Analysis of the Rosenblatt Perceptron: Convergence and Practical Application

From the mathematical proof of its convergence to a Python implementation

Hernández Pérez Erick Fernando

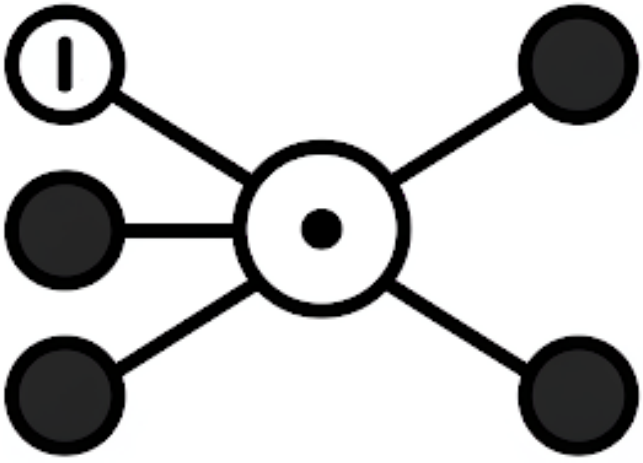


Figure 1: Oversimplified icon of a perceptron

A world where machines can learn by themselves, adapt to new situations and make decisions without having to be constantly reprogrammed. This idea, which is now part of our reality, was for centuries an unattainable dream. From the first attempts to create automata in ancient times to the sophisticated computers of the 20th century, humans have always sought ways to replicate their own intelligence in inert matter. However, although machines could process information at unimaginable speeds, they remained rigid, unable to learn from experience as the human brain does.

It was in this context of curiosity and intellectual ambition that the first attempts to model thought arose, inspired by the functioning of neurons. The question was challenging: could an artificial structure learn and make decisions as living beings do? The answer to this question would mark a before and after in the field of artificial intelligence, laying the foundations for what we now know as machine learning.

The road was long, full of skepticism and obstacles, but one discovery in particular would open the door to a new era in computing and forever change the way we understand machine learning.

As Haykin mentions, the perceptron holds a special place in the historical development of neural networks: it was the first neural network described algorithmically. Its invention by Rosenblatt, a psychologist, inspired engineers, physicists, and mathematicians alike to devote their research efforts to different aspects of neural networks in the 1960s and 1970s (2009)[1]. It was in 1958 that Rosenblatt proposed the perceptron as the first model for learning with a teacher, what is often called supervised learning. If you want to delve deeper into the McCulloch Pitts model, you can see the third issue of this series. Implementing McCulloch and Pitts Neuron in Python.

A perceptron is an artificial neuron, and therefore a unit of a neural network. This is a different model than the one proposed by McCulloch-Pitts, although it served as the basis for the model proposed by Frank Rosenblatt. However, this term is also often used to refer to the algorithm for supervised learning of binary classifiers. This algorithm is responsible for ensuring that artificial neurons learn and, therefore, can handle the elements of a series of data for classification.

It is important to emphasize the "neural network unit" part. A single perceptron is capable of classifying data by itself, but it has a very important limitation: linearly non-separable data sets. This is the case of the XOR function that was discussed in the McCulloch-Pitts entry.

The goal of the perceptron is to correctly classify a sequence of inputs $x_1, x_2, ..., x_n$ into two classes C_1 and

C_2. To do this, the perceptron will return for each point a value $\hat{y}$ which will be 1 for C_1 and -1 for C_2.

The perceptron assumption: Linearly separable sets.

For the perceptron to work properly, the two classes must be linearly separable. This, in turn, means that the patterns to be classified must be sufficiently separated from each other to ensure that the decision surface consists of a hyperplane.

Put more simply, linearly separable sets are those that can be separated by a line (in 2D), a plane (in 3D), or a hyperplane (in higher dimensions). This means that there exists a line, plane, or hyperplane such that all points of one class are on one side and all points of the other class are on the other side.

