perceptronPreamble

Brief history of neural networks and a perceptron implementation

Brandon Marquez Salazar

I. Introduction

There are several applications for Neural Networks nowadays and seems to be a new technology made no so much ago, but it's origins can be found between 1897 and 1904. When Santiago Ramón y Cajal published "*Histologie du système nerveux de l'homme et des vertébrés*" where brain cells were mentioned and described as the basic element of the nervous system; starting a new interest and acceptance over the "Neuron Doctrine" extending itself into ANN investigation.

II. Mathematical model of the neuron

With the "Neuron Doctrine" popularity the need of knowledge of neuron's inner behaviour inspired the creation of models intended to comprehend the electrical communication from electrical excitation to signal sending.

In 1943 McCulloch and Pitts [4] proposed the first model of a neuron described as a binary element with different weighed inputs $x_k w_k$ whose sum added to the bias (threshold) θ is evaluated by a nonlinear function $S(\cdot)$ called activation function or transfer function and giving the "desired" output O.

So, for any j-th neuron, the mathematical representation is given by the following expression.

$$O_j = S\left(\sum_{k=1}^{n} w_{jk}\, x_{jk} - \theta\right) \tag{1}$$

Graphically we can represent it by the following scheme.

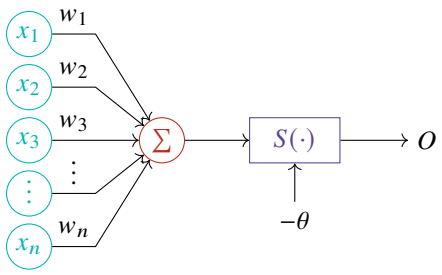


Fig. 1. First neuron model.

III. The first perceptron

The first algorithm based on the neuron model was made in 1958 and its implementation was conceived with the "Perceptron Mark I" by Rosenblatt. This algorithm was made for image recognition using potentiometers, and motors to modify each potentiometer's value.

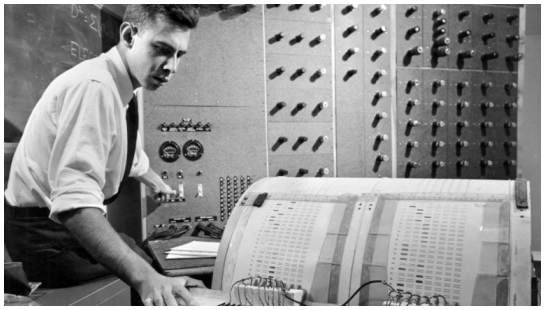


Fig. 2. In 1957, Frank Rosenblatt built the Mark I Perceptron at the Cornell Aeronautical Laboratory [8].

A. The perceptron algorithm

Suppose any desired output Y_j for the j-th neuron, and for each iteration t the algorithm follows the operations below:

$$O_j(t) = \text{sign}\left(\sum_{k=1}^{n} w_k(t) x_{jk}\right) \tag{2}$$

$$\epsilon_j(t) = \frac{\sum_1^n \left|Y_j - O_j(t)\right|}{N} \tag{3}$$

$$w_j(t+1) \longleftarrow w_j(t) - \left(Y_j - O_j(t)\right) r \tag{4}$$

Where Eq. **??** the output, Eq. **??** computes the error and Eq. **??** the weights update at iteration t.

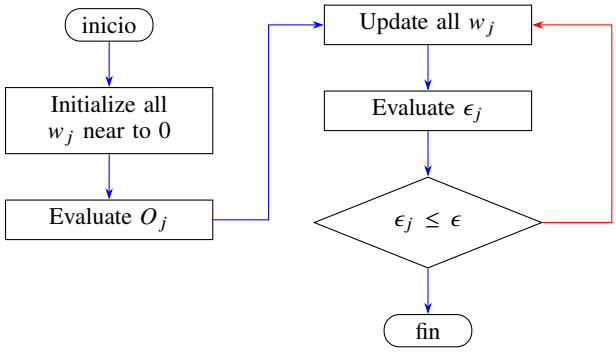


Fig. 3. Flowchart that describes Rosenblatt's perceptron algorithm.

Something important to keep in mind it that it's convergence depends on that the classes should be linearly separables.

B. Activation, update and error functions

During this period, and until today, there are different approaches to neural network implementations trying to optimize it's learning, and classifying performance.

Adaline (ADAptive Linear Neuron Element) is a neural network developed by Widrow and Hoff, which was mainly for noise reduction, adaptive filtering, echo reduction, among other applications. Widrow and Hoff proposed LMS (Least Mean Squares) algorithm.

About activation functions, there are different activation functions we can use, such as **sign**, **sigmoid**, **gaussian**, etc. Also, some functions were defined in the search of the less loss activation functions such as **ReLU**.

IV. A PERCEPTRION IMPLEMENTATION

V. Perceptron algorithm implementation

I'll use numpy as a very common package used for scientific computation, **make_blobs** from Sci-kit learn for data generation and pyplot from matplotlib for data visualization.

