Tutorial Series: Programming Artificial Neural Networks Step by Step

1: My First Perceptron

(Analyzed and Explained with a Practical Sense)

with **Python**

By: Eric Joel Barragán González

Table of Contents

Tutorial Series:

Programming Artificial Neural Networks Step by Step with Python

1: My First Perceptron

(Analyzed and Explained with a Practical Sense)

Full Code in 87 lines

By: Eric Joel Barragán González

Intent of the Series:

With this series of books, I intend to share my experience in working with Artificial Neural Networks (ANN), which I acquired when looking for their application in my projects and find myself with abundant and good theoretical material, with mathematical formulas, but impractical, very laborious to apply, or simply incomplete examples that only showed results and not implementation; Without counting the hours to understand it before applying it. So with this series of books I intend to cover the practical part with proven examples, and with each book, you can experiment with your own code and begin to adapt them to your projects, be it a robot or an application for decision making.

So I focus on guiding the reader who wants to experiment with the fascinating world of Artificial Neural Networks by coding it by itself, and having difficulty doing so, which I intend to facilitate with at least one complete example by Book.

Learn by doing. With the Series begins copying, and with your own experience, you will learn to:

1. Evaluate the needs and resources of your project.
2. Select which type of ANN to use in your project.
3. Design your own ANN topology.
4. Parametrize the topology and facilitate its improvement dynamically, with less recoding.
5. Configure your ANN according to your needs.
6. Distinguish which are the parts that consume more time of computation, to produce a topology and final code of efficient performance, with controlled memory consumption.

By sharing comments from my experience, to help you assimilate your experience more easily. The world of ANN is in its infancy, and what is known has been learned to trial and error, relying a little on mathematical theory, but especially on biological inspiration and the neural networks of the brains studied; So if you want to learn, I think, you must be willing to risk being wrong and maybe even manage to discover something that no one else has found and also contribute to this field.

With each book I will cover at least one complete, functional and proven example, explaining step by step the algorithm, showing its pseudocode to facilitate its implementation in other languages, and its coding in Python language, as it is not only easy to learn and understand, It is also widely used, and for its clarity, helps you to better understand the algorithms; On the other hand a parallel series in C language is being developed that is widely used in electronics and PC projects, generates efficient executables, that in large-scale RNA, or resource-reduced projects, is important.

I also recommend exercises in which I guide you, so that in your first experiments you know that change and where to expect affect, and soon you can refine your knowledge by your own hand and experience.

Not necessarily the examples will be like the authors of each moment in the history of neural networks, as it is not the purpose of the Series. But inspired by these we will take a sample example tour so that you can know the main contributions and incorporate them to your knowledge and experience, and after completing this Series you will be able to design your own ANN, to incorporate new knowledge with greater ease and even make your own contributions in your projects.

Table of Contents

Themes from the Series Books:

---

1. Simple Perceptron with Linear Function
2. Simple Perceptron with Bias and Linear Function
3. Simple Perceptron with Implicit Threshold and Linear Function
4. Simple Perceptron with Implicit Threshold and Nonlinear Function
5. Monolayer Perceptron with Sigmoid Function
6. Bilayer Perceptron with Backpropagation and Sigmoid Function
7. Trilayer Perceptron with Backpropagation and Sigmoid Function
8. Trilayer Perceptron with Backpropagation and Alternate Sigmoid Function
9. Multilayer Perceptron with Branches
10. Multilayer Perceptron with Overlapping Branches
11. Multilayer Perceptron with Convolution
12. Perceptron with Reinforcement Learning to Activate Robots
13. Perceptron with Associative Memory for Classification of Images
14. Perceptron for Character Recognition
15. Perceptron for Voice Recognition
16. Perceptron for Image Recognition
17. Evolutionary Perceptron for Accelerating Learning

Table of Contents

What This Book Covers:

The Perceptron of a single neuron, with an example as simple as possible, but complete, functional, tested, and according to our didactic objective; which, with its limitations, was the first algorithm that sought to imitate biological neurons, and although it was useful, it does not compare with the achievements of the current multi-layered ANN that can become self-organizing and self-adjusting to recognize voice, images, video, control movements and make decisions.

It touches on a bit of history, contributions, limitations, its usefulness, and explains the algorithm with a simple, complete, step-by-step explanation of its proven Python code and Pseudocode; accompanied by comments from my experience, and suggested exercises to direct your initial experience and notes how they affect the suggested changes to your code.

Table of Contents

Requirements to Follow the Book:

1. Knowing to Program in the Language of Your Choice.
2. With Language Chosen, a Machine with a Development Environment Ready.
3. Be willing to experiment.
4. Basic knowledge of mathematical logic.

Table of Contents

How to Follow This Book:

The Book I have thought to follow from beginning to end for those who in Percetrones have little experience, or want a little reflection to better understand what is happening, and do not want to settle for having found how to configure their Perceptions only in trial and error. However the names of the Points that I present you are descriptive, so that you distinguish if it is something that you do not need to read and continue with the next point. I recommend you codify your Perceptron in (Point 6.7), where it is in the Book, so that you continue reading and doing the exercises that I propose in the following Points.

Of the Pseudocode: It follows exactly the same order and logic as the Code in Python, in addition it has the same comments, that you will find them starting with "#". The description will find it in a thinner letter; so that although the Python language is not your choice, it can help you understand the algorithm and facilitate its coding.

When reading on a small screen device, I suggest that at least when you see the pseudocode or the code, put the screen horizontally, so that with greater width, fit the complete lines and do not go to the next row.

Table of Contents

The Next Book:

---

2 Simple Perceptron with Bias. We will work with the element that allows the Simple Perceptron to learn a greater variety of cases, without increasing too much the complexity of calculations.

Table of Contents

1 Introduction:

A *Neural Network* is a set of interconnected neurons, capable of fulfilling a purpose; *Biologicals* constitute *Organs* of the central nervous system of a living being, which in turn control organs, such as muscles, in layers that allow it with a thought to manipulate an object, where each of these layers is concerned with a level of complexity and with relative simplicity can perform complex tasks; or even abstractly make decisions. The *Artificial* constitute *Perceptrones* in a CPU, and can make decisions to handle certain complexity, such as handling a robot, recognizing an image, interpreting messages from a voice or analyzing a database.

In the case of this Book we will work with a *Simple Perceptron*, which is a single *Artificial Neuron*. It has a defined number of *Inputs* and a single *Output*. His *Memory* is a series of *Synaptic Weights*, which correspond to one for each *Input*; in addition its *Output* has defined a *Threshold*, which is the minimum required for its result to be positive (1), otherwise it will be negative (0). It was inspired by the *Biological Neurone*, which is known to have a similar operation, where each *Dendrite* receives a chemical input, which the *Dendrite* transforms into an *Electric Impulse*, with an intensity proportional to the memory that conserves (*Synaptic Weight*), which Is added to the other *Dendrites* and the *Soma* sends it to the *Axon*, if it exceeds the *Threshold*, gives its *Output*, which is finally the combined electrochemical result that the *Axon* delivers, with an intensity proportional to the result of the process.

Neural Networks offer advantages against sequential programming, as it is that:

1. Greater degree of parallelism in operations.
2. Learning capacity.
3. Generalization capacity.
4. Adaptability.
5. Fault tolerance.

Table of Contents

2 Background:

As a background we find that in 1943 Warren McCulloch, a neurophysiologist, and Walter Pitts, a mathematician, launched a theory of how neurons work by developing a neural network model for electrical circuits.

In 1957 Rosenblatt began to develop Perceptron, which is the oldest neural network, which was already able to recognize patterns, generalizing and even overcoming noise or recognizing similar patterns without having seen them before. In 1959 he developed the Theory of Convergence of the Perceptron, showing in his Book that, even starting from random initial values in their weights, the Perceptron converged towards a finite state.

