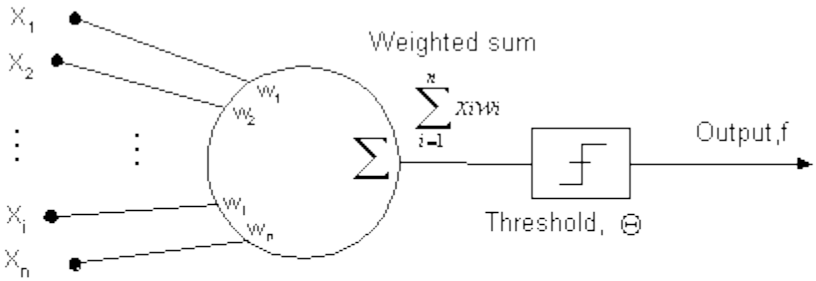
The assignment has been uploaded it on GitHub:

https://github.com/saman-nia

A. Neural Network Decision Boundary:

I will explain the question through a simpler representation called a Threshold Logic Unit.

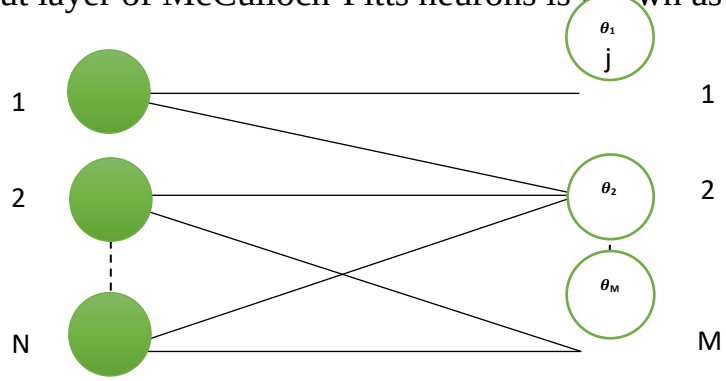


$$f = 1 \text{ if } \sum_{i=1}^{n} x_i w_i \geq \Theta$$
$$= 0, \text{ otherwise}$$

1. A set of connection between neurons gives an activations from other neurons.
2. In the circle of above shape, the unit sums the inputs, and then applies a non-linear activation function.
3. The output line transfer the result to other neurons.

The McCulloch-Pitts model was an extremely simple artificial neuron. The inputs could be either a zero or a one. And the output was a zero or a one. And each input could be either excitatory or inhibitory. Now the whole point was to sum the inputs. If an input is one, and is excitatory in nature, it added one. If it was one, and was inhibitory, it subtracted one from the sum. This is done for all inputs, and a final sum is calculated. An arrangement of one input layer of McCulloch-Pitts neurons feeding forward to one output layer of McCulloch-Pitts neurons is known as a Perceptron.

CV



$$Y_j = sgn\left(\sum_{i=1}^{n} Y_i . w_{ij} - \theta_j\right)$$

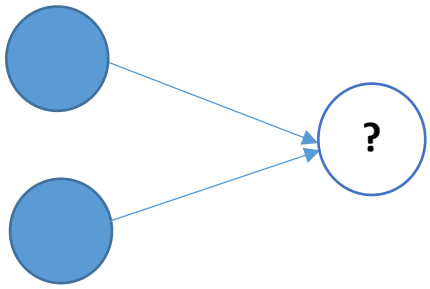
Running the Logical Gates:

NOT:

Input	Output
0	1
1	0

AND:

Input		Output
0	0	0
0	1	0
1	0	0
1	1	1



OR:

Input		Output
0	0	0
0	1	1
1	0	1
1	1	1

The purpose is train the network to compute weights and threshold from decision boundaries between classes.
Next we need to compute Decision surface which is the surface at which the output of the unit that is equal to the threshold:
$\sum_{i=1}^{n} w_i.I_{i=}\theta$

Next we need to plot the decision boundaries of our logic gates.

