McCulloch–Pitts Neurons and Perceptron Learning

Models of Neural Systems, WS 2021/22; Computer Practical 1

Gonzalo Cardenal, Sofia Pereira da Silva

1. McCulloch–Pitts neuron

PTK: Please finish by re-running all the cells prior to submission, so that the cell count starts at [1]. Otherwise at some point there'll be old 'ghost' variables in your work space/kernel that are no more defined, but which you still might be accidentally using in the code.

In [65]:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA
```

a)

Implement a McCulloch-Pitts neuron

$$y(x) = sgn(w^t x)$$

where $x = [-1, x_1, x_2, \ldots, x_k]$ is a vector of inputs, $w = [w_0, w_1, w_2, \ldots, w_k]$ is a vector of weights and $y(x)$ is the output.

In [66]:

```
def mcneuron(inputs, weights):
    y = np.sign(np.dot(weights.T, inputs))
    return y
```

b)

Take weights w = [3; 2; 2] and two binary inputs x_1 and x_2 e {-1, +1}. Show that the neuron performs a logical AND operation.

-> To show that this neuron is compututing the AND operation, we applied it to all possible combinations of input x_1 and x_2, getting a result correspondent to the truth table known for this logical function (only fires if both are +1):

In [67]:

```
w = np.array([3,2,2])

x_table = np.array([[-1,1,1], [-1,1,-1], [-1,-1,1], [-1,-1,-1]])
results_table = []
for xi in x_table:
    results_table.append(mcneuron(xi, w))
print(results_table)
```

```
[1, -1, -1, -1]
```

PTK: Very nice.

2. Activation functions

2.1. a) Sigmoid function

$$f(x) = \frac{2}{1 + e^{-ax}} - 1$$

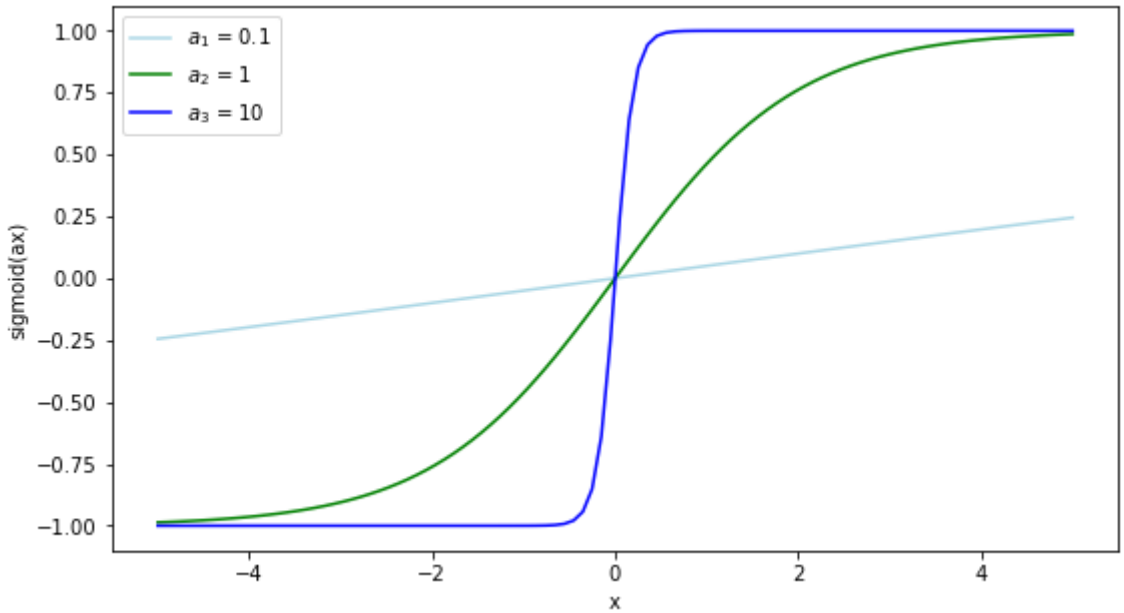
In [68]:

```
def sigmoid(a, x):
    y = 2/(1+np.exp(-a*x)) -1
    return y
```

In [69]:

```
x = np.linspace(-5,5, 100)
plt.plot(x, sigmoid(0.1,x), c = 'lightblue', label = "$a_1$ = 0.1")
plt.plot(x, sigmoid(1,x), c = 'green', label = "$a_2$ = 1")
plt.plot(x, sigmoid(10,x), c ='blue', label = "$a_3$ = 10")
plt.xlabel("x")
plt.ylabel("sigmoid(ax)")
plt.legend()
plt.show()
```

Out[69]:



b)Hyperbolic tangent function

$$g(x) = tanh(ax)$$

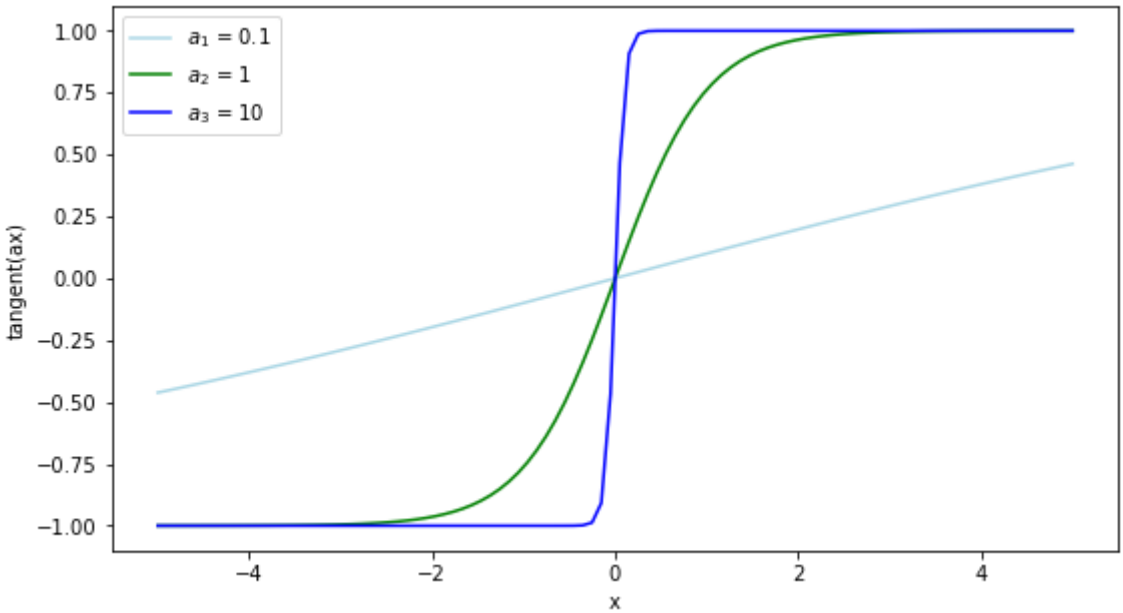
In [70]:

```
def tangent(a,x):
    y = np.tanh(a*x)
    return y
```

In [71]:

```
x = np.linspace(-5,5, 100)
plt.plot(x, tangent(0.1,x), c = 'lightblue', label = "$a_1$ = 0.1")
plt.plot(x, tangent(1,x), c = 'green', label = "$a_2$ = 1")
plt.plot(x, tangent(10,x), c ='blue', label = "$a_3$ = 10")
plt.xlabel("x")
plt.ylabel("tangent(ax)")
plt.legend()
plt.show()
```

Out[71]:



c) Piecewise linear function

$$l(x) = \begin{cases} 1 & \text{if } x \geq \frac{1}{a} \\ -ax & \text{if } -\frac{1}{a} < x < \frac{1}{a} \\ -1 & \text{if } x \leq \frac{1}{a} \end{cases}$$

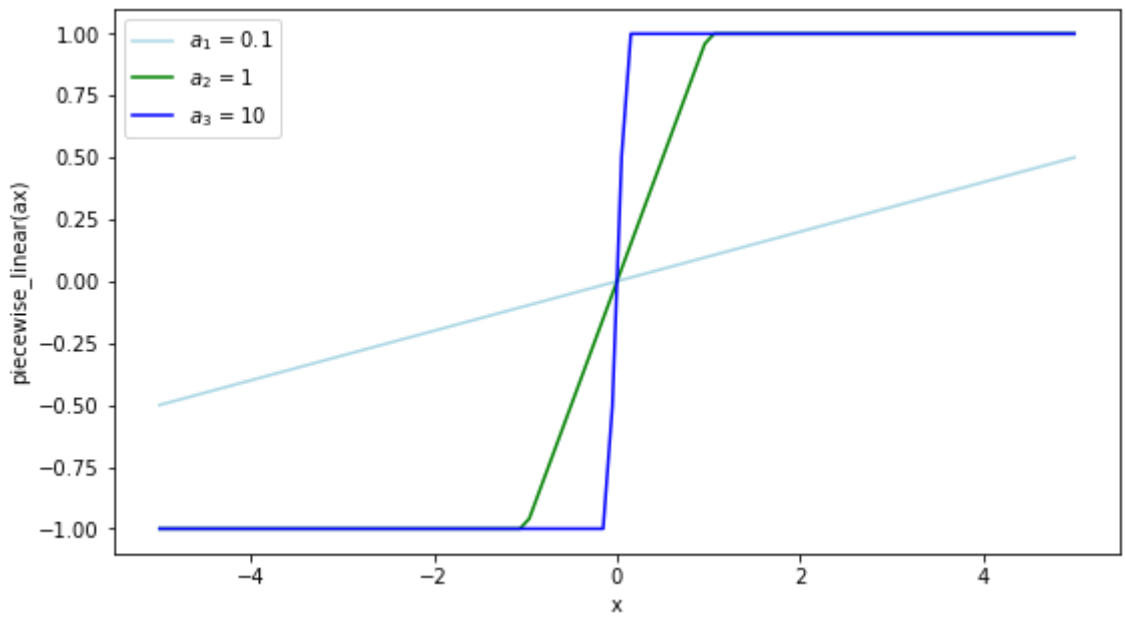
In [72]:

```
def piecewise_linear(a,x):
    y = []
    for xi in x:
        if xi >= 1/a:
            y.append(1)
        elif -1/a < xi < 1/a:
            y.append(a*xi)
        else:
            y.append(-1)
    return np.array(y)
```

In [73]:

```
x = np.linspace(-5,5, 100)
plt.plot(x, piecewise_linear(0.1,x), c = 'lightblue', label = "$a_1$ = 0.1")
plt.plot(x, piecewise_linear(1,x), c = 'green', label = "$a_2$ = 1")
plt.plot(x, piecewise_linear(10,x), c ='blue', label = "$a_3$ = 10")
plt.xlabel("x")
plt.ylabel("piecewise_linear(ax)")
plt.legend()
plt.show()
```

Out[73]:



-> By plotting the different activation functions for different values of a (as an example, we consistently chose 0.1, 1 and 10), it is possible to see that a corresponds to the "slope"/ "transition rate" of the activation function. The bigger a is, the faster is the transition from -1 to 1. If 0 < a <<1, then the function approximates a linear function, if a >> 1, it approximates the step function.

2.2

--> Therefore, if a is large enough (for example, a = 40 in the following plot), then all of the functions approximate the step function (because they transition from -1 to 1 very quickly):

In [74]:

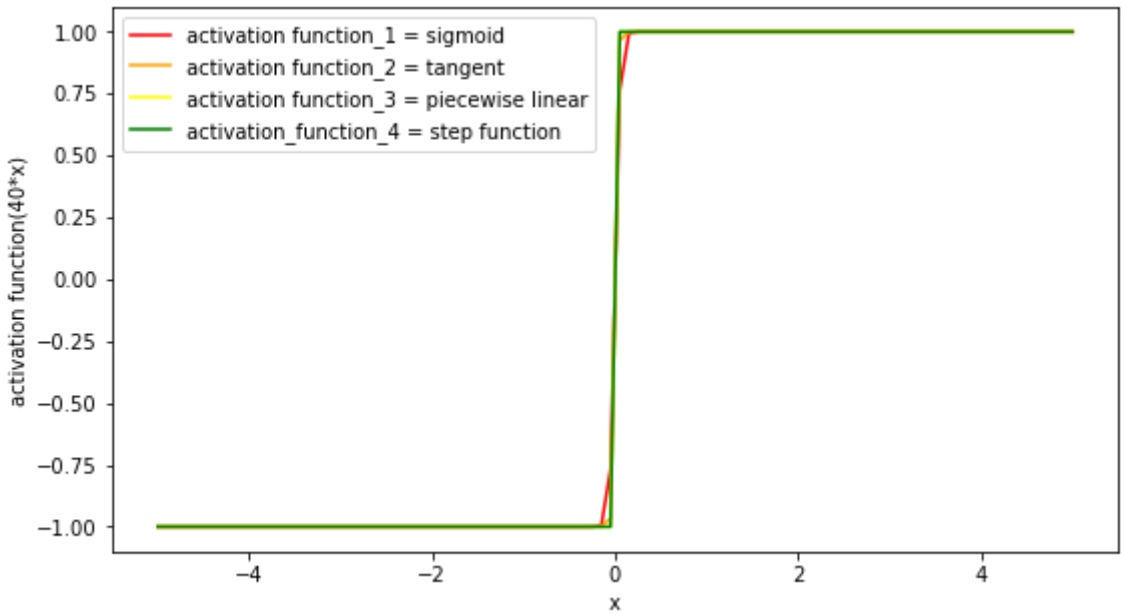
```
def step_function(x):
    y = []
    for xi in x:
        if xi >= 0:
            y.append(1)
        else:
            y.append(-1)

