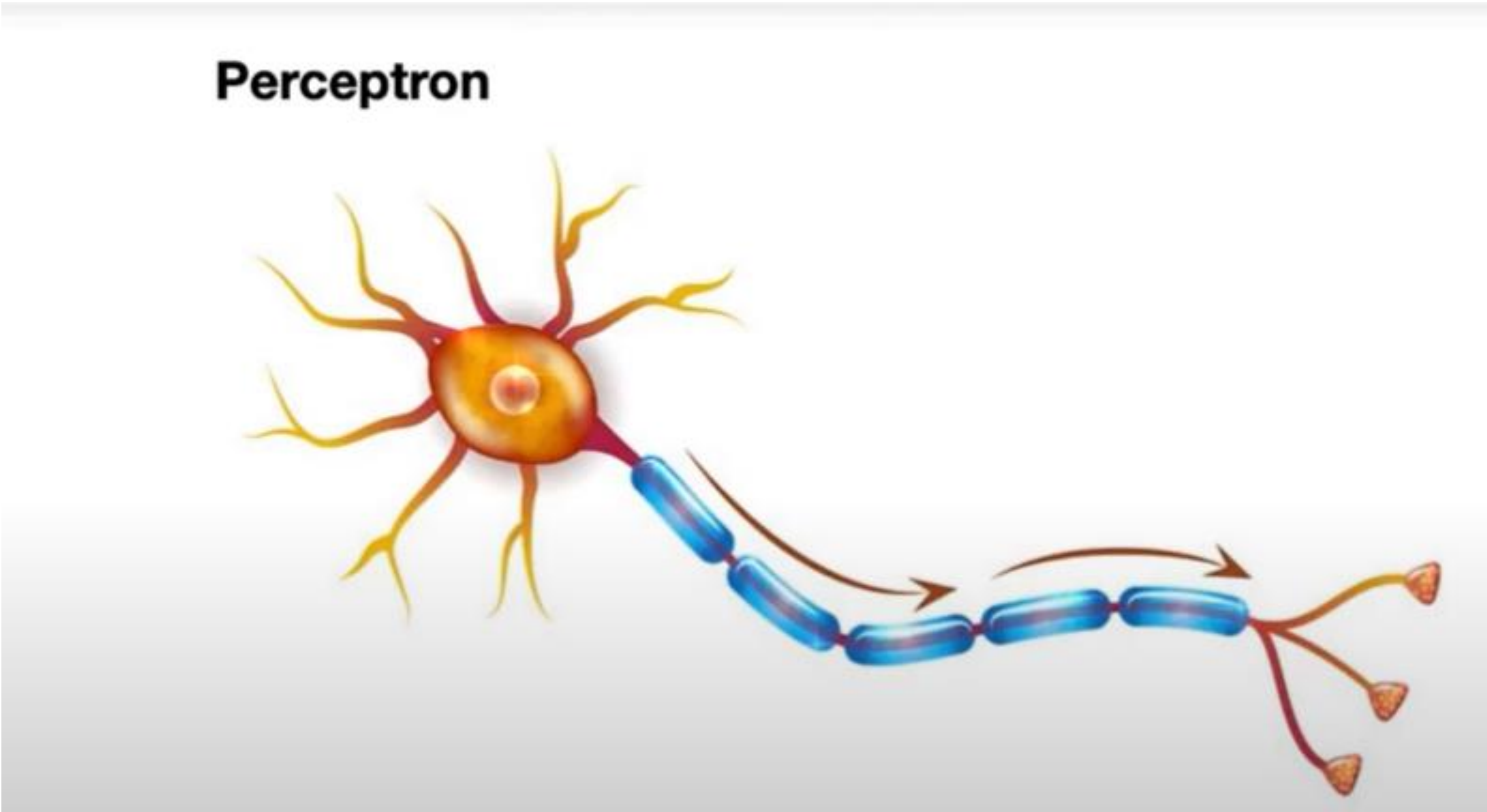
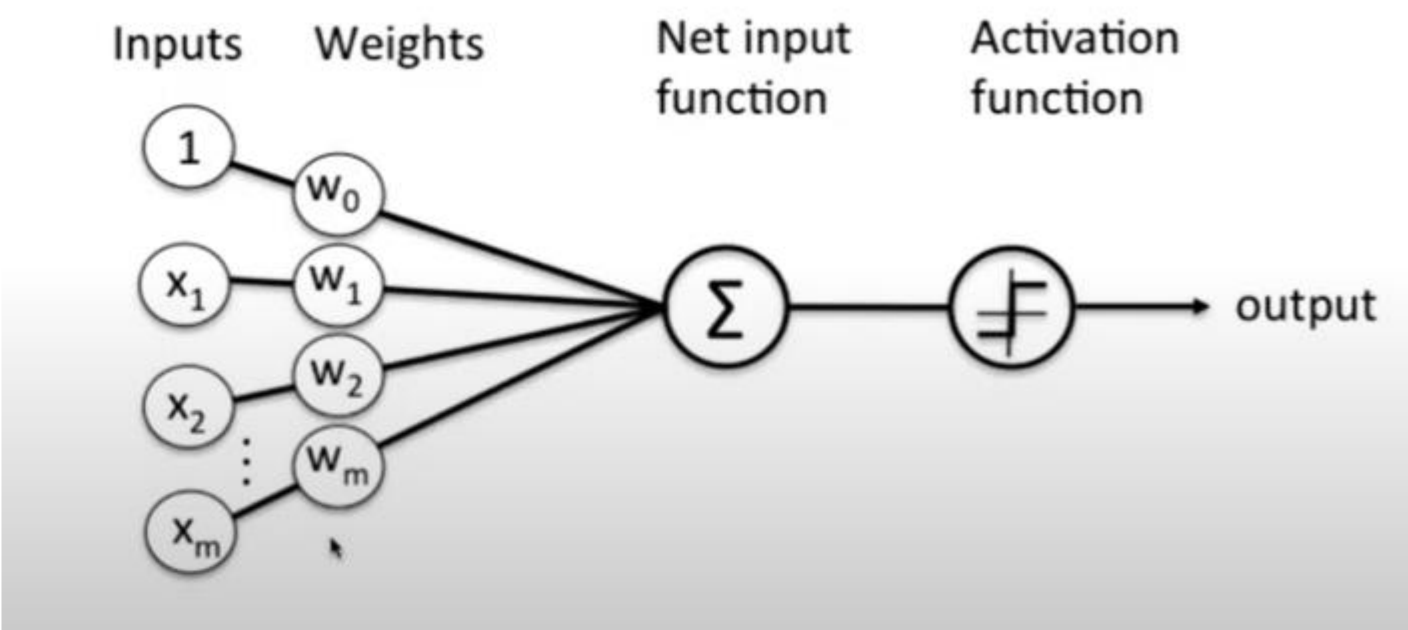
Perceptron