For example, if you have a set of blue and red points in a 2D graph and you can draw a straight line such that all the blue points are on one side of the line and all the red points are on the other side, then that set of points is linearly separable.

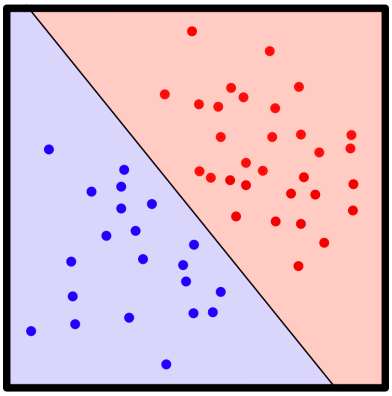
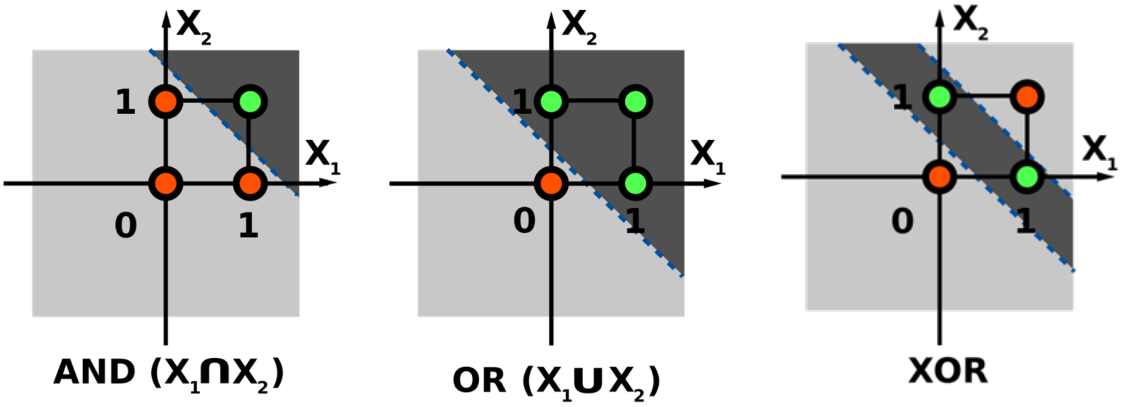


Figure 2: Linear separability

On the other hand, linearly non-separable sets are those that cannot be separated by a line, plane, or hyperplane. No matter how you try to draw the line, there will always be points of both classes on both sides of the line.

The simplest cases are arguably logic gates. Figure 3 shows that AND and OR gates are linearly separable, while XOR gates are not.



(a) (b) (c)

Figure 3: Linear separability of AND, OR and XOR gates

Perceptron components

- **Inputs**: Inputs are the values that enter the perceptron for its respective calculation.
- **Weights**: Weights are numerical values that determine the importance of each input. Each input is multiplied by its corresponding weight. The weights are adjusted during the learning process to improve the accuracy of the perceptron.
- **Bias**, which is also often referred to as a **threshold value**: The bias is a constant value (often 1) that is added to the weighted sum of the inputs and the weights. Like the weights, the bias value is adjusted during the learning process.
- **Weighted sum**: The weighted sum is the result of multiplying each input by its corresponding weight and then adding all these products. To this we add the bias. This sum is called the induced local field. Mathematically, if we have n inputs $x_1, x_2, \ldots, x_n$ and their corresponding weights $w_1, w_2, \ldots, w_n$ and the bias b, the weighted sum is calculated as follows:

  $$\sum_{i=1}^{n} w_i x_i + b$$

  Another way to deal with the bias is to consider it as a synaptic weight with a fixed input of +1, this gives us a more compact way to work:

  $$\sum_{i=0}^{n} w_i x_i = w_0 x_0 + \sum_{i=1}^{n} w_i x_i = b + \sum_{i=1}^{n} w_i x_i$$

