Report
on

Build Perceptron From Scratch
08-Jan-2019

Submitted by

Love Prakash

DRONAMAPS

DronaMaps Private Limited

Contents

1 Introduction:

Machine learning is a field within Ai that focuses on the design of algorithms that can learn from a given data and results which we call “training data” to make a prediction based on that given data from new input data. Deep learning is a sub-field within[?] Machine learning which focuses on the same goal but which uses neural networks & deep neural networks. So what’s the perceptron then ? Well the perceptron is an algorithm for supervised learning (which means we know the result we’re trying to get, say like feeding the size & location of a house to predict the price, in the other hand there is unsupervised learning which is used to draw results from datasets consisting of input data without labeled responses).

The perceptron receives input data multiplied by random weights and adds a bias value, put in[?] an activation function to get a result, if the result value is wrong, it uses back propagation & gradient descent to go back & tweak the weights to get a correct result.

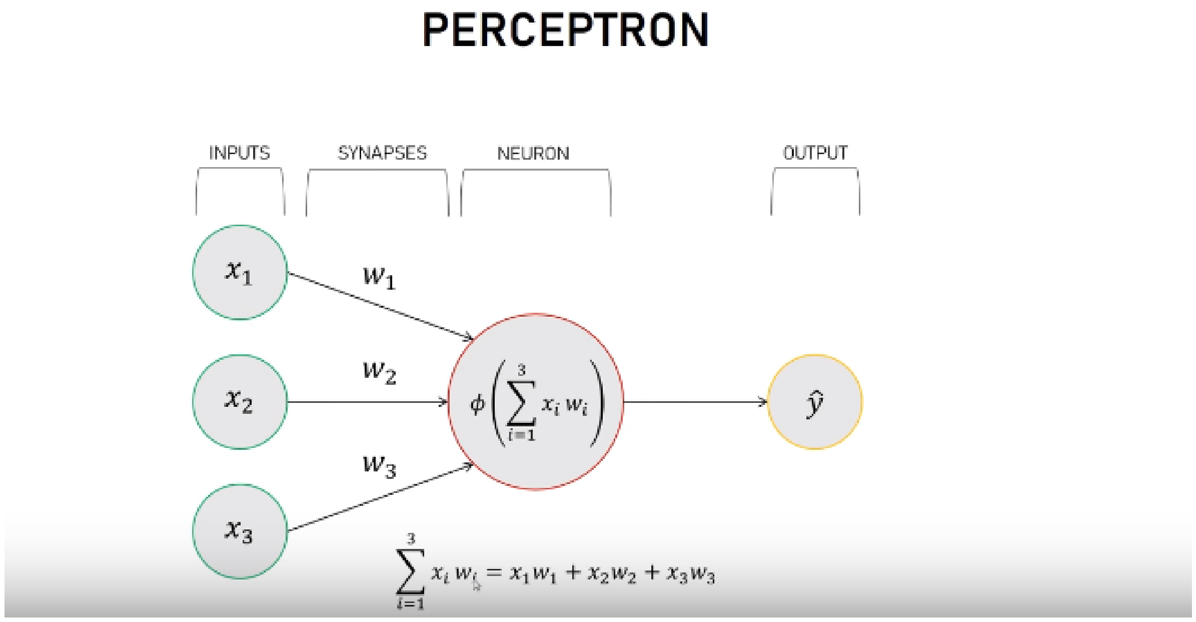
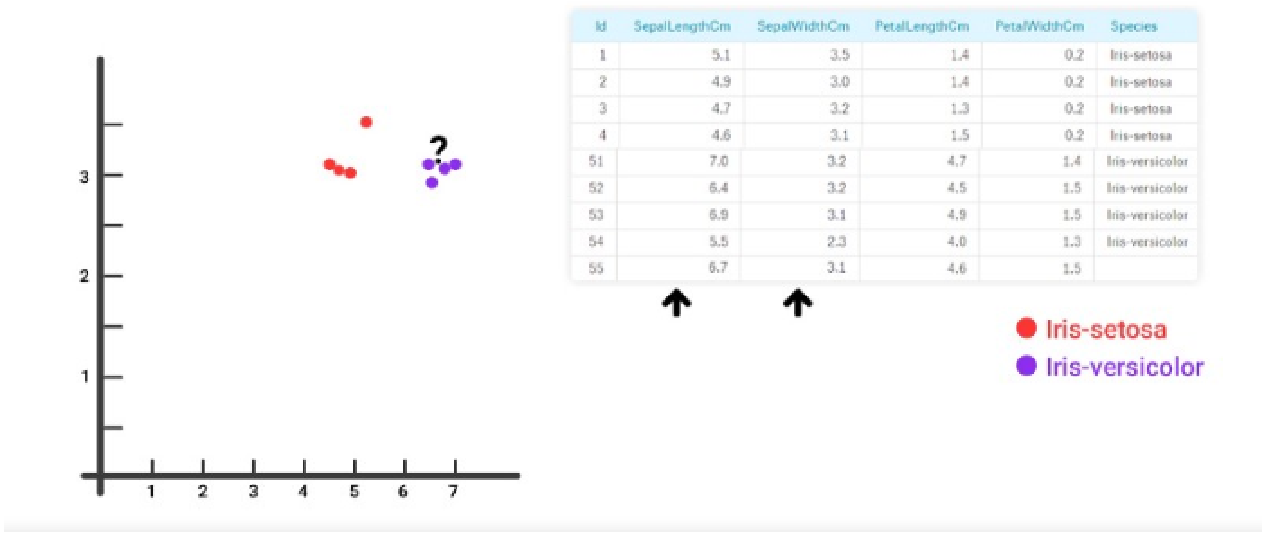