Take into account that this is the time of the first commercial computers, built with vacuum tubes, fed with punch cards and programmed in machine language. The first commercial computer was manufactured by 1950, UNIVAC with a memory of 1,000 words and a speed of only a few thousand operations per second, and shortly after the IBM 650 that already used memory of magnetic drum, that was so popular that they were sold several hundred.

Table of Contents

3 Contributions:

Rosenblatt included a learning rule with which the Perceptron tends to converge to a finite state with the correct weights to solve the problem, if any, since it can learn and solve it even if the initial weights are random.

Another contribution is that Perceptron is so simple, both in its calculations and in the data it handles, that it is easier to analyze and understand what is happening; which we can not do with complex Perceptrones, but this does not stop us from being useful and understanding with it, to apply it to the most complex.

Table of Contents

4 Limiters:

In 1969 Marvin Minsky and Seymour Papert mathematically demonstrated that perceptrons are only capable of solving linearly separable problems, leaving non-linear functions out of their capacity. What time later was solved by adding layers; which we will cover in other Books Series.

Table of Contents

5 Utility:

It can recognize linearly separable patterns, patterns similar to those trained and has a small tolerance to noise.

Table of Contents

6 Algorithm Step by Step Explanation:

To explain the Algorithm I separate it in the processes that compose it:

1. Perceptron Calculation
2. Calculation of the Error
3. Adjustment Calculation
4. Case Calculation Cycle
5. Case Upload to be Solved
6. Presentation on Screen

Each process explains its essential operation, then ready in a box the part of the Pseudocode that corresponds to it, where in detail and with the same structure of the code each step of the process is explained, and for better reference with the same comments as the code; then in another box I show the part of the corresponding code, with the same line numbers of the code, as the complete code, implemented in Python. Then complementary comments of the explanation and in the required cases support tables that help to clarify. More details of the operation, explanation of variables and exercises, will be covered in the following Points. Since you can read this Book on a small screen device, both the Pseudocode and the Code in Python, I put them in Tables, so that the line number for the Code and a stripe for the Pseudocode, will indicate that which on your right appears, belongs only to that line of code.

Table of Contents

6.1 Perceptron Calculation

It processes each Input by multiplying it by its *Weight*, adds each of these products, and compares them against the *Threshold* of the *Neuron*, if it is higher than the *Threshold* results in (1), otherwise (0). The synaptic weights are adjustable, it is where the neuron stores the learned, after adjusting them each time it makes an error, it reaches a set of synaptic weights, which when processed, produce the expected result, and there is no error for any case, then it is considered that he has already learned to solve each case of assigned training.

```
Sum = (Input[1] * Weight[1]) + (Input[2] * Weight[2]) + . . . + (Input[n] *
Weight[n])

Si (Sum > Threshold) Then

    Axon = 1

Else

    Axon = 0
```

Table of Contents

6.1.1 Pseudocode

```
- # Array: Memory Weights
- declares array type float called: Wgh of [2]

- # Arrays: 4 Example Cases, "AND Table"
- declares array type float called: Inp of [4][2]

-   # Constants of the Rigid Parameters
-   declares variable type float called: Thr
-   puts 2.0 in Thr

-   # Work Variables
-   declares variable type float called: Axn
-   declares variable type float called: Sum

-   # Start Pseudo-random of the 2 Weights
-   set Wgh[0] a random float value between (-1 y 1)
-   set Wgh[1] a random float value between (-1 y 1)

-       # Perceptron Calculation for the Case
-       multiply Inp[0] by Wgh[0]
-       and multiply Inp[1] by Wgh[1]
-       sum results and puts it in Sum

-       # Comparison against Threshold
-       If Sum Exceeds Thr:
-         puts 1 in Axn # Threshold Exceeded
-       else case:
-         puts 0 in Axn # Threshold Not Exceeded
```

Table of Contents

6.1.2 Code in Python

```python
# Array: Memory Weights
Wgh = [0.0, 0.0]

# Arrays: 4 Example Cases, "AND Table"
Inp = []

# Constants of the Rigid Parameters
Thr = 2.0

# Start Pseudo-random of the 2 Weights
Wgh[0] = random.uniform(-1.0, 1.0)
Wgh[1] = random.uniform(-1.0, 1.0)

      # Perceptron Calculation for the Case
      Sum = Inp[i][0] * Wgh[0] + Inp[i][1] * Wgh[1]
      if Sum > Thr: # Comparison against Threshold
        Axn = 1 # Threshold Exceeded
      else:
        Axn = 0 # Threshold Not Exceeded
```

Table of Contents

6.2 Error Calculation

To learn the *Perceptron*, compare the *Obtained Result* against *Expected Output*, subtracting the *Obtained Result* from the *Expected Output*, assigning this difference to the *Error*. Since both variables only work either with (1) or with (0), when the result is expected the Error will be (0); when the Perceptron result is wrong the Error will be either (1) or (-1).

Error = Output - Axon

Table of Contents

6.2.1 Pseudocode

```
- # Arrays: 4 Example Cases, "AND Table"
- declares array type float called: Out of [4]

-   # Work Variables
-   declares variable type float called: Axn
-   declares variable type float called: Err

-       # Case Error Calculation
-       Axn(resulting) is subtracted from Out(expected output) and
  assigned to Err
```

Table of Contents

6.2.2 Code in Python

```
# Arrays: 4 Example Cases, "AND Table"
Out = []

      # Case Error Calculation
      Err = Out[i] - Axn
```

Table of Contents

6.2.3 Error Table

Output (expected)	**Axon** (obtained)	**Error** (calculated)
0	0	0
0	1	-1
1	0	1
1	1	0

Table of Contents

6.3 Calculations of Adjustments

The error provides with its sign, the direction toward which we must adjust the weights with respect to the calculated case, either add or decrease, and since we do not know the exact magnitude for the solution, what we will look for is an approximation, So that the intersection or convergence of the adjustments of the different Cases of the training will be what allows to have some Synaptic Weights that solve for all the Cases, which we will calculate for each one using:

1. The direction that gives us the *Perceptron Error*.
2. The *Weight* of each Input.
3. The proportion given by the *Learning* factor.

And the result is added or subtracted, according to the sign, to the weight we occupy for the calculation, obtaining the adjusted weight, which we assign to the same weight variable.

```
If Error
  Weight[1] = Weight[1] + (Learning * Error * Input[1])
  Weight[2] = Weight[2] + (Learning * Error * Input[2])
  . . .
  Weight[n] = Weight[n] + (Learning * Error * Input[n])
```

Table of Contents

6.3.1 Pseudocode

```
- // Array: Memory Weights
- declares array type float called: Wgh of [2]

- // Arrays: 4 Example Cases, "AND Table"
- declares array type float called: Inp of [4][2]

-   // Constants of the Rigid Parameters
-   declares variable type float called: Lrn
-   puts 1.0 in Lrn

-   // Work Variables
-   declares variable type float called: Err
-   declares variable type int called: NEr

-       // Weight Adjustment with Case
-       If Err is different from 0: // Adjusts if there is an Error
-         Increase NEr in 1 // Increase the Error counter

-         // It is performed for each Weight with its corresponding
  Input
-         multiply Lrn by Err by Inp[0]
-         Add the product to Wgh[0] and put it in Wgh[0]

-         multiply Lrn by Err by Inp[1]
-         Add the product to Wgh[1] and put it in Wgh[1]
```

Table of Contents

6.3.2 Code in Python

```python
# Array: Memory Weights
Wgh = [0.0, 0.0]

# Arrays: 4 Example Cases, "AND Table"
Inp = []

# Constants of the Rigid Parameters
Lrn = 1.0

      # Weight Adjustment with Case
      if Err != 0: # Adjusts if there is an Error
        NEr += 1 # Increase the Error counter
        # It is performed for each Weight with its corresponding Input
        Wgh[0] += Lrn * Err * Inp[i][0]
        Wgh[1] += Lrn * Err * Inp[i][1]
```

The previous calculations are made for each case that the *Perceptron* is intended to learn, and the *Errors* are counted, if they are (0) considered solved, or learned to solve all the *Cases*, and it is no longer necessary to adjust the *Weights*; otherwise the *Weights* are adjusted for each *Case* that presents *Error* and a more cycle is put to test, in each cycle the *Weights* will be adjusted, approaching the solution until reaching it, if there is at least one solution for the configuration that Is tested.

