Project 4 - By Wilson Peguero Rosario

Problem Statement

In this assignment, the learning rule for perceptron is examined as an example of proper network behavior. The notions of gradient descent and stochastic gradient descent are emphasized, as well as post-training and the implication of fitting a model to a training data set. Overfitting is adderessed and the tradeoff of bias as compared to variance is also discussed.

Consider a signle neuron perceptron with a hard limit transfer function (hardlim).

Solution

A perceptron can be considered to be an artificial neuron. It receives input from other perceptrons or from sensors and has a threshold at which the perceptron decides to send a signal down the chain or not. This can be considered to be the activation and in this case, there is no gradual transition from 0 to 1 as the hard limit function is 0 where x<0 and is 1 where x>=0.

To better represent a perceptron, one can create a perceptron class with the activation function being the hard lim function.

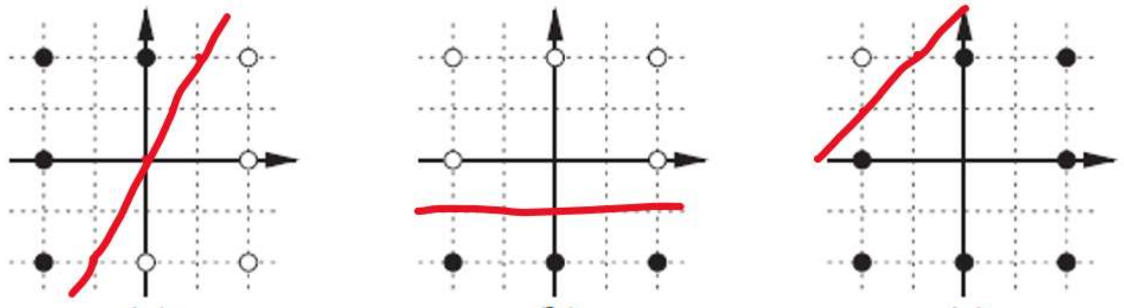
Importing the Necessary Libraries

In [ ]:
```
import numpy as np
```

Now that the library have been imported, the next step is to create the perceptron class.