PERCEPTRON
INPUTS
SYNAPSES
NEURON
OUTPUT
x_1
x_2
x_3
w_1
w_2
w_3
$\phi\left(\sum_{i=1}^{3} x_i w_i\right)$
$\hat{y}$
$\sum_{i=1}^{3} x_i w_i = x_1 w_1 + x_2 w_2 + x_3 w_3$

Figure 1: P
erceptron model

2 Why Perceptron?

I know there are some terms there you didn't get, so let's go ahead and explain that slowly, let's say you're a farmer and you want to classify two types of flowers manually, the best way to do this manually is to take the length width of each flowers' paddle and represent them on a graph just like shown bellow:

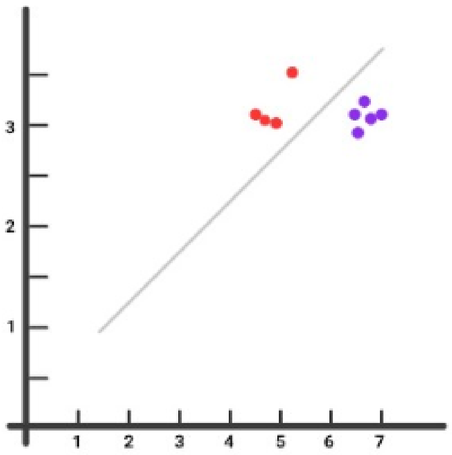


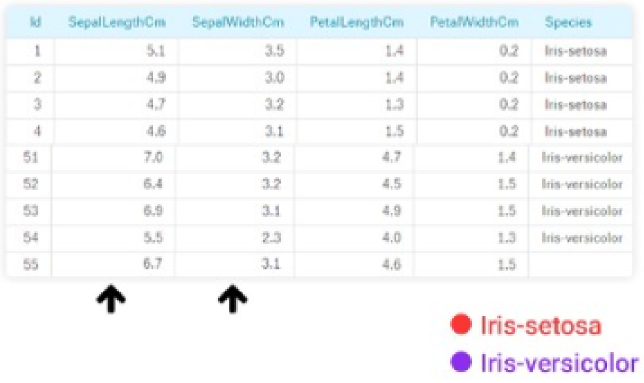
Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3.0	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	Iris-setosa
51	7.0	3.2	4.7	1.4	Iris-versicolor
52	6.4	3.2	4.5	1.5	Iris-versicolor
53	6.9	3.1	4.9	1.5	Iris-versicolor
54	5.5	2.3	4.0	1.3	Iris-versicolor
55	6.7	3.1	4.6	1.5	

Figure 2: l
inear classification

Noticed the unknown flower we're trying to figure its type, as you can see on the graph it's represented among the type 2 of flowers which means it is a type 2 flower.

What you've just did manually is called linear regression, you've represented as many flowers you have on the graph as a training data, then you noticed that pattern and drew a line between the two types.





Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3.0	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	Iris-setosa
51	7.0	3.2	4.7	1.4	Iris-versicolor
52	6.4	3.2	4.5	1.5	Iris-versicolor
53	6.9	3.1	4.9	1.5	Iris-versicolor
54	5.5	2.3	4.0	1.3	Iris-versicolor
55	6.7	3.1	4.6	1.5	

● Iris-setosa
● Iris-versicolor

Figure 3: L
inear classification line

3 Sigmoid Normalizing Function:

So now let's automate it using our own perceptron built from scratch, the activation function we'll be using is Sigmoid, here's how it looks like on the graph:

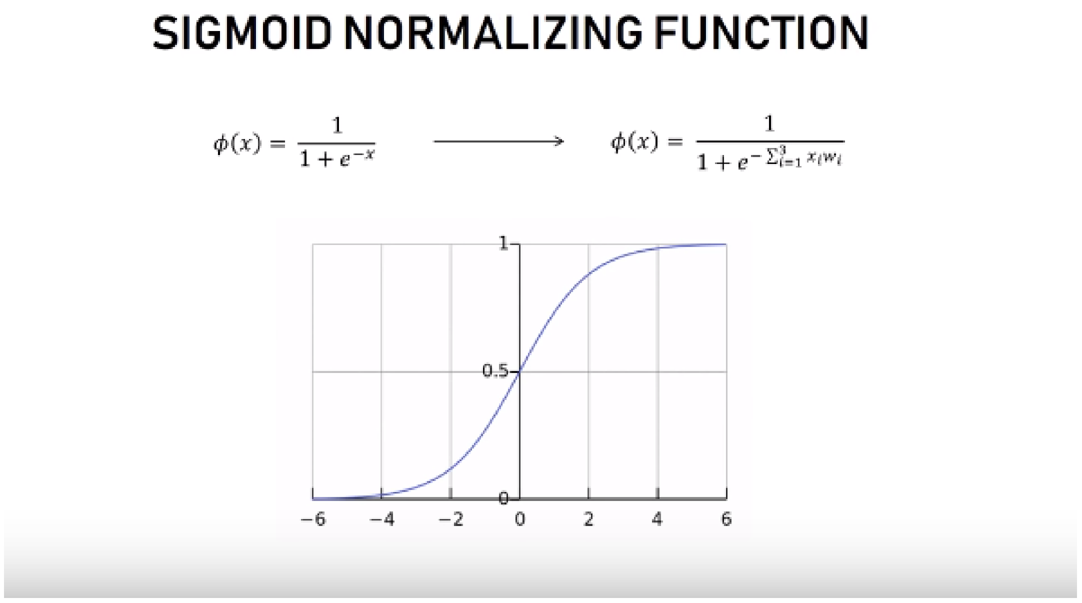


Figure 4: S
igmoid normalizing function

4 Step by step Procedure:

- Take inputs from the training example and put them through our formula to get the neuron's output.
- Calculate the error which is the difference between the output we got and the actual output.
- Depending upon the severeness of the error, adjusts the weight accordingly.
- Repeat this 100,000 times.

5 Tool used:

For the simple perceptron model we have used Spider for the coding in this report.