Convergence is an area within which the optimal solution is found, which is not usually obtained by this algorithm, it is not its nature, especially because its execution is stopped and its Weights are no longer adjusted when they resolve all Cases. We do not regularly find the optimal solution, but a good one.

Table of Contents

6.4 Cycle of Attempts

It is composed of a structure of two cycles; the internal cycle that goes through the 4 *Cases* that we want the *Perceptron* solves, testing if in each case the expected response is obtained, and as soon as it stops having *Errors*, resolving all *Cases*, it will stop making adjustments and will leave both the *Internal* cycle and the *External* cycle.

The number of external cycles is estimated to be large enough to give the algorithm the opportunity, that with the accumulated adjustments, solution to all *Cases*, for this example it is enough with 1,000 iterations, but as we will include *Cases* which can not solve, we will exaggerate 10,000 iterations, which will allow us to see what happens and there is no doubt. As we mentioned before if it is solved, the external cycle will end, so it will only complete the 10,000 cycles when it does not find a combination of *Synaptic Weights* that solve all the *Cases*.

On the other hand, the number of cycles needed to solve, will not be for each run equal, will depend both on the *Parameters* we define, and on the values taken by the *Initially Weights*, which, as pseudorandom, can vary considerably, especially if the *Initial Weights* are proportionally several times larger than the *Threshold*; hence the importance of handling values in *Weights* in ranges lower than the *Threshold*.

Table of Contents

6.4.1 Pseudocode

```
-   # Iteration Control Variables
-   declares variable type int called: i
-   declares variable type int called: j
-   declares variable type int called: k

-   # Cycle of Attempts
-   # Parameters for 10,000 Attempts
-   puts 0 in j
-   puts 10,000 in k
-   Inicia Ciclo # Start Cycle of Attempts
-     Increase j in 1 # Increment Counter of Attempts

-     # Display On Screen Attempt to Calculate

-     # Cycle of Cases, by Intent
-     puts 0 in NEr # Restart Error Counter
-     Starts Cycle # Starts Case Cycle
-       Passes i through each of 0 to 3 # Scroll through the 4 Cases

-       # Perceptron Calculation for the Case

-       # Case Error Calculation

-       # Weight Adjustment with Case

-       # Display on Screen Inputs and Output

-       # Display Weights Adjusted

-     # Ends Case Cycle

-     # Displays the Number of Errors of the Attempt

-     # Ends Early if No Errors
-     If error counter NEr is 0
-         Leaves the Cycle of Attempts
-   # Ends Cycle of Attempts

-   # Termination of Program
```

Table of Contents

6.4.2 Code in Python

```python
# Cycle of Attempts
j = 0; k = 10000 # Parameters for 10,000 Attempts
while True: # Start Cycle of Attempts
  j += 1 # Increment Counter of Attempts

  # Increment Counter of Attempts

  # Cycle of Cases, by Intent
  NEr = 0 # Restart Error Counter
  for i in range(4): # Starts Case Cycle

      # Perceptron Calculation for the Case

      # Case Error Calculation

      # Weight Adjustment with Case

      # Display on Screen Inputs and Output

      # Display Weights Adjusted

  # Ends Case Cycle

  # Displays the Number of Errors of the Attempt

  # Ends Early if No Errors
  if NEr == 0:
    break

  if j >= k: # Ends Cycle of Attempts
    break

# Termination of Program
```

The previous calculations are made for each case that the *Perceptron* is intended to learn, and the errors are counted, if they are (0) considered solved, or learned to solve all the cases, and we do not need to adjust the *Weights* anymore. Otherwise the *Weights* are adjusted for each case that presents *Error*, and a more cycle is put to test, in each cycle the *Weights* will be adjusted approaching the solution, until it is reached, if there is at least one solution for the configuration which is tested.

Table of Contents

6.5 Example Load

To exemplify what a *Simple Perceptron* can learn, we will work with the *AND Truth Table*. In (Point 9) other examples that I propose to work will be listed, and we will distinguish which ones can be solved, which ones do not and why.

Table of Contents

6.5.1 Pseudocode

```
- # Arrays: 4 Example Cases, "AND Table"
- declares array type float called: Inp of [4][2]
- declares array type float called: Out of [4]

-   # Load 4 Example Cases, "AND Table"
-   # Case 1
-   puts 0 in Inp[0][0]
-   puts 0 in Inp[0][1]
-   puts 0 in Out[0]

-   # Case 2
-   puts 0 in Inp[1][0]
-   puts 1 in Inp[1][1]
-   puts 0 in Out[1]

-   # Case 3
-   puts 1 in Inp[2][0]
-   puts 0 in Inp[2][1]
-   puts 0 in Out[2]

-   # Case 4
-   puts 1 in Inp[3][0]
-   puts 1 in Inp[3][1]
-   puts 1 in Out[3]
```

Table of Contents

6.5.2 Code in Python

```python
# Arrays: 4 Example Cases, "AND Table"
Inp = []
Out = []

# Load 4 Example Cases, "AND Table"
# Case 1
Inp += [[0,0]]
Out += [0]
# Case 2
Inp += [[0,1]]
Out += [0]
# Case 3
Inp += [[1,0]]
Out += [0]
# Case 4
Inp += [[1,1]]
Out += [1]
```

Table of Contents

6.5.3 AND Truth Table

	Input 1	Input 2	Output
Case 1	0	0	0
Case 2	1	0	0
Case 3	0	1	0
Case 4	1	1	1

Table of Contents

6.6 Presentation on Screen

So that we can evaluate the performance that is taking the *Perceptron* step by step, from the starting point to its solution or inability to do so; and we can experiment with different configurations, being able to compare them to learn from them, we will show on screen the changes that the variables have in each cycle.

Now that the form is very simple, seeking that the information shown is sufficient, and that you have the idea of how to do it as a starting point for you to personalize it; also taking into account that you may be using a different language, the example contains the essential and the form is appropriate rather to your liking.