The perceptron can be viewed as one single unit of an artificial neural network.

Signal reaches a threshold it will fire a 0 or a 1.



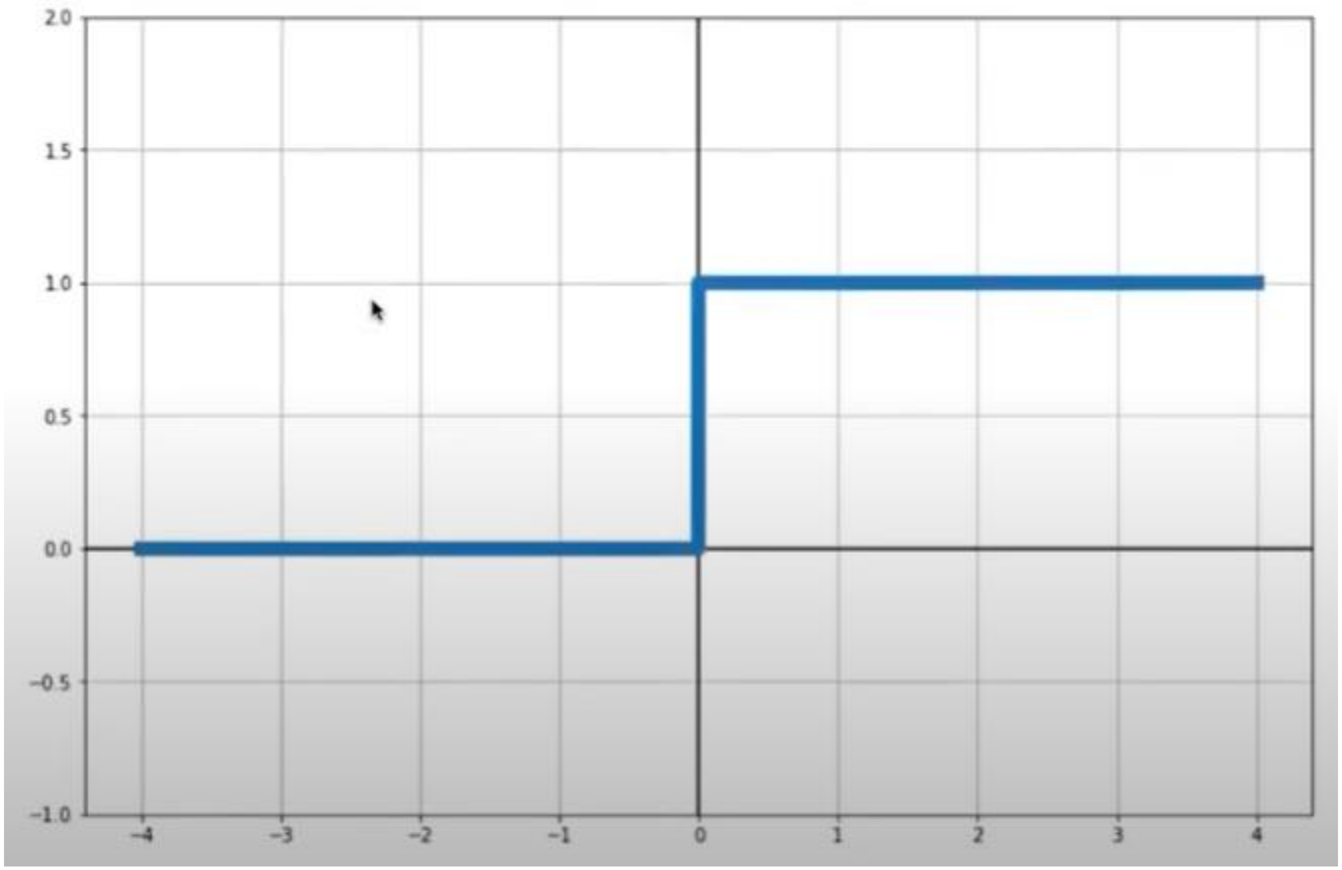
Linear Model

$f(w, b) = w^T x + b$ w transposed, the bias (or zero intercept) is w0.

Activation Function

Unit step function

$$g(\mathbf{z}) = \begin{cases} 1 & \text{if } \mathbf{z} \geq \theta \\ 0 & \text{otherwise.} \end{cases}$$



Approximation

$$\hat{y} = g(f(w, b)) = g(w^T x + b)$$

Perceptron update rule

For each training sample x_i :

$$w := w + \Delta w$$

$$\Delta w := \alpha \cdot (y_i - \hat{y}_i) \cdot x_i$$

α : learning rate in [0, 1]

For each sample apply the update step. This is the new weight is the old weight plus the delta weight.

Delta weight is the actual label – predicted label value times the training sample and multiplied by alpha, a learning rate between values 0 and 1.

The four possible cases in a two class problem

Update rule explanation

y	$\hat{y}$	$y - \hat{y}$
1	1	0
1	0	1
0	0	0
0	1	-1

CODE AND RESULTS USING LEARNING KIT

```
#PERCEPTRON IN PYTHON

import numpy as np

#Perceptron implementation
class Perceptron:
    def __init__(self,learning_rate=0.01, n_iters=1000):
        self.lr = learning_rate
        self.n_iters = n_iters
        self.activation_func = self._unit_step_func
        self.weight = None
        self.bias = None

    #the activation function is the unit step function
    def _unit_step_func(self,x):
        return np.where(x>=0,1,0)

    def fit(self,X,y): #gets the training samples X and training labels
        n_samples, n_features = X.shape
        #init weights
        self.weight = np.zeros(n_features)
        self.bias = 0