- **Activation Function**: The activation function ϕ takes the weighted sum and produces the final output of the perceptron. In our case, our activation function is the sign function.

  $$\phi = \text{sgn}(x) = \begin{cases} 1 & \text{si } x > 0, \\ -1 & \text{si } x < 0. \end{cases}$$

The total entries can be grouped into a single vector represented by $\mathbf{x} \in \mathbb{R}^{n+1\times 1} = (+1, x_1, x_2, ..., x_n)^T$. Similarly, all the weights can be put together in a vector that is represented by $\mathbf{w} \in \mathbb{R}^{n+1\times 1} = (b, w_1, w_2, ..., w_n)^T$

Thus, using linear algebra, we can write the perceptron calculation as follows:

$$\hat{y} = \phi(\mathbf{w}^T \mathbf{x})$$

Perceptron Convergence Algorithm

Before we can continue with the steps of this algorithm, we must first define what the hyperparameter η is. This refers to the learning rate and controls the size of the steps that the learning algorithm takes when adjusting the perceptron weights during the training process. As Haykin points out, its value is not important as long as it is positive (2009)[1]. It is usually defined with values of 0.1, 0.01 or 0.001.

Now we are ready to start with the algorithm. The perceptron convergence algorithm consists of 5 steps:

- Initialization: We start with the weights w set to 0 or randomly.
- Activation: Each of the x in our dataset will be passed to the perceptron as well as the desired response y
- Calculation of the current response: The result of the perceptron is calculated:

  $$\hat{y} = \text{sgn}(\mathbf{w}^T \mathbf{x})$$

- Updating the weights: The weights are updated according to:

  $$\mathbf{w} = \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$$

- Repetition: Return to step 2 until the correct weights are found or until a specific number of iterations have passed.

Perceptron Convergence Proof

Since our perceptron assumes that the data is linearly separable, this means that:

$$\exists \mathbf{w^*} : \forall (x_i, y_i) \in D \qquad y_i \mathbf{w^*}^T \mathbf{x}_i > 0$$

It is also important to note that by finding a $\mathbf{w^*}$ we can find another one, simply by rescaling it $\mathbf{w^*}' = \alpha \mathbf{w^*}$. This way we will choose our $\mathbf{w^*}$ such that $||\mathbf{w^*}|| = 1$.

After this, we are going to rescale our data so that it falls on a circle of radius 1. To do this we do $\forall i : ||x_i|| \leq 1$ by dividing each x_i by the maximum norm of the points. This will help us with the proof. Also note that scaling will not affect our solution at all, because once it is found, we can return to the original scale.

We also define what our margin is. It is the smallest distance between our data and our decision region:

$$\gamma = \min_{x_i, y_i \in D} |\mathbf{x}^T \mathbf{w^*}| > 0$$

We will assume that the value of $\eta = 1$ and that our weights are $\mathbf{w} = 0$

Now we can start with the demonstration. First we will see how $w^T w*$ changes in an update:

$$\mathbf{w}^T \mathbf{w^*} \rightarrow (\mathbf{w} + y\mathbf{x})^T \mathbf{w^*} = \mathbf{w}^T \mathbf{w^*} + y\mathbf{x}^T \mathbf{w^*} = \mathbf{w}^T \mathbf{w^*} + y\mathbf{w^*}^T \mathbf{x}$$

Where the term $y\mathbf{w^*}^T \mathbf{x}$ by our assumption is positive. This means that our initial product becomes increasingly positive and since we have established our γ we can conclude that:

$$(\mathbf{w} + y\mathbf{x})^T \mathbf{w^*} \geq \mathbf{w}^T \mathbf{w^*} + \gamma$$

This means that if we make an update, our inner product of the hyperplane we are searching for with the hyperplane we want to find grows by at least γ. This is our lower bound.