Table of Contents

6.6.1 Pseudocode

```
-   # Display 2 Initial Weights
-   display ("Initial: W1: Wgh[0] W2: Wgh[0] (enter)(enter)")

-   # Start Cycle of Attempts

-     # Display On Screen Attempt to Calculate
-     display ("Cycle:j - - - - - - - - -(enter)")

-     # Starts Case Cycle

-       # Display on Screen Inputs and Output
-       display ("- Case i I1: Inp[i][0] I2: Inp[i][1] O:Out[i]
  Axn:Axn")
-       If Err different from 0 # If there is Error
-         change font color to red; # Show Errors in Red
-       display ("Err: Err")
-       Changes font color to previous;

-       # Shows Weights Adjusted
-       display ("Adjusts: W1: Wgh[0] W2: Wgh[1](enter)");

-     # Ends Case Cycle

-     # Displays the Number of Errors of the Attempt
-     display ("NEr Errors - - - - - - - - -(enter) (enter)")

-   # Ends Cycle of Attempts

-   # Terminación de Programa
```

Table of Contents

6.6.2 Code in Python

```python
# Display 2 Initial Weights
print "\033[32m Initial: W1:%f W2:%f\n\n" % (Wgh[0], Wgh[1])

# Start Cycle of Attempts

  # Display On Screen Attempt to Calculate
  print "Cycle:%d - - - - - - - - - -" % (j)

  # Starts Case Cycle

      # Display on Screen Inputs and Output
      print ("- Case %d I1:%d I2:%d O:%d Axn:%f Sum:%f "
      % (i, Inp[i][0], Inp[i][1], Out[i], Axn, Sum)),
      if Err != 0: # If there is Error
        print "\033[33m", # Show Errors in Red
      print "Err: %f\033[32m" % (Err)

      # Shows Weights Adjusted
      print " Adjusts a: W1:%f W2:%f" % (Wgh[0], Wgh[1])

  # Ends Case Cycle

  # Displays the Number of Errors of the Attempt
  print "%d Errors - - - - - - - - -\n" % (NEr)

  # Ends Cycle of Attempts

# Terminación de Programa
```

Table of Contents

6.7 Full Algorithm, in Pseudocode and C Language

Then the *Complete Algorithm* is ready to be implemented in another language, or copied and compiled in C for you to perform your tests. With this you will have your *"Hello World"* speaking in neural networks; and in the following points, the explanation is extended and I add elements for you to extend this experience; other examples, comments, and suggested exercises to help you understand *Simple Perceptron*.

Table of Contents

6.7.1 Pseudocode Complete

```
- // Array: Memory Weights
- declares array type float called: Wgh of [2]

- // Arrays: 4 Example Cases, "AND Table"
- declares array type float called: Inp of [4][2]
- declares array type float called: Out of [4]

- // Main Procedure
- Main Procedure Begins
-   // Constants of the Rigid Parameters
-   declares variable type float called: Lrn
-   declares variable type float called: Thr
-   puts 1.0 in Lrn
-   puts 2.0 in Thr
-   // Iteration Control Variables
-   declares variable type int called: i
-   declares variable type int called: j
-   declares variable type int called: k
-   // Work Variables
-   declares variable type float called: Axn
-   declares variable type float called: Sum
-   declares variable type float called: Err
-   declares variable type int called: NEr

-   // Load 4 Example Cases, "AND Table"
-   // Case 1
-   puts 0 in Inp[0][0]
-   puts 0 in Inp[0][1]
-   puts 0 in Out[0]
-   // Case 2
-   puts 0 in Inp[1][0]
-   puts 1 in Inp[1][1]
-   puts 0 in Out[1]
-   // Case 3
-   puts 1 in Inp[2][0]
-   puts 0 in Inp[2][1]
-   puts 0 in Out[2]
-   // Case 4
-   puts 1 in Inp[3][0]
-   puts 1 in Inp[3][1]
-   puts 1 in Out[3]

-   // Start Pseudo-random of the 2 Weights
-   set Wgh[0] a random float value between (-1 y 1)
-   set Wgh[1] a random float value between (-1 y 1)

-   // Display 2 Initial Weights
-   display ("Initial: W1: Wgh[0] W2: Wgh[0] (enter)(enter)")

-   // Cycle of Attempts
-   // Parameters for 10,000 Attempts
-   puts 0 in j
-   puts 10,000 in k
-   Inicia Ciclo// Start Cycle of Attempts
-     Increase j in 1 // Increment Counter of Attempts

-     // Display On Screen Attempt to Calculate
-     display ("Cycle:j - - - - - - - - -(enter)")

-     // Cycle of Cases, by Intent
-     puts 0 in NEr // Restart Error Counter
-     Starts Cycle // Starts Case Cycle
-       Passes i through each of 0 to 3 # Scroll through the 4 Cases

-       // Perceptron Calculation for the Case
-       multiply Inp[0] by Wgh[0]
-       and multiply Inp[1] by Wgh[1]
-       sum results and puts it in Sum

-       // Comparison against Threshold
-       If Sum Exceeds Thr:
-         puts 1 in Axn # Threshold Exceeded
-       else case:
-         puts 0 in Axn # Threshold Not Exceeded
-       // Case Error Calculation
-       Axn(resulting) is subtracted from Out(expected output) and
  assigned to Err

-       // Weight Adjustment with Case
-       If Err is different from 0: // Adjusts if there is an Error
-         Increase NEr in 1 // Increase the Error counter

-         // It is performed for each Weight with its corresponding
  Input
-         multiply Lrn by Err by Inp[0]
-         Add the product to Wgh[0] and put it in Wgh[0]

-         multiply Lrn by Err by Inp[1]
-         Add the product to Wgh[1] and put it in Wgh[1]

-       // Display on Screen Inputs and Output
-       display ("- Case i I1: Inp[i][0] I2: Inp[i][1] O:Out[i]
  Axn:Axn")
-       If Err different from 0 // If there is Error
-         change font color to red; // Show Errors in Red
-       display ("Err: Err")
-       Changes font color to previous;

-       // Display Weights Adjusted
-       display ("Adjusts: W1: Wgh[0] W2: Wgh[1](enter)");

-     // Ends Case Cycle

-     // Displays the Number of Errors of the Attempt
-     display ("NEr Errores - - - - - - - - -(enter) (enter)")

-     // Ends Early if No Errors
-     If error counter NEr is 0
-         Leaves the Cycle of Attempts

-   Continuous Cycle While j is less than k // Ends Cycle of
  Attempts
```

```
- // Termination of Program
```

Table of Contents

6.7.2 Code in Python Complete

```python
# Complement to generate random
import random

# Array: Memory Weights
Wgh = [0.0, 0.0]

# Arrays: 4 Example Cases, "AND Table"
Inp = []
Out = []

# Procedimiento Principal
# Constants of the Rigid Parameters
Lrn = 1.0
Thr = 2.0

# Load 4 Example Cases, "AND Table"
# Case 1
Inp += [[0,0]]
Out += [0]
# Case 2
Inp += [[0,1]]
Out += [0]
# Case 3
Inp += [[1,0]]
Out += [0]
# Case 4
Inp += [[1,1]]
Out += [1]

# Start Pseudo-random of the 2 Weights
Wgh[0] = random.uniform(-1.0, 1.0)
Wgh[1] = random.uniform(-1.0, 1.0)

# Display 2 Initial Weights
print "\033[32m Initial: W1:%f W2:%f\n\n" % (Wgh[0], Wgh[1])

```

```python
# Cycle of Attempts
j = 0; k = 10000 # Parameters for 10,000 Attempts
while True: # Start Cycle of Attempts
  j += 1 # Increment Counter of Attempts

  # Display On Screen Attempt to Calculate
  print "Cycle:%d - - - - - - - - -" % (j)

  # Cycle of Cases, by Intent
  NEr = 0 # Restart Error Counter
  for i in range(4): # Starts Case Cycle

      # Perceptron Calculation for the Case
      Sum = Inp[i][0] * Wgh[0] + Inp[i][1] * Wgh[1]
      if Sum > Thr: # Comparison against Threshold
        Axn = 1 # Threshold Exceeded
      else:
        Axn = 0 # Threshold Not Exceeded

      # Case Error Calculation
      Err = Out[i] - Axn

      # Weight Adjustment with Case
      if Err != 0: # Adjusts if there is an Error
        NEr += 1 # Increase the Error counter
        # It is performed for each Weight with its corresponding
Input
        Wgh[0] += Lrn * Err * Inp[i][0]
        Wgh[1] += Lrn * Err * Inp[i][1]

      # Display on Screen Inputs and Output
      print ("- Case %d I1:%d I2:%d O:%d Axn:%f Sum:%f "
      % (i, Inp[i][0], Inp[i][1], Out[i], Axn, Sum)),
      if Err != 0: # If there is Error
        print "\033[33m", # Show Errors in Red
      print "Err: %f\033[32m" % (Err)
      # Display Weights Adjusted
      print " Adjusts a: W1:%f W2:%f" % (Wgh[0], Wgh[1])

  # Ends Case Cycle

  # Displays the Number of Errors of the Attempt
  print "%d Errors - - - - - - - - -\n" % (NEr)

  # Ends Early if No Errors
  if NEr == 0:
    break

  if j >= k: # Ends Cycle of Attempts
    break

# Termination of Program
```

Table of Contents

7 Algorithm Characteristics:

In this point I explain some of the characteristics of this algorithm that distinguish it from others and limit it to solve some problems. According to several criteria under which other proposals of Perceptron algorithms have been made, is that classifications have been created.

