# McCulloch–Pitts Neurons and Perceptron Learning

## 1. McCulloch–Pitts neuron

```
(a) Implement a McCulloch–Pitts neuron (see the diagram):
        y(x) = sgn(wTx)
    where x = [-1; x1; x2; : : : ; xk] is a vector of inputs, w = [w0;w1;w2; : : : ;wk] is a vector of
    weights and y(x) is the output.
```

In [385]:
```python
import numpy as np
def mcCullochPittsNeuron(w, x):
    x = np.insert(x, 0, -1, axis=1)
    h = np.dot(w, x.T) # matching the dimensions
    return (x, np.sign(h))
```

```
(b) Take weights w = [3; 2; 2] and two binary inputs x1, x2 in { -1, +1 }. Show that the neuron
    performs a logical AND operation.
```

In [390]:
```python
w = np.array([3,2,2])
x = np.array([[-1,-1],[-1, 1], [1, -1], [1, 1]])
(x1,y1) = mcCullochPittsNeuron(w,x)
for i in range(y1.size):
    print("AND(", x[i],") = ", y1.T[i])
```

```
AND( [-1 -1] ) =  -1
AND( [-1  1] ) =  -1
AND( [ 1 -1] ) =  -1
AND( [1 1] ) =  1
```

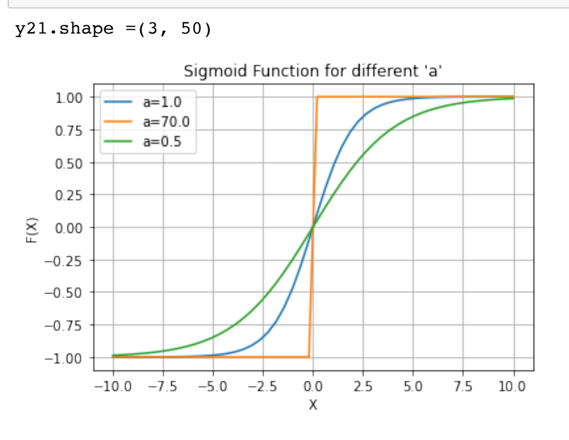
```
    since the output is 1 only when both the input are 1, it proves that the mcCullochPittsNeuron can implement
AND logic gate
```

## 2. Activation functions

```
(a) Sigmoid function:
```

In [139]:
```python
import matplotlib.pyplot as plt
def sigmoid_function(a, x):
    y21 = (2/(1+ np.exp(-1*a*x)))-1
    return y21

a=np.array([1, 70, 0.5])
x = np.linspace(-10, 10)
y21 = sigmoid_function(a[:, np.newaxis], x[np.newaxis, :])
print(f'{y21.shape =}')
for i in range(a.size):
    plt.plot(x, y21[i, :], label=str('a='+ str(a[i])))

plt.grid()
plt.title("Sigmoid Function for different 'a'")
plt.xlabel('X')
plt.ylabel('F(X)')
plt.legend()
plt.show()
```

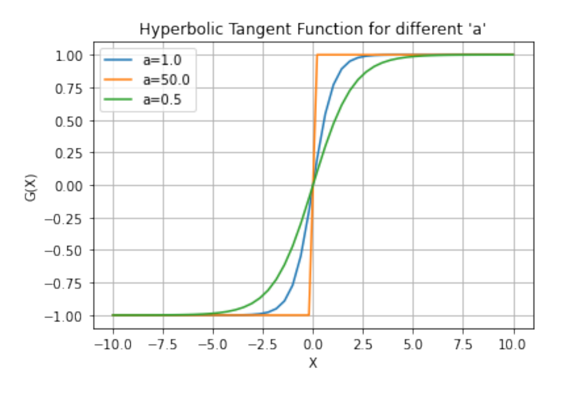
```
y21.shape =(3, 50)
```




```
(b) Hyperbolic tangent function:
```

In [412]:
```python
import matplotlib.pyplot as plt
def sigmoid_function(a, x):
    y22 = np.tanh(a*x)
    return y22

a=np.array([1, 50, 0.5])
x = np.linspace(-10, 10)
y22 = sigmoid_function(a[:, np.newaxis], x[np.newaxis, :])
print(f'{y22.shape =}')
for i in range(a.size):
    plt.plot(x, y22[i, :], label=str('a='+ str(a[i])))

plt.grid()
plt.title("Hyperbolic Tangent Function for different 'a'")
plt.xlabel('X')
plt.ylabel('G(X)')
plt.legend()
plt.show()
```