AND:

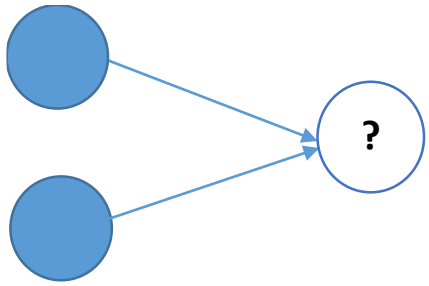
$w1=1, w2=1, \theta=1.5$

OR:

$w1=1, w2=1, \theta=0.$

Single-Layer Feed-Forward neural nets can be represented as weighted directed graphs. One input layer and one output layer of processing units.

Single-Layer Feed-Forward:



Activation and Transfer Function:

Threshold: $f(x)= sgn(\sum_{i=1}^{n} w_i x_i - \theta)$, $f(x)=\begin{cases}1 & if\,(w,x) \geq \theta \\ 0 & if\,\langle w,x\rangle < \theta\end{cases}$

Piecewise-Linear: $f(x)= \begin{cases} -x-3 & if\ x \leq -3 \\ x+3 & if -3 < x < 0 \\ -2x+3 & if\ 0 \leq x \leq 3 \\ 0.5x-4.5 & if\ x \geq 3 \end{cases}$

Sigmoid: $S(t) = \frac{1}{1+e^t}$

To compute the weights, The Perceptron equation can be simplified if we consider that the threshold is another connection weight then The Perceptron equation then becomes:

$Y_j=sgn\ (\sum_{i=1}^{n} I_i.w_{ij})$

B. Classification with a perceptron:

To define Gaussians data sets, I have used from the *numpy* library import *random.multivariate_normal* which draw random samples from a multivariate normal distribution. I have declare a tuple data set which [0] is the narrow and [1] is the wide.

Run the Perceptron:

Here I run my perceptron to classify dataset into two classes.

```
class Perceptron(object):
    def __init__(self, alpha=0.0001, epochs=10):
        self.alpha = alpha
        self.epochs = epochs
    def train(self, X, y):
        self.w = numpy.zeros(1 + X.shape[1])
        self.cost_ = []
        for _ in range(self.epochs):
            cost = 0
            for xi, target in zip(X, y):
                update = self.alpha * (target - self.predict(xi))
                self.w[1:] +=  update * xi
                self.w[0] +=  update
                cost += int(update != 0.0)
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        return numpy.dot(X, self.w[1:]) + self.w[0]

    def predict(self, X):
        return numpy.where(self.net_input(X) >= 0.0, 1, -1)
```

after implementation of the dataset through the perceptron, the weight is:

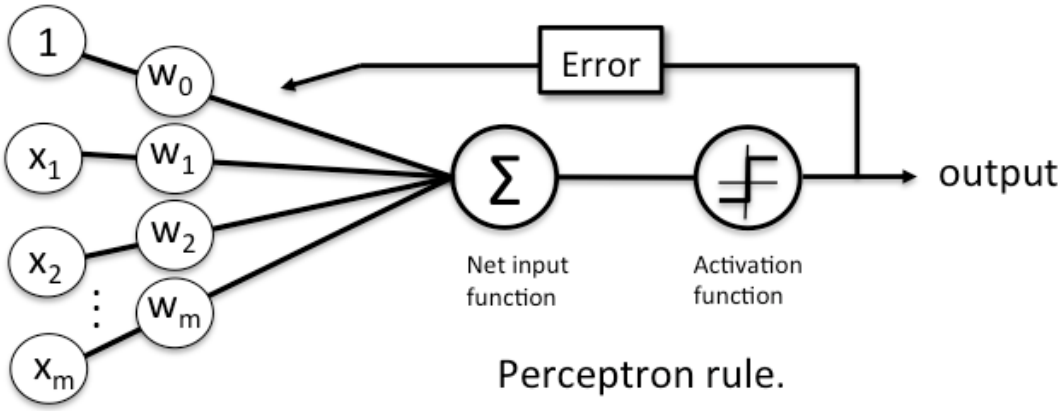
```
print('Weights: %s' % perceptron.w)

Weights: [ 0.34        0.89359412 -0.14830237]
```

After the implementation perceptron, convergence a problem. The perceptron learning can converge if the two classes can be separated by linear hyperplane. To do this issue, I will use two different classes and features from the dataset.