Table of Contents

7.1 Supervised Learning

Analogous to biological learning, Simple Perceptron learns under supervision, in the sense that every time it is asked to perform its operation, it is evaluated and feedback, letting it know of each calculation, whether it is correct or not; so you can adjust the calculations and direct your learning trend to solve all the Case; when he gets the solution of all the cases of the example worked, he is also warned that he has solved them, and no longer requires learning, he has succeeded, and learning ends.

It is considered supervised because it is indicated what is the expected result, and it is feedback, if it has Error or not in each Case, from which also the adjustments to be made to arrive at this expected result are calculated. The other alternative as classification, would be with Non-Supervised Learning, which we will cover in another Book Series.

At the same time the Supervised Learning classification is subclassified in:

- Learning by Correction of Errors
- Reinforcement Learning
- Stochastic Learning

Our algorithm works with a **Correction of Errors**, since every time you commit an Error, action is taken to seek to correct it.

As for the algorithm, the supervision is implicit, so to distinguish it at the beginning is not obvious. I hope that with the previous explanations you understand it, and that in (Point 13), where in a function I completely eliminate supervision, I just made it clear.

Table of Contents

7.2 Type of Association between Inputs and Outputs

In contrast to ours, there are the auto-associative, in which their algorithm when performing the calculations, takes each combination of Inputs, to the Output with which they have more in common. In our case the Perceptron is **Heteroasciative**, in this the association is not free, but predetermined, so it requires supervision and feedback, so that the particular combination of Inputs, Learn to Associate to a particular Output.

Table of Contents

7.3 Architecture

Some of the characteristics with which they are classified depend on the way the neurons of the network interconnect, since only in this algorithm we manage a single neuron, they do not apply these classifications; and only applies *Monolayer* of a *Single Neurone*, from where it acquires the name of **Simple Perceptron**

Table of Contents

7.4 Type of Representation of Information

The ways in which the *Inputs*, *Outputs* and *Axon* information can be represented specifically are:

- Discrete
- Continuous
- Hybrid

Our *Perceptron* works with **Discrete** information, since we only represent either with (1) or with (0) any information of the *Inputs*, the one that the *Axon* throws, and with the *Expected Output* against which we compare it.

Table of Contents

7.5 Activation Function

Simple Perceptron is characterized by having a **Linear Activation Function**. Refers to the function that is applied to the *Sum* of the *Products* of each Input by its corresponding *Weight*. The value of this function of the *Sum* is that which is compared against the *Threshold*. In our case the algorithm does not apply any function to the *Sum*, which is equivalent to multiplying it by (1), which gives a linear relation.

Table of Contents

8 Main Variables and Constants:

Now that you know how the *Simple Perceptron* structure is composed, I'm going to explain how each variable and constant works in more detail.

Table of Contents

8.1 Inputs

They are the *Variables* where the information is stored, which represents the combinations that *Perceptron* must recognize. For this algorithm we work in binary form; and since we only work with (1) or with (0) the information we want the *Perceptron* to work, it must be coded in this way.

In the example we have worked on in this Book, the *Truth Table AND* has been represented, where simply (1s) represent true, and (0s) represent false. Although rendering to process, it is not always so easy and straightforward to encode. We will be working with other examples in the following Series Books, progressively adding more complexity.

Table of Contents

8.2 Outputs

They are the *Variables* where we store what we want to learn the *Perceptron*, in response to the combination of associated *Inputs*. Against what we will compare what has been learned, to verify if it is correct or adjustments are required in your *Weights*.

They represent the *Supervision* that is given during *Perceptron Learning*. Like *Inputs*, *Outputs* need to be encoded in binary. I recommend you, as in this example, before writing your code of other exercises that you propose, in the language of your choice, write on a spreadsheet, or even on paper, the *Tables* that you later load; this will allow you to facilitate checking that the information is correct, and does not create ambiguities, in that a single combination of *Inputs* only corresponds to an *Output*; the opposite has no problem, because as in the example that we work, several combinations of *Inputs* can correspond to the same *Output*.

Table of Contents

8.3 Weights

It is the *Variables* that store what is learned by the *Perceptron*, control the proportion of significance of each *Input*, to contribute to the result that exceeds or does not exceed the *Threshold* and result *Axon* contains for the *Case* a (1) or a (0), according to expect.

Unlike *Inputs* and *Outputs* that are *Discrete* and *Binary*, the *Weights* are *Continuous*, they can have a wide range of values, so they are stored in *Floating-point Variables*. Its value is progressive and gradually adjusted again and again, looking for that in combination with the Inputs, allow to solve each and every one of the *Cases* of the example worked. These are called *Convergence* values of the *Weights*.

It was previously thought that since the *Algorithm* is able to obtain at the values of *Convergence* (of solution), starting from different initial values of the *Weights*; that no matter the initial values given to the *Weights*, and some simply assigned (0). With experience we learned that it is important, and that starting with (0s) can be a problem; in terms of fixed and equal values can lead to solving in a few iterations in some cases, and to solve others requires many more attempts, in short, requiring more computation time to find the solution.

It was then concluded that it was better to start with pseudorandom values, but there are still different criteria to define the appropriate range for better results. So in another point, later, I recommend practice that will allow you to experiment and draw your conclusions from what range is convenient to use for the initial values.

Table of Contents

8.4 Sum

It is a *Temporary Variable*, which stores the result of the *Sums* of the products of each *Input* by its corresponding *Weight*. Its function is to clarify the algorithm, and to facilitate the presentation of the comparison of its value acquired in the calculation, with the *Threshold*.

Particularly in this simple example it is dispensable, but in more complex algorithms it is not, because in greater complexity, it not only gives clarity to the algorithm, besides flexibility. It is a *Continuous variable*, so it is defined as *floating-point* type.

Table of Contents

8.5 Threshold

It is the *Constant* that defines the point, from which the result of the calculation stored in the *Sum*, will cause the *Axon* to result in a (1) or a (0). It must have a midpoint in the range of values that the *Sum* function can acquire, so that it does not find a tendency to either, this symmetry will increase the possibility that the greater variety of cases of an example has equality of possibilities to be solved; otherwise it will solve some cases and it will be difficult for others, to the degree that simply some can not solve them.

Table of Contents

8.6 Axon

It is the *Variable* that stores the value resulting from the entire *Perceptron* calculation process. The one that is compared against the *Output* (supervision), with what can be known: or, that the calculation of the *Case* is correct, does not generate *Error*, and does not require adjustments; or otherwise, the calculation of the *Case* is not right, there is an *Error*, the *Weights* require adjustments, and the training is not over, and requires continuing. Like the *Output*, it only acquires the binary values or (1) or (0); the variable is *Discrete*.

Table of Contents

8.7 Error

It is the *Variable* that saves the difference between the expected and the obtained result; seeing it in another way, the difference between the *Case Output* and the *Axon* calculation. The variable is *Discrete*, it can only result in (-1, 0 or 1). As mentioned before (Table 6.2.3), it provides with its *Sign*, if not (0), the direction in which the *Weights* must be adjusted, to continue approaching the solution.