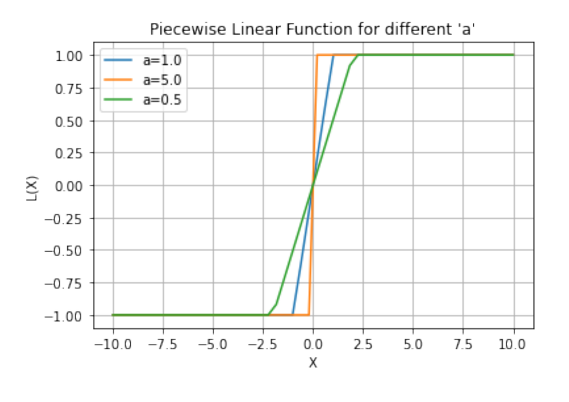
```
y22.shape =(3, 50)
```




```
(c) Piecewise linear function
```

In [310]:
```python
import matplotlib.pyplot as plt
def piecewise_linear_function(aa, x):
    return np.array([np.piecewise(x, [x <= -1/a, (x>-1/a) & (x<1/a), x >= 1/a], [-1, lambda x: a*x, 1]) for a in aa])

a = np.array([1, 5, 0.5])
x = np.linspace(-10, 10)
y23 = piecewise_linear_function(a, x)
print(f'{x.shape =}')
print(f'{y23.shape =}')
for i in range(a.size):
    plt.plot(x, y23[i,:].T, label=str('a='+ str(a[i])))

plt.grid()
plt.title("Piecewise Linear Function for different 'a'")
plt.xlabel('X')
plt.ylabel('L(X)')
plt.legend()
plt.show()
```

```
x.shape =(50,)
y23.shape =(3, 50)
```




By studying these activation function we can say that 'a' act as the scaler for non linearity for all these functions. Decreasing 'a' < 1 (but a>0), increases the non linearity of the activation functions , whereas increasing the 'a' > 1 makes the graph *steeper*. With the increase in 'a' all of these acivation function converges to the step function( though the value might be different for different functions).

For (a) Sigmoid function: if a>=70 it becomes step function, (b) Hyperbolic tangent function: if a>=50 it becomes step function, (c) Piecewise linear function: if a>=5 it becomes step function.

## 3. Rosenblatt's perceptron

```
(a) Preparing training set
```

In [393]:
```python
import numpy as np

def desired_result(x1,x2):
    if x2 >= 0.5 - x1:
        return 1
    else:
        return -1

def getTrainingSet():
    x31 = np.empty([1000, 3])
    d = np.ones([1000, 1])
    for i in range(1000):
        x31[i][0] = -1
        x = np.random.normal(size=2)
        x31[i][1] = x[0]
        x31[i][2] = x[1]
        d[i] = desired_result(x[0],x[1])
    return x31, d
```

```
(b) Train a McCulloch–Pitts neuron
```

In [355]:
```python
def getRandomWeights():
    return np.random.rand(1, 3)


def simplifiedMcCullochPittsNeuron(w, x):  # not using the 1st exercise since there, inside the method we are insertin
g the -1 as the first element
    h = np.dot(w, x)  # matching the dimensions
    return np.sign(h)
```

In [501]:
```python
def training_using_McCullochPittsNeuron(lr, x, w, dy):
    for j in range(lr.size):
        print('------------------------')
        print('learning rate: ' + str(lr[j]))
        w_new = np.zeros([3, 1])
        epoch_loop = 0
        w_example = w.copy()
        while np.all(w != w_new) and epoch_loop < 1000:
            w = w_new.copy()
            for i in range(dy.size):
                y = simplifiedMcCullochPittsNeuron(w_example, x[i])
                w_example = w_example + lr[j] * (dy[i] - y) * x[i]
            w_new = w_example.copy()
            epoch_loop = epoch_loop + 1
        print('final wieght vector: ' + str(w_new))
        if(epoch_loop<1000):
            print('Epoch Count for Convergence: ' + str(epoch_loop))
        else :
            print('Did not Convergence even after Epoch Count: ' + str(epoch_loop))
    return w_new
```

