Perceptron Algorithm

Ab Waheed Lone
Department of Computer Engineering
Yildiz Technical University
Student No: 18501053

17th October 2019

1 Introduction

A perceptron is a neural network unit (an artificial neuron) that does certain computations to detect or classify the given input data. It is an algorithm for supervised learning of binary classifiers which enables neurons to learn and processes elements in the training set one at a time.

Perceptrons had perhaps the most far-reaching impact of any of the early neural nets and perceptron learning rule is more powerful than the Hebb rule. The Perceptron model is a more general computational model.It takes an input,aggregates it(weighted sum) and returns 1 only if the aggregated sum is more than some threshold else returns 0. There are two types of perceptrons:Single Layer and Multilayer perceptrons.A single perceptron can only be used to implement linearly separable functions.It takes both real and boolean inputs and associates a set of weights to them, along with a bias. algorithm

1.1 Perceptron Algorithm

The algorithm given below is suitable for either binary or bipolar input vectors(n-tuples),with a bipolar target,fixed threshold Θ, and adjustable bias.

Step 0. Initialize weights and bias. (For simplicity,set weight and bias to zero.)
Set learning rate $\alpha(0¡\alpha \leq 1)$ (For simplicity value of alpha can be set to 1.)
Step 1.

While stopping condition is false,do steps 2-6.
Step 2. For each training pair s:t,do steps 3-5.

Step 3. Set activations of input units: $\mathrm{x}_i = s_i$.
Step 4. $Computeresponseofoutputunit.y_{in} = b + \sum_{i=1} \mathrm{x}_i * w_i$

Step 5. Update weights and bias if an error occurred for this pattern.
If y$\neq t$
$w_i(new) = w_i(old) + \alpha * t * x_i$
$b(new) = b(old) + \alpha * t.$

(where t is the mean square difference between expected and observed value(also called the error function))
else
$w_i(new) = w_i(old)$
$b(new) = b(old)$

Step 6. Test stopping condition
if no weights changed in step 2,stop; else,continue

Note that only weights connecting active input units ($x_i \neq 0$) are updated.Also,weights are updated only for patterns that do not produce the correct values of y. This means that as more training patterns produce the correct response,less learning occurs.

Instead of one separating line(unlike ADALINE units),we have a line separating the region of positive response from the region of zero response,namely, the line bounding the inequality

$$w_1x_1 + w_2x_2 + b > \theta$$

and a line separating the region of zero response from the region of negative response,namely, the line bounding the inequality

$$w_1x_1 + w_2x_2 + b < -\theta$$

1.1.1 Applications

Two basic applications of Perceptron model

- Logical OR Gate (Linearly separable)
- Logical XOR Gate (Non-linearly separable)

OR Logical function		
X_1(First input feature value)	X_2(Second input feature value)	Y(Output activation)
0	0	0
0	1	1
1	0	1
1	1	1

Since OR logical function is linearly separable, model can be implemented using single Neuron with logical 0 and 1 as its input values. Single Neuron OR model consists of 2 different phases.

1) Learning phase 2) Testing Phase
Implementation of Learning phase involves **Forward propagation** function which includes weight vector initialization and then summation of dot products between input vectors (0's and 1's) and randomly chosen weight vectors.

$$z = x_1 * w_1 + x_2 * w_2 + b$$

where b is some bias.

Learning Phase's second step involves using some activation function depending on the problem under consideration and the range of the expected output values.There are different activation functions used for getting the activated output from input values like Step function, Sigmoid, tanh, ReLU, Non-ReLu etc.
Here we used **tanh** activation function with its range equal to [-1,1].
Activated output of a neuron

$$y = \mathbf{tanh}(z)$$

.
With activated output in hand,we calculate the value of error between the expected output and observed value using Mean Square Error function.In case of OR function Expected output values are [0,1,1,1] and the observed values are activation values.

To minimise the error (equal to zero) or obtain better predicted values and check the effect of different variables an algorithm called **Back-propagation** is used. Back-propagation checks how changing the weights and biases in a network changes the cost function (error function).Ultimately this means computing the partial derivatives of error function with respect to weights and biases. Along every direction of the computational graph of network,partial derivatives are calculated which involves calculating local gradient and upstream gradient (Gradient which comes from nodes which are after the current node in a computational graph). Using the Gradient descent algorithm weights are updated continuously.