Table of Contents

8.8 Number of Errors

It is the *Variable* that contains the *Counter of Errors* of each attempt, so that in each *Case* of intent, when the *Output* is compared with the *Axon*, if they are not equal, it is considered *Error* and this counter is increased by one. An *Error* is enough so that the learning is not considered successfully learned, the *Weights* are adjusted and another attempt is made.

Table of Contents

8.9 Learning

It is the *Constant* that defines how much the values of the *Weights* are changed, which have (1) their corresponding *Input* values, in the combination that is calculated, to try to correct the *Error* in the attempt. The one who is apparently ignorant of the *Learning Factor*, what he is actually doing when he omit it, is to define it with a value of (1), which gives rise to jumps so large that at times the *Perceptron* can only end up oscillating around values of convergence without reaching them. The same thing can happen if the *Learn* parameter, explicitly used, is assigned large values for the worked example with respect to the *Threshold*, but this will become clearer in (Point 11.3) with the exercises I propose to you.

On the other hand, if we assign very small values, whenever there is a possible solution for our exercise, we will surely arrive at it, only that the approximations if they are so small, may require many more iterations, and with this, more computation time for to reach the values of the *Convergence Weights*.

A formula has not yet been defined in order to calculate the optimal value of *Learning*, so it is important the experience to reduce the time spent on tests and adjustments in the parameter. For this I propose the exercises of (Point 11.3) that will allow you to accumulate some of this experience.

Table of Contents

9 Other Examples to Work:

In addition to the example that already comes in the code, (Table of truth AND), I propose you work with (OR, NAND, NOR and XOR); so that you realize what the *Perceptron* is capable of solving, of what no, and why. In addition as a complement, we will reverse the representation of (true => 1 and false = 0), for (true => 0 and false = 1).

To apply it in the Python code we will only substitute the lines of (Point 6.5.2) (lines 17 to 28). Next you are ready the Tables of Truth, and its Representation or Inverted Codification:

Table of Contents

9.1 AND Truth Table

	Input 1	Input 2	Output
Case 1	0	0	0
Case 2	1	0	0
Case 3	0	1	0
Case 4	1	1	1

Table of Contents

9.2 OR Truth Table

	Input 1	Input 2	Output
Case 1	0	0	0
Case 2	1	0	1
Case 3	0	1	1
Case 4	1	1	1

Table of Contents

9.3 NAND Truth Table

	Input 1	Input 2	Output
Case 1	0	0	1
Case 2	1	0	1
Case 3	0	1	1
Case 4	1	1	0

Table of Contents

9.4 NOR Truth Table

	Input 1	Input 2	Output
Case 1	0	0	1
Case 2	1	0	0
Case 3	0	1	0
Case 4	1	1	0

Table of Contents

9.5 XOR Truth Table

	Input 1	Input 2	Output
Case 1	0	0	0
Case 2	1	0	1
Case 3	0	1	1
Case 4	1	1	0

* Now I put the Tables of Truth, with the representation or encoding inverted:

Table of Contents

9.6 AND Truth Table (reverse coding)

	Input 1	Input 2	Output
Case 1	1	1	1
Case 2	0	1	1
Case 3	1	0	1
Case 4	0	0	0

Table of Contents

9.7 OR Truth Table (reverse coding)

	Input 1	Input 2	Output
Case 1	1	1	1
Case 2	0	1	0
Case 3	1	0	0
Case 4	0	0	0

Table of Contents

9.8 NAND Truth Table (reverse coding)

	Input 1	Input 2	Output
Case 1	1	1	0
Case 2	0	1	0
Case 3	1	0	0
Case 4	0	0	1

Table of Contents

9.9 NOR Truth Table (reverse coding)

	Input 1	Input 2	Output
Case 1	1	1	0
Case 2	0	1	1
Case 3	1	0	1
Case 4	0	0	1

Table of Contents

9.10 XOR Truth Table (reverse coding)

	Input 1	Input 2	Output
Case 1	1	1	1
Case 2	0	1	0
Case 3	1	0	0
Case 4	0	0	1

Table of Contents

9.11 Comment

As you will realize when applying these new *Tables of Truth*, inverting the codification allows to solve *Tables* that with its previous representation could not, only that the cost is that the ones that resolved before, not now. Why? I'll explain it to you in the next Point.

Table of Contents

10 Why does it only solve some?:

As you will have noticed, when analyzing the information that the program shows with the runs, it is necessary, when you want an *Output* with value (1), that at least one of the *Inputs* of the *Case* that you want to solve, has the value of (1); since if all Inputs have the value of (0), the calculations could never result in the value of (1).

If we analyze the *Formula* by replacing all Inputs with (0):

1. Sum = (Input[1] * Weight[1]) + (Input[2] * Weight[2]) + . . . + (Input[n] * Weight[n])
2. Sum = (0 * Weight[1]) + (0 * Weight[2]) + . . . + (0 * Weight[n])
3. Sum = (0) + (0) + . . . + (0)
4. Sum = 0

So this algorithm has no way to solve this situation, as long as all the *Inputs* have the value of (0) the *Output* will be (0).

The solution, is the topic of the following Book of this Series: "2 Simple Perceptron with Bias"

Table of Contents

11 Experimentation with Parameters:

When you reach this Point, you already have a general idea of how *Perceptron Simple* works, I highly recommend that you perform the exercises I propose later, they will help you to strengthen your experience in the *Perceptron* configuration.

To do the experimentation I do not recommend using the *OR Table*, that when the two inputs are (0) and you expect to learn to be (0), any *Weight* multiplied by (0) will give (0) the rest of the *Cases* must give (1), and that it is enough that the *Weights* are greater than the *Threshold* so that they surpass it, without affecting the only *Case* that is different, where regardless of their value as we have seen, being multiplied by (0) will be (0). Instead I recommend you use *Table AND*, which is expected, taking *Inputs* in (1), will give (0) in its *Output* in a pair of *Cases*, and only *Output* in (1) is expected, when all its entries are (1); which is obtained if all the values of the *Weights* are less than the value of the *Threshold*, and only summing all the values of the *Weights* exceeds the value of the *Threshold*. Thus we are a little more demanding and is more useful for analysis.

We must take into account that this is an extremely simplified example, it allows us to give an idea of how they affect the parameters and their relation between them, but we must be careful in applying the conclusions we draw to other more complex *Perceptrones*, and especially with variants in its algorithms, hence the importance of knowing these variants, with their disadvantages and contributions, as we will have more opportunity to choose the right algorithm for our project. For this reason it is one of the objectives of the Series to give you an example of the main *Perceptron* algorithms.

Now as a starting point let us focus on a few issues that will allow us to have an idea of what we do when configuring *Perceptron*. We will focus on those we can modify: *Initial Weights Range*, *Learning* and *Threshold*.

1. The initial *Weights* may have positive, negative and preferably no (0) values.

2. *Threshold* and *Learning* can not take negative values, nor (0).

3. The value of the *Threshold*, as long as it is positive, can arbitrarily take any value.

4. The *Learning* value must be less than half the *Threshold* value.

5. In less iterations, the solution is found when *Learning* takes a value of half the value of the *Threshold*. (For this Algorithm).

6. In less iterations, the solution is found when the *Initial Weights* tend to values slightly lower than half the value of the *Threshold*, in its positive part of the range, and with the same magnitude for the negative part of the range.

7. In less iterations, the solution is found when the initial *Weights* have values different from each other, pseudorandom is recommended.

I speak of trends in *Perceptron* values, because although it may take other values, most will take the bias. It is necessary to run the program several times with the same parameters to identify these trends, because even parameters beyond the optimum can produce an optimal result; since it solves in the first attempt for all *Cases*, without requiring some adjustment, because the pseudorandom generator generated the values of convergence. That is why I underline the importance of experience, and to be more complete, comparisons with mathematical curiosity, what is happening and why. I am sharing observations to shorten your learning curve, but for knowledge to do its own, it is best to experiment and analyze it yourself.