In [499]:
```python
np.random.seed(1)
(x31, d) = getTrainingSet()
w_old = getRandomWeights()
print('initial weight vector: ' + str(w_old))
eta_learningRate = np.array([.0001, 1, 13])
w_new = training_using_McCullochPittsNeuron(eta_learningRate, x31, w_old, d)
training_output_weight_vector = w_new.copy()
```

```
initial weight vector: [[0.87919834 0.16880019 0.79354773]]
------------------------
learning rate: 0.0001
final wieght vector: [[0.33519856 0.67243444 0.6681146 ]]
Epoch Count for Convergence: 7
------------------------
learning rate: 1.0
final wieght vector: [[0.33519856 0.67243444 0.6681146 ]]
Epoch Count for Convergence: 2
------------------------
learning rate: 13.0
final wieght vector: [[0.33519856 0.67243444 0.6681146 ]]
Epoch Count for Convergence: 2
```

```
    What values for eta(learning rate) do you use, and why?
```

The learning rate(eta) should be optimal and should be determined by the convergence rate. If eta is too small, then the gradient descent can slow down. If eta is too large, gradient descent can overshoot the minimum. It may even fail to converge. Failure to converge or too many iterations to obtain the minimum value would imply that our eta value is incorrect. Thus I have chosen eta as 1 in this case.

```
(c)Test on a new dataset(validation set) that the neuron can indeed perform the trained comparison function.
```

In [496]:
```python
def getValidationSet():
    x = np.random.rand(11,2)
    return x
```

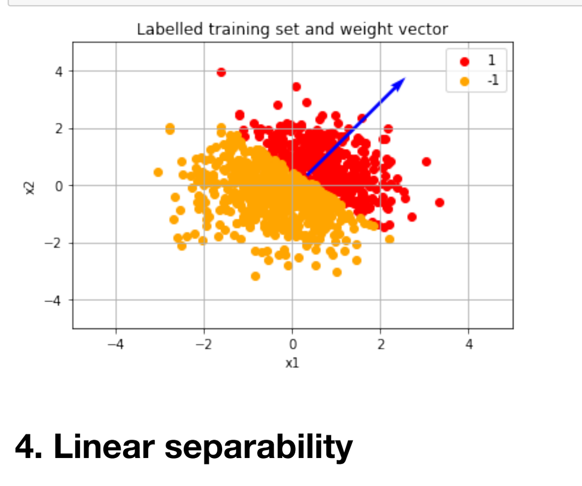
In [521]:
```python
print(f'{training_output_weight_vector = }')
x32 = getValidationSet()
(x32,y32) = mcCullochPittsNeuron(training_output_weight_vector, x32)
for i in range(y32.size):
    pass_validation = (y32.T[i] == desired_result(x32[i][1], x32[i][2]))
    print(x32[i], ' x ', training_output_weight_vector , ' = ', y32.T[i], ' : ' , pass_validation)
```

```
training_output_weight_vector = array([[0.33519856, 0.67243444, 0.6681146 ]])
[-1.  [illegible]  [illegible]] x [[0.33519856 0.67243444 0.6681146 ]] = [1.] : [ True]
[-1.  [illegible]  [illegible]] x [[0.33519856 0.67243444 0.6681146 ]] = [1.] : [ True]
[-1.  [illegible]  [illegible]] x [[0.33519856 0.67243444 0.6681146 ]] = [-1.] : [ True]
[-1.  [illegible]  [illegible]] x [[0.33519856 0.67243444 0.6681146 ]] = [1.] : [ True]
[-1.  [illegible]  [illegible]] x [[0.33519856 0.67243444 0.6681146 ]] = [1.] : [ True]
[-1.  [illegible]  [illegible]] x [[0.33519856 0.67243444 0.6681146 ]] = [1.] : [ True]
[-1.  [illegible]  [illegible]] x [[0.33519856 0.67243444 0.6681146 ]] = [1.] : [ True]
[-1.  [illegible]  [illegible]] x [[0.33519856 0.67243444 0.6681146 ]] = [1.] : [ True]
[-1.  [illegible]  [illegible]] x [[0.33519856 0.67243444 0.6681146 ]] = [-1.] : [ True]
[-1.  [illegible]  [illegible]] x [[0.33519856 0.67243444 0.6681146 ]] = [1.] : [ True]
[-1.  [illegible]  [illegible]] x [[0.33519856 0.67243444 0.6681146 ]] = [1.] : [ True]
```

This shows that the training output weight vector is working perfectly for validation set

```
(d)Plot the training set and label each input vector according to its response class. Superimpose the weight vec
tor on the same plot (think about the bias term w0). Explain in what sense the weight vector is optimal.
```