Ahora checaremos como cambia $\mathbf{w}^T \mathbf{w}$.

$$\mathbf{w}^T \mathbf{w} \rightarrow (\mathbf{w} + y\mathbf{x})^T (\mathbf{w} + y\mathbf{x}) = \mathbf{w}^T \mathbf{w} + 2y\mathbf{w}^T \mathbf{x} + y^2 \mathbf{x}^T \mathbf{x}$$

Since we are doing an update, that means that $y\mathbf{w}^T \mathbf{x} < 0$; we also have that $y^2 = 1$ and $\mathbf{x}^T \mathbf{x} \leq 1$. Therefore, we can conclude that:

$$(\mathbf{w} + y\mathbf{x})^T (\mathbf{w} + y\mathbf{x}) \leq \mathbf{w}^T \mathbf{w} + 1$$

This being our upper bound.

Now consider the case that occurs after M updates, and applying the Cauchy-Schwarz inequality:

$$M\gamma \leq \mathbf{w}^T \mathbf{w^*} = |\mathbf{w}^T \mathbf{w^*}| \leq ||\mathbf{w}|| \cdot ||\mathbf{w^*}|| = ||\mathbf{w}|| = \sqrt{\mathbf{w}^T \mathbf{w}} \leq \sqrt{M}$$

And this can be expressed as:

$$M\gamma \leq \sqrt{M} \rightarrow \sqrt{M}\gamma \leq 1 \rightarrow M \leq \frac{1}{\gamma^2}$$

Which means that our number of iterations cannot be greater than the inverse of the square of our margin.

Python implementation

Perceptron

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

class Perceptron:
    def __init__(self, iterations=100, eta=0.1, random_state=1):
        self.iterations = iterations
        self.eta = eta
```

```
        self.random_state = random_state
        self._w = None
        self._b = None
        self._errors = []

    def __str__(self):
        return f'Iterations = {self.iterations}; Eta = {self.eta}; Random State =
        ↪  {self.random_state}; W = {self._w}; b = {self._b}'

    def __repr__(self):
        output = f'{self.__class__.__name__}('
        output += ', '.join(f'{key}={value}' for key, value in vars(self).items())
        output += ')'
        return output

    def forward(self, X):
        return np.dot(X, self._w) + self._b

    def predict(self, X):
        """
        Classifies data with labels (1, -1).

        Args:
            X (array): Data to classify. Shape [num_examples, num_features].

        Returns:
            array: Class labels.
        """
        return np.where(self.forward(X) > 0, 1, 0)

    def fit(self, X, y):
        """
        Trains the perceptron with data X and labels y.

        Args:
            X (array): Training data. Shape [num_examples, num_features].
            y (array): Target values. Shape [num_examples].

        Returns:
            self: Trained perceptron object.
        """
        rgen = np.random.RandomState(self.random_state)
        self._w = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self._b = 0.0
        self._errors = []

        for _ in range(self.iterations):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self._w += update * xi
                self._b += update
                errors += int(update != 0.0)
            self._errors.append(errors)
```

```
        return self
```

Visualization

```
def plot_decision_regions(X, y, classifier, resolution=0.002):
    # setup marker generator and color map
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
    np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class examples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
        y=X[y == cl, 1],
        alpha=0.8,
        c=colors[idx],
        marker=markers[idx],
        label=f'Class {cl}',
        edgecolor='black')
```

AND gate

AND

```
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])

y_and = np.array([0, 0, 0, 1])

perceptron = Perceptron(iterations=10, eta=0.1, random_state=1)