[] #!pip install scikit-learn import numpy as np from sklearn.datasets import make_blobs import matplotlib.pyplot as plt

Now, in order to implement a perceptron we have to define the three steps needed and the evaluation function

A. Neuron Model

First the perceptron output, which is the most important element, it's the neuron model. It will receive a vector of inputs X_{jk} a vector of weights W_{jk} and a reference for the activation function $S(\cdot)$.

[] def perceptronOutput(W,X,bias,ActivationFunction): WeighedInputs = np.dot(W,X) Net = WeighedInputs - bias O = ActivationFunction(Net) return O

B. Error equation

This function computes the error of the perceptron based on its output compared to the desired output. It receives the perceptron output vector O_j, the desired output vector Y_j and the number of patterns N.

[] def computeError(Y,O,N): DeltaSum = np.sum(np.abs(Y-O)) Err = DeltaSum/N return Err

C. Weights update function

This function receives the current weights vector W_j, the perceptron and desired output vectors O_j and Y_j and learning rate r. Then returns the new weights vector.

[] def updateWeights(Y,O,W,X,r): NewWeights = W - (Y-O)*X*r return NewWeights

D. Evaluation function

For this one, I'll create my own evaluation function based on a normalized [0,1] sign function.

$$S(X) = \begin{cases} 1, & X > 0 \\ 0, & X \leq 0 \end{cases}$$

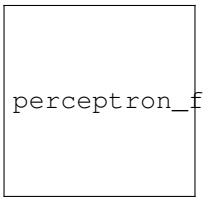
[] def S(Net): if Net<1 : return 0; return 1;

VI. Training

A. Input data

Using Scikit-Learn I'll create a new dataset of inputs and its desired output (classification) in order to train our perceptron.

[] # This one is to get 3000 elements to classify N=3000 # Two features to input the perceptron nFeatures=2 X,Y = make_blobs(# Number of elements # Number of features n_samples=N, n_features=nFeatures, # Two classes # Standard deviation centers=2, cluster_std=1.9, # Only to get randomly positioned shuffle=True, # Random Seed random_state=1201931) plt.scatter(X[:,0],X[:,1],c=Y,cmap='magma',s=5) plt.xlabel('Feature 1') plt.xlabel('Feature 2') plt.title('Sample space') plt.colorbar(label="Classes") plt.show()



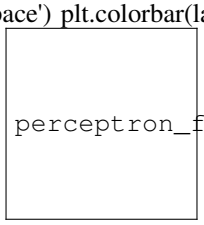
Then I'll derived of the features quantity, I'll create a weights vetor W. And create a function which will process a datum.

[] bias = -0.2 learnRate = 0.6 W=np.zeros(nFeatures) for i in range(len(X)): Xi = X[i]; Yi = Y[i]; Oi=perceptronOutput(Xi,W,bias,S) Err=computeError(Yi,Oi,N) W=updateWeights(Yi,Oi,W,Xi,learnRate)

Then I'll plot the sample space with a frontier made by the weights

[] m=-W[0]/W[1] Po=[min(X[:,0]) , max(X[:,1])] Pf=np.multiply(Po,m)+np.divide(bias,W[1])

plt.plot(Po,Pf,'g-',linewidth=0.8) plt.scatter(X[:,0],X[:,1],c=Y,cmap='magma',s=5) plt.xlabel('Feature 1') plt.xlabel('Feature 2') plt.title('Sample space') plt.colorbar(label="Classes") plt.show()



REFERENCES

[1] Mo Costandi. *The discovery of the neuron*. 2006. URL: https://neurophilosophy.wordpress.com/2006/08/29/the-discovery-of-the-neuron/.

[2] M. Ekman. *Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow*. Pearson Education, 2021. ISBN: 9780137470297. URL: https://books.google.com.mx/books?id=wNnPEAAAQBAJ.

[3] S.S. Haykin. *Neural Networks and Learning Machines*. Neural networks and learning machines v. 10. Prentice Hall, 2009. ISBN: 9780131471399. URL: https://books.google.com.mx/books?id=K7P36lKzI_QC.

[4] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *Bulletin of Mathematical Biology* 52.1 (1990), pp. 99–115. ISSN: 0092-8240. DOI: https://doi.org/10.1016/S0092-8240(05)80006-0. URL: https://www.sciencedirect.com/science/article/pii/S0092824005800060.

[5] quarto. *Guide*. URL: https://quarto.org/docs/guide/.

[6] quarto. *PDF Options*. URL: https://quarto.org/docs/reference/formats/pdf.html.

[7] quarto. *Welcome Page*. URL: https://quarto.org.

[8] Parul Singh. *A Must-Read History of Artificial Intelligence*. 2020. URL: https://cyfuture.com/blog/history-of-artificial-intelligence/.