In [ ]:
```
class Perceptron(object):
    """Simple Perceptron

    This is a representation of a perceptron within an Artificial Neural Network.

    Parameters
    ----------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Number of times algorithm passes over the training data set.
    random_state : int
        Random number generator seed for random weight initialization.

    Attributes
    ----------

    w : 1D-array
        Weights after fitting.

    errors : list
        Number of misclassifications, or updates, in each epoch.
    """
    def __init__(self, eta=0.1, n=50, random_state=1):
        self.eta = eta
        self.n = n
        self.random_state = random_state


    def fit(self, X, y):
        """Trains the model based on  the given data for making predictions

        Parameters
        ----------

        X : array-like, shape = [samples, features]
            - Vectors used for training the model.
            - length of the vectors is equivalent to the number of features.

        y : array-like, shape = [samples]
            - Target values.
        """
        rgen = np.random.RandomState(self.random_state)
        self.w = rgen.normal(loc=0.0, scale=0.01, size=1+X.shape[1])
        self.errors = []
        for i in range(self.n):
            errors = 0
            for xi, target in zip(X,y):
                update = self.eta * (target - self.predict(xi))
                self.w[1:] += update * xi
                self.w[0] += update
                errors += int(update != 0.0)
            self.errors.append(errors)
        return self


    def net_input(self, X):
        """Calculate the net input.

        This is where the sum of the products between the weights and the inputs are ca
        """
        return np.dot(X, self.w[1:]) + self.w[0]


    def activation(self, X):
        """Activation function."""
        return 1.0 if self.net_input(X) >= 1 else 0.0


    def predict(self, X):
        """Return class label after unit step."""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

The code above is an example of a perceptron with a hard limit function as its activation function. Now moving on to the decision boundaries, we draw the decision boundaries in red marker (as shown below)



To find the weights and bias of the single-neuron perceptron, one must first consider that the points where the single neuron perceptron's decision boundaries intercepts are where the function is equal to 0.

Taking into account part a, the net input function is $f(x_1, x_2) = w_1x_1 + w_2x_2 + b = 0$

When $x_1 = 0$ and $x_2 = 0$, then $f(x_1, x_2) = 0$. This then allows us to learn the value of the bias (which is 0) and in turn convert the equation to the following function:

$$f(x_1, x_2) = w_1x_1 + w_2x_2 = 0$$

Now the nature of the decision boundary is such that $x_2 = 2x_1$

This means that the weights can be considered to be the following:

- $w_1 = -2$
- $w_2 = 1$

Taking into account part b, the net input function is $f(x_1, x_2) = w_1x_1 + w_2x_2 + b = 0$

Based on the nature of the boundary shown on b, one can immediately tell that it is of the nature y=x where y is a constant (x = -1).

One can then reformulate the equation for part b above and obtain the following formula:

$$f(x_1, x_2) = w_1x_1 + w_2(-1) + b = 0$$

$$f(x_1, x_2) = w_1x_1 - w_2 + b = 0$$

Now we substitute for the point where $x_1 = 0$ and obtain the following formula:

$$w_2 = b$$

substituting again, one learns that the weight $w_1 = 0$ and that the second weight is $w_2 = 1$ and the bias is 1. We obtain the formula below based on the aformentioned:

$$f(x_1, x_2) = 0 * x_1 - 1 + 1 = 0$$

Taking into account part c, the net input function is $f(x_1, x_2) = w_1x_1 + w_2x_2 + b = 0$

There are two points where the decision boundary intercepts the parameters x_1 and x_2. These two points are (-3,0) and (0,3).

Substituting for the first point, we obtain the following equation:

$-3w_1 + b = 0$

$w_1 = b/3$

$3w_2 + b = 0$

$w_2 = -b/3$

Once can select any arbitrary bias which will provide one with the weights. For the sake of convenience, lets allow the bias to equal 3, this allows the weights to be the following:

$w_1 = 1$

$w_2 = -1$

Now to find the variance one must first calculate the difference between predicted values and and the actual values of a data set. Once that difference has been discovered, one squares the sum of the differences and divides by the number of data points.

In [ ]:

```
perceptron_a = Perceptron()
X = np.array(
    [
        [-2, -2],
        [-2, 0],
        [-2, 2],
        [0, -2],
        [0, 2],
        [2, -2],
        [2, 0],
        [2, 2]
    ]
)

y_a = np.array(
    [
        [1],
        [1],
        [1],
        [0],
        [1],
        [0],
        [0],
        [0],
    ]
)
perceptron_a.fit(X, y_a)

perceptron_b = Perceptron()

y_b = np.array(
    [
        [1],
        [0],
        [0],
        [1],
        [0],
        [1],
        [0],
        [0],
    ]
)

perceptron_b.fit(X, y_b)

perceptron_c = Perceptron()

y_c = np.array(
    [
        [1],
        [1],
        [0],
        [1],
        [1],
        [1],
        [1],
        [1],
    ]
)

perceptron_c.fit(X, y_c)
```

Out[ ]:

```
<__main__.Perceptron at 0x286f8243f40>
```

Now that the three models have been completed and trained, one should be able to make accurate predictions. One can compare with any new point or value as one also knows the decision boundary.

In [ ]:

```
prd_a = perceptron_a.predict(np.array([1,1]))
prd_b = perceptron_b.predict(np.array([1,1]))
prd_c = perceptron_c.predict(np.array([-2,3]))

print(f"The estimate for model a is {prd_a}")
print(f"The estimate for model b is {prd_b}")
print(f"The estimate for model c is {prd_c}")
```

```
The estimate for model a is 1
The estimate for model b is 1
The estimate for model c is 1
```

As shown above, the model is highly inaccurate, this is due to the fact that there are little to no data points to properly create the model.