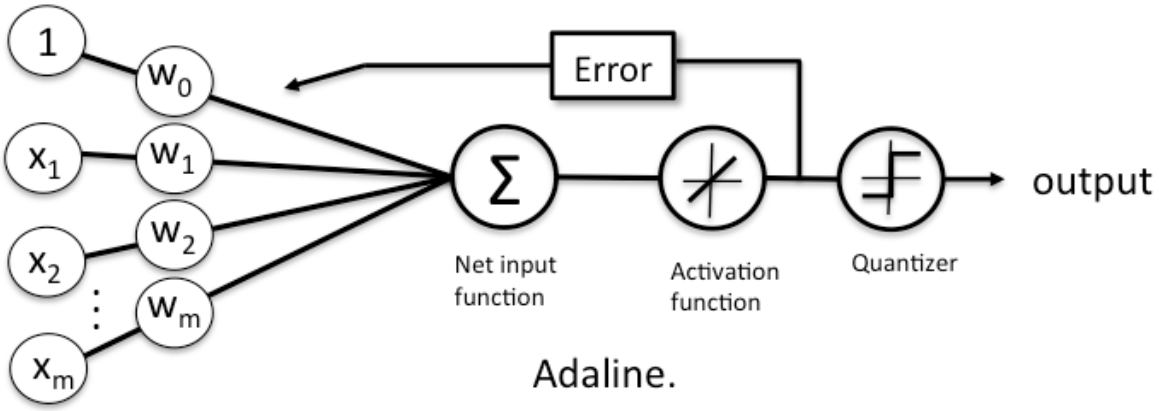
Adaptive Linear Neurons:

In contrast to the perceptron rule, the delta rule of the or Adaline rule updates the weights based on a linear activation function rather than a unit step function; here, this linear activation function g(z)g(z) is just the identity function of the net input g(wTx)=wTxg(wTx)=wTx. In the next section.

I implement the model based on Adaptive Linear Neurons.



Perceptron rule.