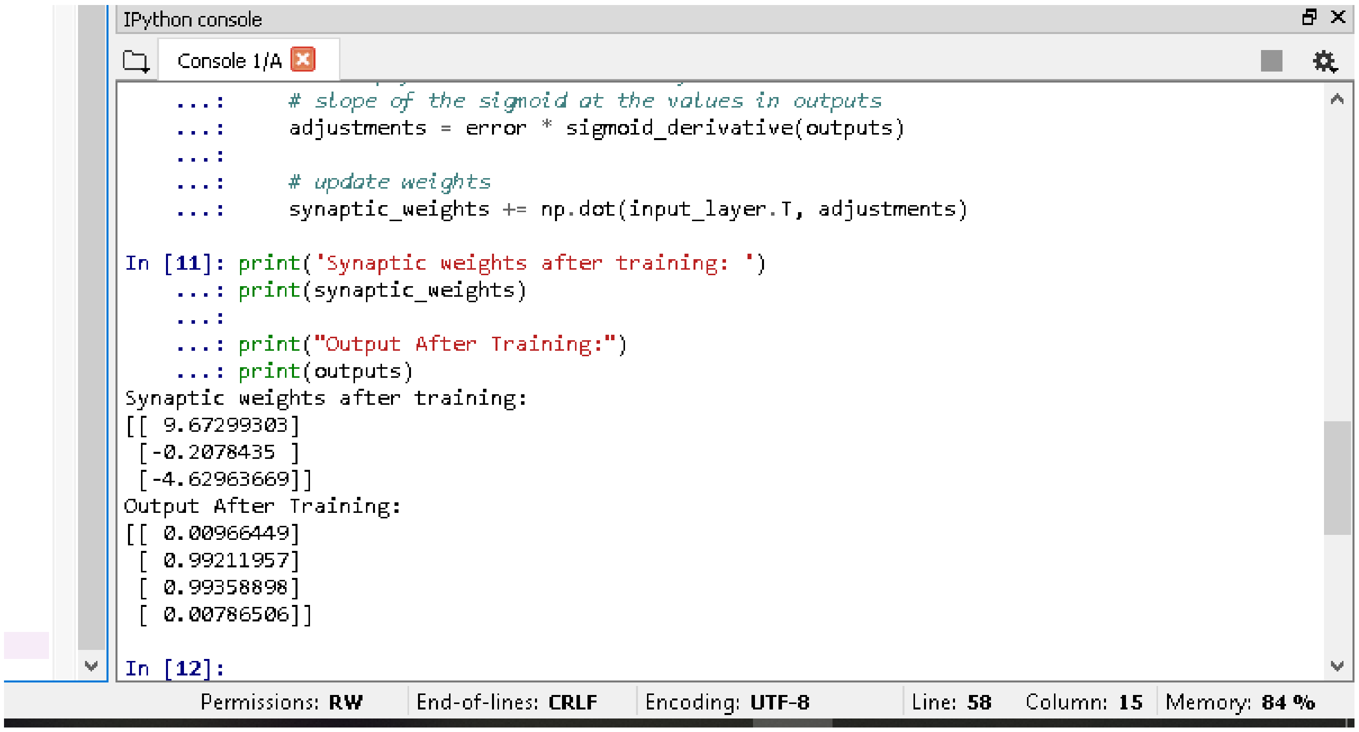
6 Python Code:

```
import numpy as np

# sigmoid function to normalize inputs
def sigmoid(x):
return 1 / (1 + np.exp(-x))
```

```
# sigmoid derivatives to adjust synaptic weights
def sigmoid_derivative(x):
return x * (1 - x)

# input dataset
training_inputs = np.array([[0,0,1],
[1,1,1],
[1,0,1],
[0,1,1]])

# output dataset
training_outputs = np.array([[0,1,1,0]]).T

# seed random numbers to make calculation
np.random.seed(1)

# initialize weights randomly with mean 0 to create weight matrix, synaptic
synaptic_weights = 2 * np.random.random((3,1)) - 1

print('Random starting synaptic weights: ')
print(synaptic_weights)

# Iterate 10,000 times
for iteration in range(10000):

# Define input layer
input_layer = training_inputs
# Normalize the product of the input layer with the synaptic weights
outputs = sigmoid(np.dot(input_layer, synaptic_weights))

# how much did we miss?
error = training_outputs - outputs

# multiply how much we missed by the
# slope of the sigmoid at the values in outputs
adjustments = error * sigmoid_derivative(outputs)

# update weights
synaptic_weights += np.dot(input_layer.T, adjustments)

print('Synaptic weights after training: ')
print(synaptic_weights)

print("Output After Training:")
print(outputs)
```

7 Output of the perceptron:

Below shown is the output of the percepton model.



IPython console

Console 1/A

```
    ...:     # slope of the sigmoid at the values in outputs
    ...:     adjustments = error * sigmoid_derivative(outputs)
    ...:
    ...:     # update weights
    ...:     synaptic_weights += np.dot(input_layer.T, adjustments)

In [11]: print('Synaptic weights after training: ')
    ...: print(synaptic_weights)
    ...:
    ...: print("Output After Training:")
    ...: print(outputs)
Synaptic weights after training:
[[ 9.67299303]
 [-0.2078435 ]
 [-4.62963669]]
Output After Training:
[[ 0.00966449]
 [ 0.99211957]
 [ 0.99358898]
 [ 0.00786506]]

In [12]:
```

Permissions: RW   End-of-lines: CRLF   Encoding: UTF-8   Line: 58   Column: 15   Memory: 84 %

Figure 5: F
inal Output