        #we only accept classes 0 and 1
        y_ = np.array([1 if i > 0 else 0 for i in y ])
        for _ in range(self.n_iters):
            for idx,x_i in enumerate(X):
                linear_output = np.dot(x_i,self.weight) + self.bias
                y_predicted = self.activation_func(linear_output)
```

```python
                update = self.lr * (y_[idx] - y_predicted)
                self.weight += update * x_i
                self.bias += update


    def predict(self, X): #gets test samples
        linear_output = np.dot(X,self.weight ) + self.bias
        y_predicted = self.activation_func(linear_output)
        return y_predicted

##################################################################
from sklearn import datasets
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

def accuracy(y_true, y_pred):
    accuracy = np.sum(y_true == y_pred) / len(y_true)
    return accuracy

X,y = datasets.make_blobs(n_samples=150,n_features=2,centers=2,cluster_std=1.05,random_state=2)
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_state=123)

p = Perceptron(learning_rate=0.01,n_iters=1000)
p.fit(X_train,y_train)
predictions = p.predict(X_test)
print("Preceptron classification accuracy ", accuracy(y_test,predictions))
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
plt.scatter(X_train[:,0],X_train[:,1],marker = 'o', c=y_train)
```

```
x0_1 = np.amin(X_train[:,0])
x0_2 = np.amax(X_train[:,0])
x1_1 = (-p.weight[0] * x0_1 - p.bias)/ p.weight[1]
x1_2 = (-p.weight[0] * x0_2 - p.bias)/ p.weight[1]

ax.plot([x0_1,x0_2],[x1_1,x1_2] ,'k' )
ymin = np.amin(X_train[:,1])
ymax = np.amax(X_train[:,1])
ax.set_ylim([ymin-3,ymax+3])
plt.show()
```

