|  |
| --- |
| A |
| Perceptron Classifier |
| to Simulate Boolean AND Logic |

|  |
| --- |
| Prinkesh Sharma  11-26-2014 |

**Perceptron Classifier (**A Brief Introduction**):**

A perceptron classifier is a simple model of a neuron and is one of the simplest linear classifier of a neural network .

It has different inputs (x1...xn) with different weights (w1...wn) .

The weighted sum ***s*** of these inputs is then passed through a step function ***f***

The perceptron classifier can be represented as

Adjusting the weights for the classifier depending upon a particular training data set allows us to simulate Boolean AND Logic . Next we present such a to do so and look at the various factors in the Training Phase of the perceptron.

**Perceptron Classifier (**Simulation of Boolean AND Logic**):**

Here we, present such a perceptron using python and some math libraries for easier and faster computation of dot products .

The **step function** or **activation function** :

1. unit\_step = **lambda** x: 0 **if** x < 0 **else** 1

**Training Data** (for 2 inputs) :

1. training\_data = [ (array([0,0,1]),0),
2. (array([0,1,1]),0),
3. (array([1,0,1]),0),
4. (array([1,1,1]),1)
5. ]

Mapping of all possible input to the expected output is provided as training data .

Each tuple in the training data is an input to expected output mapping , where each input is a numpy array of 0 or 1 and output is a single number 0 or 1.

For this case (2 input AND logic ) ,  
The first two elements in the input array are the inputs provided .  
The third entry in the array is “***dummy***” input (also called the bias) which is needed to move the threshold (also known as the decision boundary) up or down as needed by the step function. Its value is always 1, so that its influence on the result can be controlled by its weight .

This training sequence for two inputs maps exactly to the Boolean logic definition of the AND function

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **A AND B** |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Next we choose ***initial*** ***weights*** for the 2 inputs and dummy input in a range of ***0*** to ***1***

1. w = random.rand(3)

***w*** is a numpy array of ***3*** random numbers between ***0*** and ***1***

1. >>> w=random.rand(3)
2. >>> w
3. array([ 0.34630468,  0.41118564,  0.21630602])

**Other Initializations :**

1. errors = []
2. eta = 0.2
3. n = 150

***errors*** list is only used to store the error values which is later used to visualize the stabilization of weights given to each input and to visualize the learning process of the perceptron

***eta*** variable controls the learning rate of the perceptron

***n*** specifies the number of learning iterations for the perceptron

In order to find the ideal values for the weights ***w*** , we try to reduce the error magnitude to zero.

Here we take n=150 , as for this simple case these number of iterations are sufficient to learn and adjust the weights ; for a bigger and probably “noisier” set of input data much larger number should be used .

**Training Phase :**

1. **for** i **in** xrange(n):
2. x, expected = choice(training\_data)
3. result = dot(w, x)
4. error = expected - unit\_step(result)
5. errors.append(error)
6. w += eta \* error \* x

For every Iteration,  
First we choose a random input set from the training data.

***X*** is the input numpy array and variable ***expected*** is the expected output for the given input

Then we calculate the dot product of the input and the weight vector , which is then compared to the expected value . If the expected value is bigger, we need to increase the weights, if it's smaller, we need to decrease them.

***Error***  is hence expected result negation the result achieved by dot product , we store this error for each iteration in the ***errors*** list.

The correction factor to adjust the weight is found out by multiplying ***error*** , learning rate ( ***eta*** ) and the input vector (***x***) . It its then added to respective weights in order to improve result in the next iteration .  
  
By , going through the training phase mentioned above with the give training data , the perceptron has now “learned” to simulate the behaviour of a logical AND function

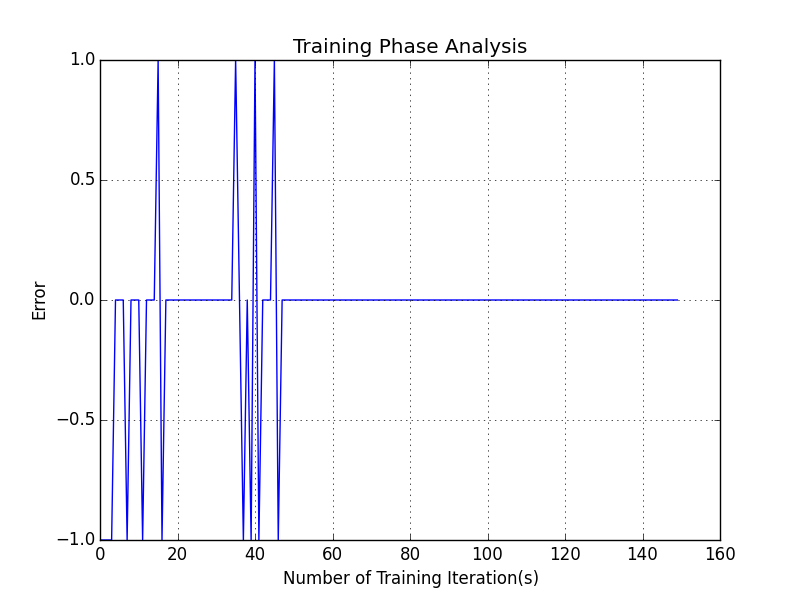
**Results:**

Now , the perceptron is tested for the accuracy of the results

1. **for** x, \_ **in** training\_data:
2. result = dot(x, w)
3. **print**("{}: {} -> {}".format(x[:2], result, unit\_step(result)))
4. >>>
5. >>>
6. [0 0]: -0.238167808308 -> 0
7. [0 1]: -0.0846635000308 -> 0
8. [1 0]: -0.0919312887159 -> 0
9. [1 1]: 0.0615730195615 -> 1
10. >>>

The result cleary shows that , for the given combinations of input the expected results and the obtained results are same . Hence , the perceptron simulated Boolean AND Logic with two inputs and can be easily trained for more than two inputs using appropriate training data .

**Training Phase Analysis :**

Error in each iteration that was stored in the list ***errors*** is now used to plot the learning behaviour of the perceptron .

From plotting the error graph we can see that the perceptron stabilises by 50 iterations and now is capable of simulating Boolean **AND** Logic