Table of Contents

11.1 Break the Rules

Do not stay with doubts, try, experiment and check yourself what happens if you skip the rules I mentioned in the previous point. As I mentioned before, I recommend you use *Table AND* (true => 1, and false => 0). Let's start with those what simply do not allow *Perceptron* to function properly and observe:

1. With **Negative Learning**. The *Sum* is simply increased by moving away from the point of equilibrium, in which it could cause *Axon* to acquire the opposite value to the one he has with *Error*.
   - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
   - Threshold **(2)**.
   - Learning **(-1)**.

2. With **Negative Threshold**. Although the value of the *Threshold* does not deviate too much, it does not cross it to acquire the value that would give the correct value to the *Axon*.
   - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
   - Threshold **(-2)**.
   - Learning **(1)**.

3. With **Negative Learning and Negative Threshold**. As *Negative Learning* is, the sum value is increased away from the solution only in this case as the *Negative Threshold*, it moves away in the opposite direction.
   - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
   - Threshold **(-2)**.
   - Learning **(-1)**.

4. With **Zero Learning**. There will simply be *No Learning*, and values will remain as they started. As I told you about the initial pseudorandom values, you can touch the case in which it produce that the first one will solve, only that is unlikely to be repeated.
   - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
   - Threshold **(2)**.
   - Learning **(0)**.

5. With **Zero Threshold**. When the working range is eliminated, from (0 to 0), the values of *Sum* will begin to become smaller, but will not cross the *Threshold*.
   - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
   - Threshold **(0)**.
   - Learning **(1)**.

6. With **Learning and Zero Threshold**. To begin with, as we have already seen, with *Learning in Zero*, there will be *No Learning* and values will remain as they began; in addition to the *Threshold in Zero*, there would be nowhere to go. Something interesting is that you can modify the algorithm and work with values for the *Axon* of (-1 and 1) and for the *Threshold* of (0); I personal see an advantage to work with values for the *Axon* of (0 and 1), because when we have very robust *Perceptrones*, every *Input* with (0) is not required to perform calculations, since any multiplication will give zero, with which can save computing time.
    - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
    - Threshold **(0)**.
    - Learning **(0)**.

7. With **Learning equal to Threshold**. It produces oscillations that although they will not be far from the range for the crossing of the *Threshold*, it will not, and can not reach the values of *Convergence*.
    - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
    - Threshold **(2)**.
    - Learning **(2)**.

8. With **Learning greater than Threshold**. It will also produce an oscillation, where the values of *Sum* will be furthest from the *Threshold* value, as the difference between the value of *Learning* and *Threshold* is greater. Something interesting that we can see is that some values will approach *Zero*, and others the value of *Learning*.
    - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
    - Threshold **(2)**.
    - Learning **(4)**.

Table of Contents

11.2 Modifying the Ratio

The proportion that matters is the one between our parameters, and not with respect to the numbers themselves, which are only the representation of the system, the mathematical rules themselves do the rest. But this is clearer by doing so, remember to perform several runs of the program for each configuration, choose a number of runs, you do with each configuration, and notes so you can compare them, say 10 runs, It is clear what is happening; so let's observe.

1. **Initial Proportion** of the Representation. Note that although the number of attempts required to reach the response changes, it tends to be similar.
   - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
   - Threshold **(2)**.
   - Learning **(1)**.

2. **Reducing the Scale** of the Representation. Although they take values 10 times smaller, the tendency of the number of attempts necessary to reach the convergence values is the same.
   - Range of Weights **(-0.1 a 0.1)**. **random.uniform(-0.1, 0.1)**
   - Threshold **(0.2)**.
   - Learning **(0.1)**.

3. **Further reducing the Scale** of the Representation. 100 times lower than the initial values and the trend remains.
   - Range of Weights **(-0.01 a 0.01)**. **random.uniform(-0.01, 0.01)**
   - Threshold **(0.02)**.
   - Learning **(0.01)**.

4. **Increasing the Scale** of the Representation. Note that now that we have increased 10 times from the initial values, the trend of number of attempts has not changed.
   - Range of Weights **(-10 a 10)**. **random.uniform(-10.0, 10.0)**
   - Threshold **(20)**.
   - Learning **(10)**.

5. **Increasing further the Scale** of the Representation. And neither 100, nor 1,000, nor any other scale of numbers will make the trend change, as long as the ratio between the parameters themselves is maintained. For what I recommend you use the representation that suits your project, in this algorithm so simple, does not seem to make a difference, but in *Perceptrones* more complex and in combination with some projects in particular, we can facilitate coding and decoding, remembering that is not always so direct to discrete variables, and we must encode or decode to continuous variables.
   - Range of Weights **(-100 a 100)**. **random.uniform(-100.0, 100.0)**

- Threshold (**200**).
- Learning (**100**).

Table of Contents

11.3 Optimizing the Learning Value

Now let's work with the configurations that optimize Perceptron's work, in terms of *Learning* value. These depend partly on Perceptron itself, so we will have to take them into account when working with more complex *Perceptrons* and want to perfect their operation. I recommend you perform each exercise about 20 times at least, so that you have an opportunity to present the variants you can produce. Notice what happens when:

1. **Learning at 7/8 of the Threshold**. It is already a value close to the Threshold, so it may already sometimes not reach the convergence values in the weight, depending on the initial values that these *Weights* acquire.
   - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
   - Threshold **(2)**.
   - Learning **(1.75)**.

2. **Learning at 3/4 of the Threshold**. It is still a value close to the Threshold, and we will continue to have frequent runs in which we do not obtain the Weights of convergence.
   - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
   - Threshold **(2)**.
   - Learning **(1.5)**.

3. **Learning at 1/2 of the Threshold**. This is already a value, which for a case as simple as this Book, is optimal, however for more complex *Perceptrones* can cause changes in the values of the *Weights* too large that cause oscillations that delay the *Weights* reach the *Values of Convergence*, or even never reach them.
   - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
   - Threshold **(2)**.
   - Learning **(1.0)**.

4. **Learning at 3/8 of the Threshold**. For our example, we will notice how we begin to increase the number of attempts necessary for the *Weights* to acquire the *Convergence Values*.
   - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
   - Threshold **(2)**.
   - Learning **(0.75)**.

5. **Learning at 1/4 of the Threshold**. The tendency to require a greater number of attempts continues to increase, and remember is a trend, because even with a much lower *Learning Value*, pseudorandom values of initial *Weights*, there will always be the possibility that they acquire the *Values of Convergence*, with all *Cases* being resolved and no adjustment required.
   - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**

- Threshold (**2**).
- Learning (**0.5**).

Table of Contents

11.4 Optimizing the Range of the Initial Weights

In this case with the *Weights*, we will not have the problem so critical, since although the configuration of the *Weights* is not adequate the *Perceptron* will be able to reach the solution; what yes, is that in *Perceptrones* more complex they make that takes later more attempts to reach the solution; just imagine you require *Very Small Learnings* and that you use *Very Large Initial Weights*, multiplying the work by a thousands times, and that the combinations are complex, a *Perceptron* that could require a few hours, with *Weights* too high, could have us waiting days for the solution. Taking the reference of the *Range* the positive maximum value to compare it with the *Threshold*. Now let's look at what happens when:

1. **Range of Weights 5 times the Threshold**. The tendency for the example of this Book is to require about 5 times more attempts to reach at the solution.
    - Range of Weights **(-10 a 10)**. **random.uniform(-10.0, 10.0)**
    - Threshold **(2)**.
    - Learning **(1)**.