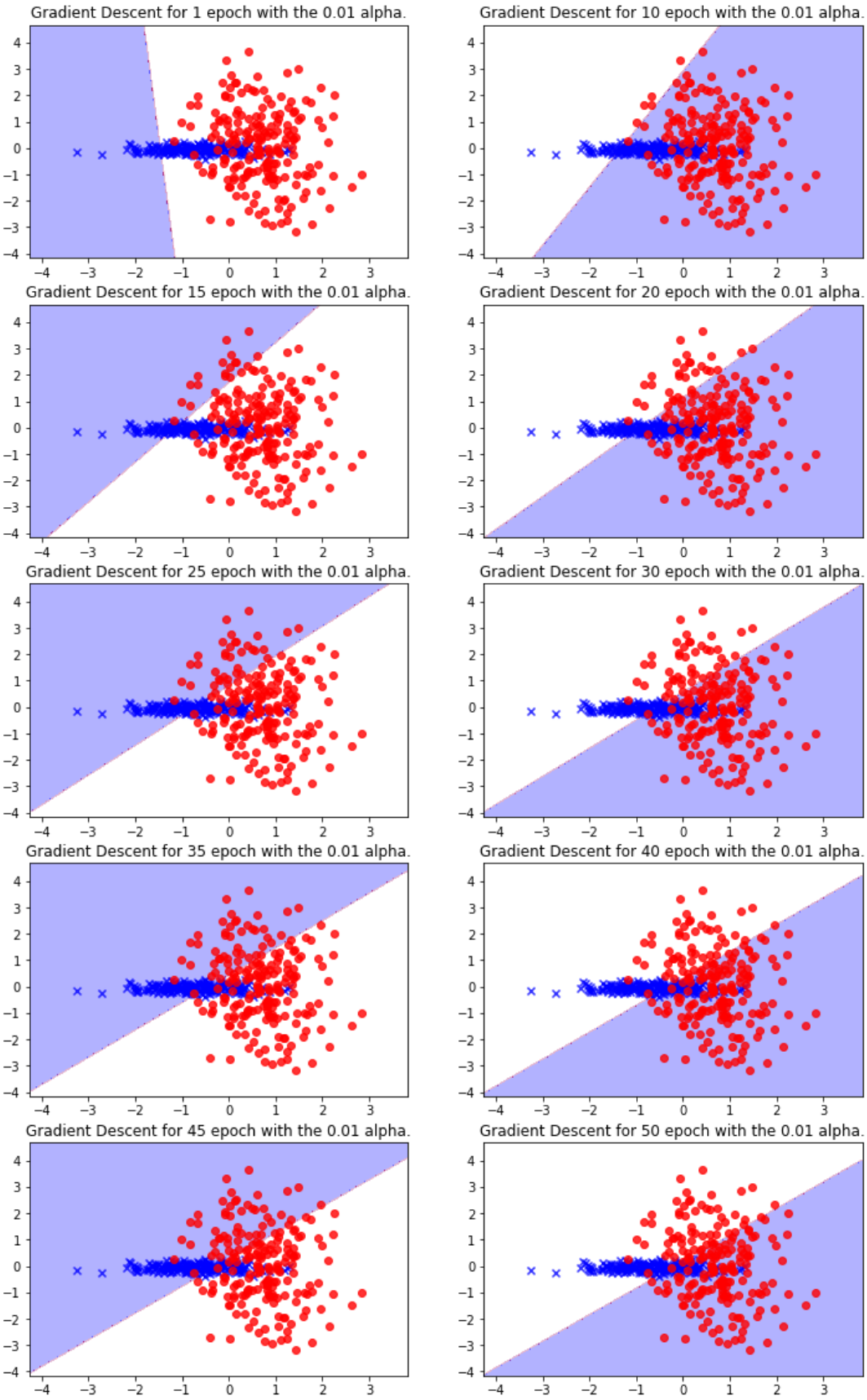
Adaline.