In [346]:
```python
# Create an empty list
x341 = []
x342 = []

for i in range(d.size):
    if(d[i] == 1):
        x341.append(x31[i])
    else:
        x342.append(x31[i])
x341 = np.array(x341)
x342 = np.array(x342)

plt.scatter(x=x341[:,1], y=x341[:,2], c='red', label='1')
plt.scatter(x=x342[:,1], y=x342[:,2], c='orange', label='-1')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Labelled training set and weight vector')
plt.grid()
plt.xlim(-5,5)
plt.ylim(-5,5)
plt.legend()

max_theta = training_output_weight_vector.T[0]
max_weight_x1 = training_output_weight_vector.T[1]
max_weight_x2 = training_output_weight_vector.T[2]

origin = np.array([max_theta, max_theta]) # origin point
plt.quiver(*origin, max_weight_x1, max_weight_x2, color=['b'], scale=3)
plt.show()
```




## 4. Linear separability

```
(a) Train a perceptron to perform the XOR function on binary inputs.
```

In [498]:
```python
def xor(x):
    d = np.zeros(len(x))
    for i in range(len(x)):
        if x[i][1] == x[i][2] :
            d[i] = 0
        else:
            d[i] = 1
    return x, d
```

In [557]:
```python
np.random.seed(1)
x_41 = np.random.choice([0,1], size=(1000,2))
x_41 = np.insert(x_41, 0, -1, axis=1)
(x_41, d_41) = xor(x_41)
w_old_41 = getRandomWeights()
print('initial weight vector: ' + str(w_old_41))
eta_learningRate_41 = np.array([.0001, 1, 13])
w_new_41 = training_using_McCullochPittsNeuron(eta_learningRate_41, x_41, w_old_41, d_41)
training_output_weight_vector_41 = w_new_41.copy()
```

```
initial weight vector: [[0.32580997 0.88982734 0.75170772]]
------------------------
learning rate: 0.0001
final wieght vector: [[ 9.96661321e-06  2.73414475e-05 -2.92279081e-04]]
Epoch Count for Convergence: 38
------------------------
learning rate: 1.0
final wieght vector: [[-9.00333868e-05  2.00002734e+00 -9.22790807e-05]]
Did not Convergence even after Epoch Count: 1000
------------------------
learning rate: 13.0
final wieght vector: [[ 9.99909967e-01  2.73414475e-05 -2.80000923e+01]]
Did not Convergence even after Epoch Count: 1000
```

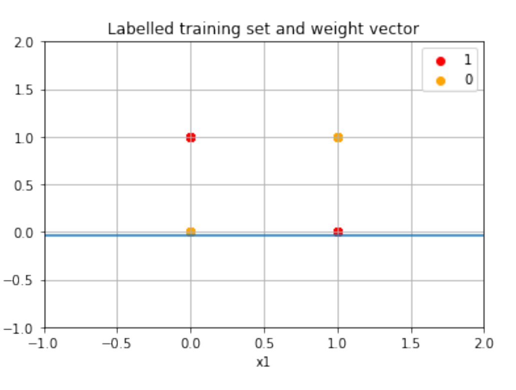
```
(b) Show that the learning algorithm does not converge, i.e., the weights do not settle on fixed values.
```

In the above example, we have trained the model for XOR logic gate. And also checked that the weights are not converging.

```
(c) Plot the XOR classification problem on an Euclidean plane. Explain why the problem is not
linearly separable.
```

In [558]:
```python
x431 = []
x432 = []

for i in range(d_41.size):
    if(d_41[i] == 1):
        x431.append(x_41[i])
    else:
        x432.append(x_41[i])
x431 = np.array(x431)
x432 = np.array(x432)

plt.scatter(x=x431[:,1], y=x431[:,2], c='red', label='1')
plt.scatter(x=x432[:,1], y=x432[:,2], c='orange', label='0')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Labelled training set and weight vector')
plt.grid()
plt.xlim(-1,2)
plt.ylim(-1,2)
plt.legend()

print(training_output_weight_vector_41)
max_theta = training_output_weight_vector_41.T[0]
max_weight_x1 = training_output_weight_vector_41.T[1]
max_weight_x2 = training_output_weight_vector_41.T[2]

x = np.linspace(-2,2,100)
y = (-max_weight_x1/max_weight_x2)*x +max_theta/max_weight_x2

plt.plot(x,y)

plt.show()
```

```
[[ 9.99909967e-01  2.73414475e-05 -2.80000923e+01]]
```




There is no one line, that can actually act as the decision boundary and seperate out 0 and 1 labelled data. The decision boundary the above plot that is trying to plot is invalid.