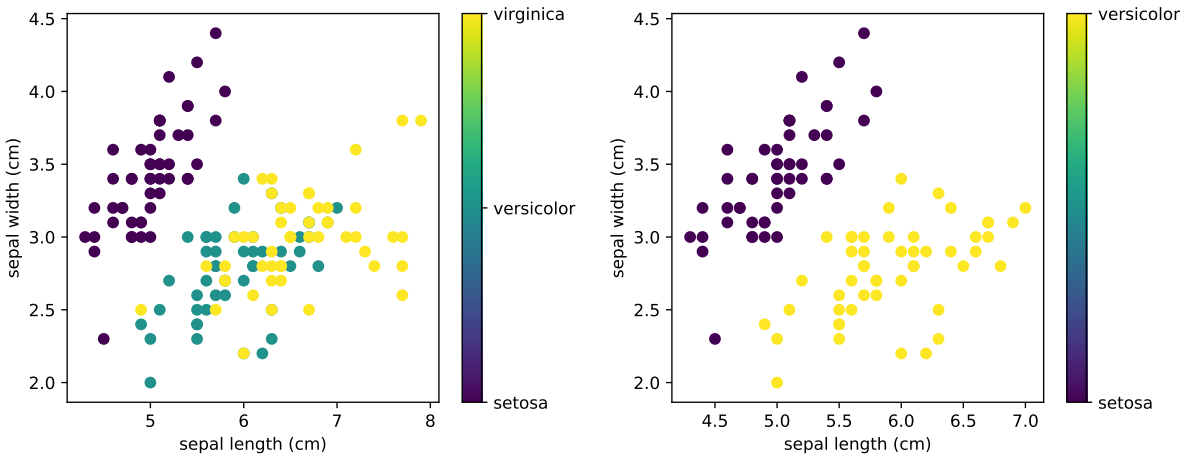
Short description: This data set contains a set of 150 samples, which consists of 50 samples from each of three species of Iris: *setosa* (label 0), *versicolor* (label 1), and *virginica* (label 2). Each sample was measured in four features: sepal length, sepal width, petal length, and petal width.

Data Preparation and Visualization: Split the data set into a balanced (with respect to the labels) training and test set, containing respectively 80% and 20% of the data set.

2.1 Application to binary classification

1. Visualize the first two features of the training set, *i.e.*, sepal length and sepal width, and their corresponding labels/classes. You should obtain a figure similar to the Fig. 4a.

2. Now consider only the data set containing two classes: *setosa* and *versicolor*, you should get the Fig. 4b. Classify the data sat into two classes with the Perceptron. Report the training and test errors. Comment.



(a) (b)

Figure 4: Visualization of two features of the Iris data set: (a) shows the data set of all 3 classes and (b) shows only the data set of the first 2 classes.

2.2 (Optional) Application to multi-label classification

In this section, we take into account all 3 classes of the data set for multiclass, by implementing the *One-versus-All* method (for more information check the chapter 17 of the Book: "Understanding Machine Learning: From Theory to Algorithms", by Shai Shalev-Shwartz and Shai Ben-David).

Propose a method to implement a three class classifier with multiple binary classifiers.