p_and = perceptron.fit(X, y_and)
plot_decision_regions(X,y_and,p_and)
```

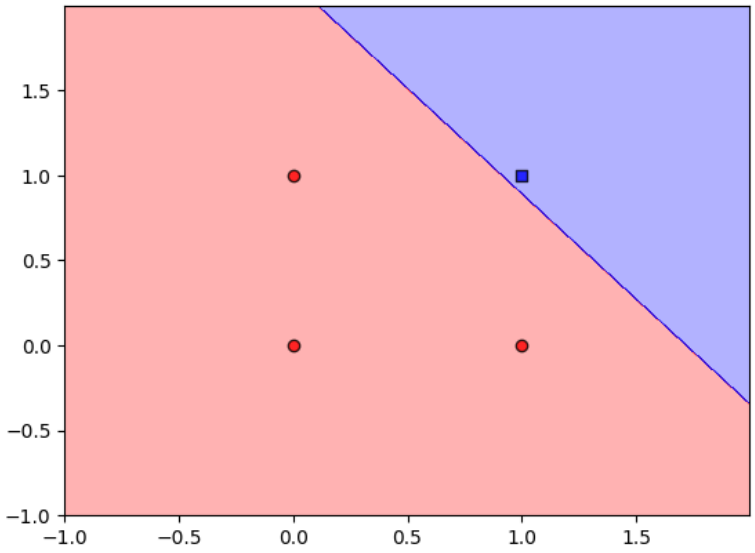


Figure 4: Perceptron AND

The two-moon problem

The two-moon problem is a classic dataset in the field of machine learning used to illustrate the limitations of linear classifiers and the importance of models capable of handling non-linear relationships in data.

The moon labeled "Region A" is located symmetrically with respect to the y-axis, while the moon labeled "Region B" is displaced to the right of the y-axis by an amount equal to the radius r and below the x-axis by a distance d. The two moons have identical parameters:

$$r = 10 \qquad w = 6$$

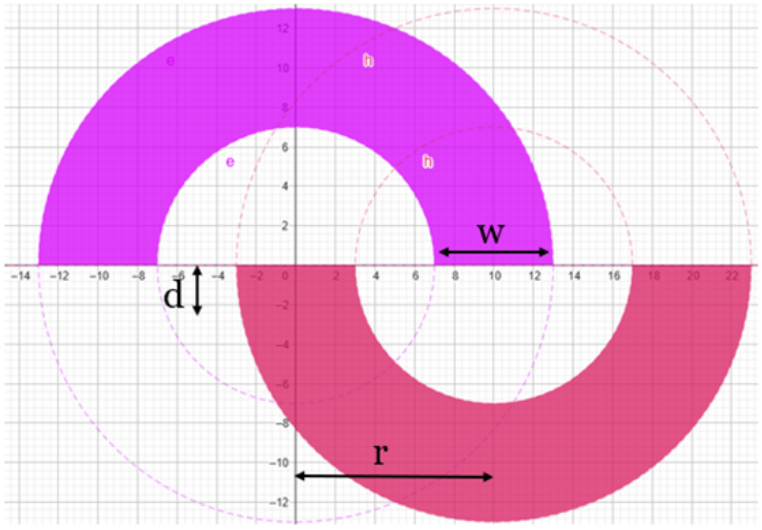


Figure 5: The two moons

The training sample T consists of 1,000 pairs of data points, each pair consisting of one point chosen randomly from region A and one point chosen from region B. There are 2,000 points in total.

The two-moon problem

```
def insideMoonA(x, y):
    if (49 <= x**2 + y**2) and (x**2 + y**2 <= 169) and y > 0:
        return 1
    else:
        return 0

def insideMoonB(x, y):
    if (49 <= (x - 10)**2 + (y - 0)**2) and ((x - 10)**2 + (y - 0)**2 <= 169) and y < -0:
        return 1
    else:
        return 0

def line_separator(x, w, b):
    return (-w[0] * x - b) / w[1]