In case of logical OR function,during Back phase (or gradient calculation) partial derivative of **tanh** function is used like we used **tanh** in forward propagation phase. Weight adjustments are calculated by doing the dot between training inputs and a term which involves multiplication between error function and derivative of activation function. Then using the gradient descent algorithm, weight updates are made.

In the test phase, the non-learned inputs are provided to the learned Neuron to check the output through the model to get the expected output value.

1.1.2 Python code for single Neuron OR Function

```
import numpy as np
from numpy import random,array,dot,exp,tanh
class NeuralNetwork():
    def __init__(self):
        random.seed(1)
        self.weight_matrix=2*np.random.random((2,1))-1

    def tanh(self,x):
        return tanh(x)
    def tanh_derivative(self,x):
        return 1.0-tanh(x)**2

    #forward propagation
    def forward_prop(self,inputs):
        return self.tanh(np.dot(inputs,self.weight_matrix))
    def train(self,train_inputs,train_outputs,number_iterations):
        for iteration in range(number_iterations):


           output=self.forward_prop(train_inputs)
           #print(output)


          #error
           error=(train_outputs-output)**2
           print(error)
        #adjustment
           adjustment = np.dot(train_inputs.T, error * self.
    tanh_derivative(output))
           # adjust
           self.weight_matrix += adjustment



if __name__=="__main__":
    neural_network=NeuralNetwork()
    print("random weights before training")
    print(neural_network.weight_matrix)
    train_inputs=np.array([[0,0],[0,1],[1,0],[1,1]])
    train_outputs=np.array([[0,1,1,1]]).T
    neural_network.train(train_inputs,train_outputs,10)
    print('new weights')
    print(neural_network.weight_matrix)
    print('testing ')
    print(neural_network.forward_prop(array([[1,
    0],[0,0],[0,1],[1,1]])))
```

```
import numpy as np
from numpy import random,array,dot,exp,tanh
class NeuralNetwork():
    def __init__(self):
        random.seed(1)
        self.weight_matrix=2*np.random.random((2,1))-1

```

```
    def tanh(self,x):
        return tanh(x)
    def tanh_derivative(self,x):
        return 1.0-tanh(x)**2

    #forward propagation
    def forward_prop(self,inputs):
        return self.tanh(np.dot(inputs,self.weight_matrix))
    def train(self,train_inputs,train_outputs,number_iterations):
        for iteration in range(number_iterations):


           output=self.forward_prop(train_inputs)
           #print(output)


          #error
           error=(train_outputs-output)**2
           print(error)
        #adjustment
           adjustment = np.dot(train_inputs.T, error * self.
    tanh_derivative(output))
           # adjust
           self.weight_matrix += adjustment



if __name__=="__main__":
    neural_network=NeuralNetwork()
    print("random weights before training")
    print(neural_network.weight_matrix)
    train_inputs=np.array([[0,0],[0,1],[1,0],[1,1]])
    train_outputs=np.array([[0,1,1,1]]).T
    neural_network.train(train_inputs,train_outputs,10)
    print('new wieghts')
    print(neural_network.weight_matrix)
    print('testing ')
    print(neural_network.forward_prop(array([[1,
    0],[0,0],[0,1],[1,1]])))
```

Neuron3.JPG

```
random weights before training
[[-0.16595599]
 [ 0.44064899]]
[[0.        ]
 [0.34318245]
 [1.35594156]
 [0.53584413]]
[[0.00000000e+00]
 [2.47608570e-02]
 [5.00108236e-03]
 [3.89494020e-05]]
[[0.00000000e+00]
 [2.35940979e-02]
 [4.95592364e-03]
 [3.66244022e-05]]
[[0.00000000e+00]
 [2.25368386e-02]
 [4.91160414e-03]
 [3.45348230e-05]]
```

Figure 1: Testing results

```
[[0.00000000e+00]
 [1.84492702e-02]
 [4.70176235e-03]
 [2.66272183e-05]]
[[0.00000000e+00]
 [1.78093922e-02]
 [4.66199321e-03]
 [2.54168171e-05]]
new wieghts
[[1.67346263]
 [1.32815586]]
testing
[[0.93200801]
 [0.        ]
 [0.86879789]
 [0.9950707 ]]
```

Neuron4.JPG

Figure 2: Testing results