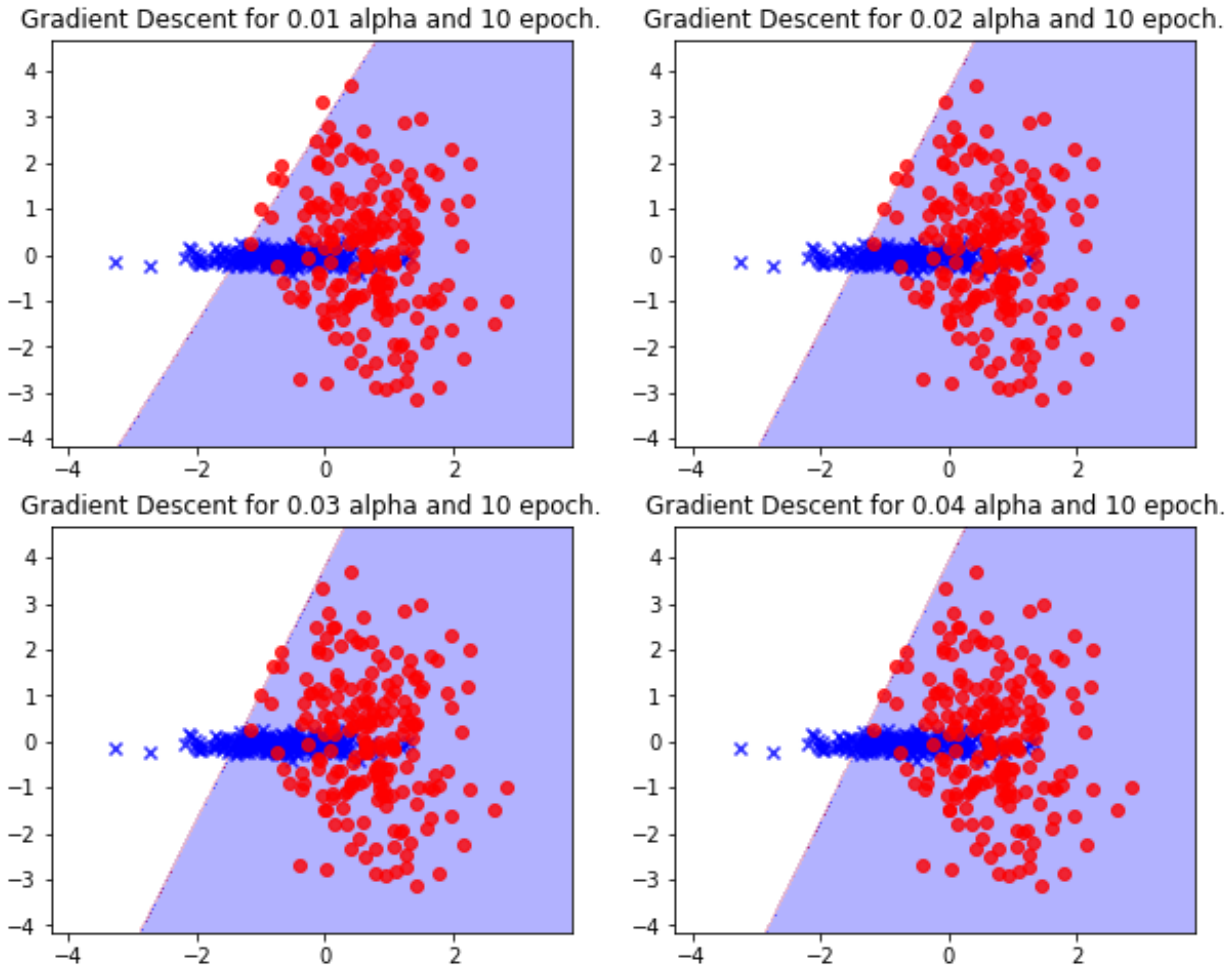
Gradient Descent:

The biggest advantages of the linear activation function over the unit step function is that it is differentiable. This property allows us to define a cost function J(w)J(w) that we can minimize in order to update our weights. In the

case of the linear activation function, we can define the cost function J(w)J(w) as the sum of squared errors (SSE), which is similar to the cost function that is minimized in ordinary least squares (OLS) linear regression.

```
class stochastic_gradient_descent(object):

    def __init__(self, alpha, epochs):
        self.alpha = alpha
        self.epochs = epochs
    def train(self, X, y):
        self.w = numpy.zeros(1 + X.shape[1])
        self.cost_ = []
        for i in range(self.epochs):
            output = self.net_input(X)
            errors = (y - output)
            self.w[1:] += self.alpha * X.T.dot(errors)
            self.w[0] += self.alpha * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self
    def net_input(self, X):
        return numpy.dot(X, self.w[1:]) + self.w[0]
    def activation(self, X):
        return self.net_input(X)
    def predict(self, X):
        return numpy.where(self.activation(X) >= 0.0, 1, -1)
```


Gradient Descent for 1 epoch with the 0.01 alpha.
Gradient Descent for 10 epoch with the 0.01 alpha.
Gradient Descent for 15 epoch with the 0.01 alpha.
Gradient Descent for 20 epoch with the 0.01 alpha.
Gradient Descent for 25 epoch with the 0.01 alpha.
Gradient Descent for 30 epoch with the 0.01 alpha.
Gradient Descent for 35 epoch with the 0.01 alpha.
Gradient Descent for 40 epoch with the 0.01 alpha.
Gradient Descent for 45 epoch with the 0.01 alpha.
Gradient Descent for 50 epoch with the 0.01 alpha.


Gradient Descent for 0.01 alpha and 10 epoch.
Gradient Descent for 0.02 alpha and 10 epoch.
Gradient Descent for 0.03 alpha and 10 epoch.
Gradient Descent for 0.04 alpha and 10 epoch.