# Generar puntos dentro de la región
num_points = 1000
moonAPointsTraining = []
moonBPointsTraining = []

moonAPointsTest = []
moonBPointsTest = []

while len(moonAPointsTraining) < num_points:
    x = np.random.uniform(-13, 14)
    y = np.random.uniform(0, 14)

    if insideMoonA(x, y):
        moonAPointsTraining.append([x, y])

while len(moonBPointsTraining) < num_points:
    x = np.random.uniform(-3, 24)
    y = np.random.uniform(-14, 0)

    if insideMoonB(x, y):
        moonBPointsTraining.append([x, y])

while len(moonAPointsTest) < num_points:
    x = np.random.uniform(-13, 14)
    y = np.random.uniform(0, 14)

    if insideMoonA(x, y):
        moonAPointsTest.append([x, y])

while len(moonBPointsTest) < num_points:
    x = np.random.uniform(-3, 24)
    y = np.random.uniform(-14, 0)

    if insideMoonB(x, y):
        moonBPointsTest.append([x, y])

trainingA = np.array(moonAPointsTraining)
trainingB = np.array(moonBPointsTraining)
```

```
# Combine test data
X_train = np.concatenate((trainingA, trainingB), axis=0)
# Create the corresponding labels
y_train = np.array([1]*len(trainingA) + [0]*len(trainingB))

# Mix data and labels in the same way
indices = np.arange(X_train.shape[0])
np.random.shuffle(indices)
X_train = X_train[indices]
y_train = y_train[indices]

# Combine test data
X_test = np.concatenate((testA, testB), axis=0)
# Create the corresponding labels
y_test = np.array([1]*len(testA) + [0]*len(testB))

# Mix data and labels in the same way
indices = np.arange(X_test.shape[0])
np.random.shuffle(indices)
X_test = X_test[indices]
y_test = y_test[indices]


perceptron = Perceptron(iterations=100, eta=0.1, random_state=1)
perceptron.fit(X_train, y_train)
y_pred = perceptron.predict(X_test)

accuracy = np.mean(y_pred == y_test)
print(f'Accuracy: {accuracy * 100:.2f}%')

plot_decision_regions(X_test, y_test, perceptron, resolution=0.05)
```

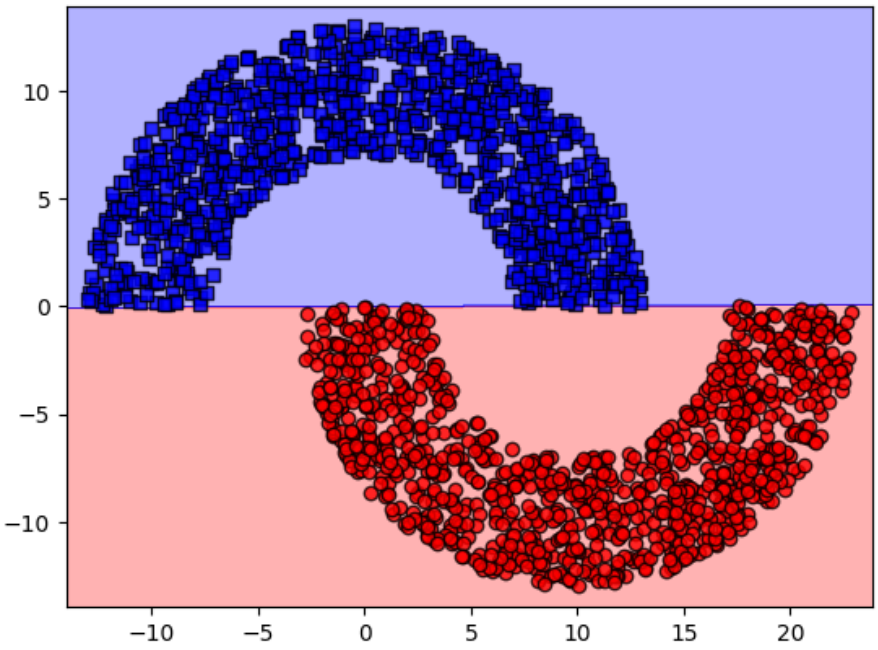


Figure 6: Result

References

[1] Haykin, S. (2009). *Neural networks and learning machines* (3rd ed.). Pearson.