2. **Range of Weights 2 times the Threshold**. We get a double try trend.
    - Range of Weights **(-4 a 4)**. **random.uniform(-4.0, 4.0)**
    - Threshold **(2)**.
    - Learning **(1)**.

3. **Range of Weights 1 time Threshold**. Although the trend is similar to that obtained with 1/2 times the *Threshold*, as the *Perceptron* becomes more complex and the value it requires in *Learning* is lower, the requirement for solution attempts will become greater with respect to ratio of 1/2.
    - Range of Weights **(-2 a 2)**. **random.uniform(-2.0, 2.0)**
    - Threshold **(2)**.
    - Learning **(1)**.

4. **Range of Weights 1/2 times the Threshold**. This is the ratio that I recommend you take from initial reference to adjust the configuration of your projects in the range of *Initial Weights*.
    - Range of Weights **(-1 a 1)**. **random.uniform(-1.0, 1.0)**
    - Threshold **(2)**.
    - Learning **(1)**.

5. **Range of Weights 1/4 times the Threshold and smaller proportions**. For this particular example and the like, there will be no significant difference, because with the value we have in *Learning*, it will cause such changes in the *Weights*, which in a few attempts will reach the solution. However, in more complex *Perceptrons*, in my experience, I have found that they tend to require more attempts

if we continue to reduce the relationship between the *Initial Weight Range* and the *Threshold*.

- Range of Weights **(-0.5 a 0.5)**. **random.uniform(-0.5, 0.5)**
- Threshold **(2)**.
- Learning **(1)**.

Table of Contents

12 Moments of the Perceptron:

Depending on the criteria, the following may vary slightly in divisions and names, but the important thing is that you have an idea of what stage you are working in, where you have advanced and what you have to do to finish your work, and start working the Perceptron for you. In fact as we advance in the Series we can further detail, according to the knowledge incorporated.

Table of Contents

12.1 Information Coding

Once we decided to incorporate a *Perceptron* into our project. We must identify what information will be handled, what will be the *Inputs* and what the *Outputs*, I remind you that it is advisable to make a *Table* that will facilitate us to review the information, verify there are no ambiguities, that is, no more than one Output for a combination of *Inputs*. In our particularly simple example, capacity is limited, and no matter how many entries we want to handle, there can only be one binary *Output*, or (1) or (0).

Table of Contents

12.2 Initial

It is simply the situation in which it is found during the first iteration, in this attempt, the Perceptron has the values that have been assigned to it pseudorandomly, and although it is expected that they are not correct, it may happen that these values solve each and every one of *Cases*, and no adjustment is required.

An alternative to the pseudorandom weights is that having worked with a *Perceptron* with similar task, we assign the *Weights* of its *Learning*, that although it may not solve all the *Cases*, in more complex *Perceptron*, can drastically reduce the number of iterations necessary to reach the values of convergence.

Table of Contents

12.3 Training in Process

It is when the *Perceptron* has begun to adjust the values of the *Weights*, and attempt after attempt approaches to the expected solution. Remember that it may be that the solution is never reached, if the *Perceptron* is not sufficiently robust, or the configuration is not adequate.

When instead of an example as simple as this Book, we have a long list of *Weights*, and it is required to find the solution of hours and hours of computation time; we may have to interrupt the process, save the >Weights, and continue at another time, with the *Weights* saved. The necessary code for this we will incorporate it when it is required, in examples in other Books of this same Series.

Table of Contents

12.4 Finished Training

It is when there are no more *Errors*, and simply the *Weights* that the *Perceptron* has found, solve each and every one of the *Cases* defined, only that unlike the next stage, our code still loads lines that are no longer necessary, but training is over.

Table of Contents

12.5 Working without Supervision

At this moment it is no longer necessary to verify if the calculated is correct, we already know that it is; it is no longer necessary to compare against what we wanted to learn, already learned; no adjustments are required, no changes are shown on the screen, and above all, no further attempts are required to look for new *Weights*, just calculate it with the values learned and we have what we need; with which we can distinguish above all an important reduction in the time of calculation of the *Perceptron* because no longer the following is required:

1. Check whether *Perceptron* calculations are correct or not.
2. Compare the *Output* against expected.
3. Store or retrieve the *Inputs* and *Outputs* of the *Cases*.
4. Calculate the *Error*.
5. Make adjustments to *Weights*
6. Perform iterations and more iterations to find the combination of *Weights* to solve all the *Cases*

What you will have to do is collect the *Inputs* from the environment that you define, and feed it back with the *Output* obtained. The above allows that according to the complexity of *Perceptron* that we are working, it reduces the time of computation consumed, from less than 25%, to thousands of times the one occupied in the training.

Table of Contents

13 Applying the Trained Perceptron:

In order to facilitate incorporation into our *Project*, it is convenient to incorporate our unsupervised code into a function, which will take only the values of the *Weights* learned and the *Inputs* with which we will feed the function, and return the calculated *Output*.

There will be applications in which, by their nature, we can not do without supervision, as it is when we do not have a perfectly defined environment for the *Perceptron*, and it is not known, mainly by its magnitude, all *Cases* and the answer that should be given to them; which may be because we are interacting with a complex environment, as a Robot does, because in reality it does not learn from a *Table of Values*, but from the environment itself.

Now let's build the function. Of course, this is not the only way to do it, but it is a starting point, and I put it so that you realize, that not only reduces the computation time, it is also reduced code, without supervision. The code and its pseudocode, after omitting the supervision, and transforming it into a function in which you substitute (Value_Learned_1 and Value_Learned_2) for the *Weights* learned, can be as in the following Point is shown:

Table of Contents

13.1 Pseudocode of Function

```
- # Define Function: Simple Perceptron
- Function Begins Perceptrón
-     receive variable type int called: Inp_1
-     receive variable type int called: Inp_2

-   # Memory Weights
-   declares variable type float called: Wgh_1
-   declares variable type float called: Wgh_2
-   puts Value_Learned_1 in Wgh_1
-   puts Value_Learned_2 in Wgh_2

-   # Constants of the Rigid Parameters
-   declares variable type float called: Thr
-   puts 2.0 in Thr

-   # Work Variables
-   declares variable type float called: Sum

-   # Perceptron Calculation
-   Multiply Inp_1 by Wgh_1
-   and Multiply Inp_2 by Wgh_2
-   sum results and puts it in Sum

-   # Comparison against Threshold
-   If Sum Exceeds Thr:
-     return 1 # Threshold Exceeded
-   else case:
-     return 0 # Threshold Not Exceeded

- # Function Ends: Simple Perceptron
```

Table of Contents

13.2 Function Code in Python

```python
# Define Function: Simple Perceptron
def Perceptron(Inp_1, Inp_2):
    # Memory Weights
    Wgh_1 = Value_Learned_1
    Wgh_2 = Value_Learned_2
    # Constants of Rigid Parameters
    Thr = 2.0
    # Perceptron Calculation
    Sum = Inp_1 * Wgh_1 + Inp_2 * Wgh_2
    if Sum > Thr: # Comparison against Threshold
      return 1 # Threshold Exceeded
    else:
      return 0 # Threshold Not Exceeded
```

Another exercise you can perform is to modify your code to solve for the same *Tables of Truth*, but now with a greater number of *Inputs*. Remember that modifying your code allows you to acquire more domain of this and make yours this knowledge, facilitating to acquire new knowledge that is related.

Remember that this is only the first example of the Series, extremely simple to facilitate its analysis, both the elements that compose it and its configuration, but as the Series progresses, new elements will be incorporated, which we will analyze and test, which will increase the ability of algorithms to learn to solve more complex problems.

Table of Contents

Feedback:

Your comments, doubts or suggestions will be well received. I leave you my email:

neural.networks.step.by.step@gmail.com

Table of Contents