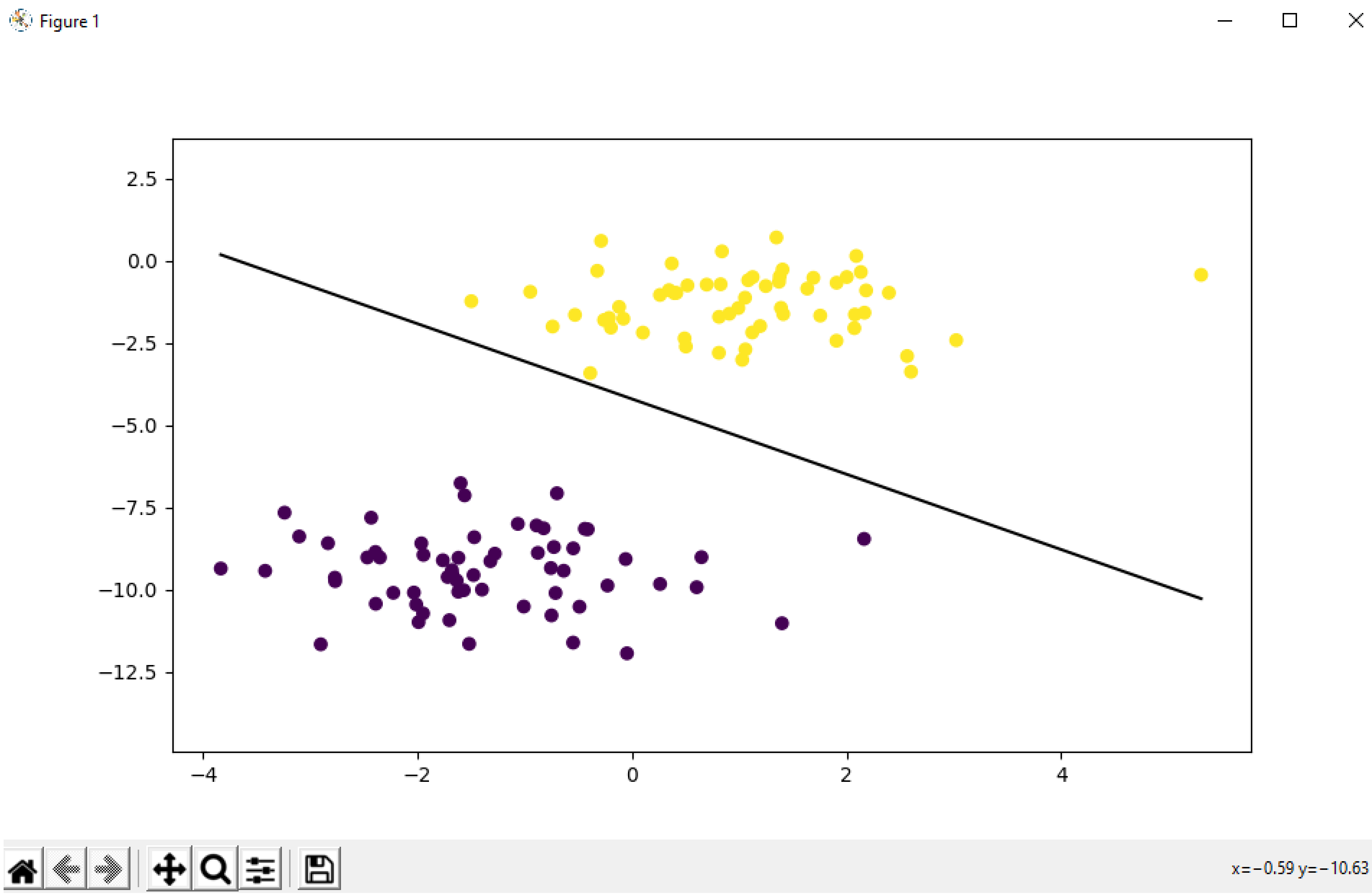


Figure 1
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−4
−2
0
2
4
x=−0.59 y=−10.63