    return np.array(y)
```

In [75]:

```
x = np.linspace(-5,5, 100)
plt.rcParams["figure.figsize"] = (9,5)
plt.plot(x, sigmoid(40,x), c ='red', label = "activation function_1 = sigmoid")
plt.plot(x, tangent(40,x), c = 'orange', label = "activation function_2 = tangent")
plt.plot(x, piecewise_linear(40,x), c ='yellow', label = "activation function_3 = pi
plt.plot(x, step_function(x), c = "green", label = "activation_function_4 = step fur
plt.xlabel("x")
plt.ylabel("activation function(40*x)")
plt.legend(loc=2)
plt.show()
```

Out[75]:



PTK: Good work!

3. Rosenblatt's perceptron

a)

Prepare a training set $\{x_i, d(x_i)\}$ for $i = 1, 2, \ldots, 1000$, where $x_i = [-1, x_{i,1}, x_{i,2}]$ is an input vector with random values following a standard normal distribution. d(x) is the desired response, which is a comparision betweeen the coordinates x_1 and x_2 :

$$d(x) = \begin{cases} 1 & \text{if } x_2 \geq 0.5 - x_1, \\ -1 & \text{if } x_2 < 0.5 - x_1, \end{cases}$$

We defined functions to get the random input variables x_{i,1}, x_{i,2} and, separately, compute the desired output for them (get_dvalues using the function d(x)). Finally, we "combined" both arrays using np.hstack in order to get a single training_set array with all the information:

In [76]:

```
def get_xvalues(n):
    x = []
    for i in range(1,1001):
        xi1 = np.random.randn()
        xi2 = np.random.randn()
        xi = np.array([-1, xi1, xi2])
        x.append(xi)
    return np.array(x)

x = get_xvalues(1000)
```

In [77]:

```
def d(x):
    y = []
    if x[2] >= 0.5 - x[1]:
        y.append(1)
    else:
        y.append(-1)
    return np.array(y)
```

In [78]:

```
def get_dvalues(x):
    d_values = []
    for i in x:
        d_values.append(d(i))
    return np.array(d_values)

d_values = get_dvalues(x)
```

In [79]:

```
training_set = np.hstack((x, d_values))
print(training_set[:5,:])
print(training_set.shape)
```

```
[[-1.         -0.17912238 -0.51872877 -1.        ]
 [-1.         -0.6496307   0.76402445 -1.        ]
 [-1.          0.33957673 -0.22789532 -1.        ]
 [-1.          1.07453678  0.08229654  1.        ]
 [-1.          0.455002   -0.4202677  -1.        ]]
(1000, 4)
```

-> As an example, we printed the first 5 rows of our training set array. One can observe the four collumns, corresponding, respectively, to the input vector with the bias (=-1 for all) and 2 random inputs x_1 and x_2, and lastly to the result of $d(x)$, the desired response. We also printed the shape of our array (1000,4) to garantee it is working correctly.

b)

Train a MCCulloch-Pitts neuron on the training set by presenting all examples iteratively and, after each example x_i, updating the weights using an error-correction update rule:

$$w_{new} = w_{old} + \eta(d(x_i)) - y(x_i))x_i$$

where $\eta > 0$ is the learning rate and y(x) is the response of a McCulloch-Pitts neuron, as defined in exercise 1:

$$y(x) = sgn(w^T x)$$

In [80]:

```
def mcneuron(inputs, weights):
    y = np.sign(np.dot(weights.T, inputs))
    return y

def wnew(wold, eta, dx, yxi, xi ):
    return wold + eta*(dx-yxi)*xi
```

First, we started by defining the function for updating the weights and used the same function for the McCulloch-Pitts neuron as in exercice 1. Then, we defined the function epoch, which returns all the weights that resulted from presenting all examples of the training set once (an epoch):

In [81]:

```
def epoch(value_set, weight_vector, eta):
    output = []
    weights = []
    for x in range(0,value_set.shape[0]):
        output.append(mcneuron(value_set[x, 0:3],weight_vector))
        weights.append(weight_vector)
        weight_vector = wnew(weight_vector, eta, value_set[x,3],output[x], value_set

    output = np.array(output)
    weights = np.array(weights)
    return weights
```

We decided to start with the same weight vector as in exercise 1 (we could also have opted to create a random weight vector instead) and computed one training epoch. We opted for an eta learning rate of 0.01 (we discuss this choice a bit later). We printed the beginning and end of the resulting array of weight vectors to show that the weights were updated during the epoch:

In [82]:

```
w = np.array([3.,3.,2.])
w1 = epoch(training_set, w, 0.01)
print(w1)
```

```
[[3.         3.         2.        ]
 [3.         3.         2.        ]
 [3.         3.         2.        ]
 ...
 [1.72       3.02663082 2.86447054]
 [1.72       3.02663082 2.86447054]
 [1.72       3.02663082 2.86447054]]
```

To determine if weights have been changed over the epoch, we defined a stopping criterium function.

Because the arrays are composed of floating numbers, it might have taken a lot of iterations to stop if we had defined it as checking if both arrays (all weights before and after epoch) are equal. Therefore, instead we opted for subtracting the arrays and verifying if the resulting norm is below a certain number (we chose η=0.001).

We also printed the result for the stopping criterium w.r.t w1 and the original weight vector w (works by python broadcasting):

In [83]:

```
def stopping_criterium(a, b):
    if (LA.norm(np.array(a)-np.array(b))) < 0.001:
        return True
    else:
        return False
```

In [84]:

```
print(stopping_criterium(w1, w))
print(LA.norm(np.array(w1)-np.array(w)))
```

```
False
37.40188137746007
```

Because the stopping criterium is not fulfilled, we need to train the neuron over multiple epochs. We used a while loop that checked the stopping criterium and kept substituting wi and wi_minus1 in each iteration for it to work autonomously:

In [85]:

```
wi_minus1 = w1
wi = epoch(training_set, wi_minus1[-1,:], 0.01)
while stopping_criterium(wi, wi_minus1) is not True:
    wi_minus1 = wi
    wi = epoch(training_set, wi_minus1[-1,:], 0.01)


print("Norm of difference of weight arrays after final epoch:")
print(LA.norm(np.array(wi)-np.array(wi_minus1)))
print("Final weight (from final epoch):")
print(wi[-1,:])
```

```
Norm of difference of weight arrays after final epoch:
0.0
Final weight (from final epoch):
[1.5        3.01273183 2.997886  ]
```

PTK: You might want to insert a counter (and a related break condition) into the while loop, for the case that the stopping criterion is not reached at all.

For the learning rate eta, we opted for consistenty using 0.01, considering that it should always be 0 < eta << 1. The smaller this step size, the longer it takes to stop the weight update, but the less likely it to "skip" a local minimum of the error function (in this case just the difference between the desired response and the current response). We tested the epoch for a few values of eta (1, 0.1, 0.01, 0.001 - not shown) and thought that 0.01 was a good compromise (with 1, the resulting weights are very different to the weights determined with the other etas, which are all relatively similar to each other).

Another (perhaps better) possibility would have been to use an "adaptive step size", that is, increase eta it if the error is lower and diminish it if the error is higher. That way, the computation of the new weights is more efficient, but also avoids "skipping" the minimum of the error function.

c)

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

In [86]:

```
x2 = get_xvalues(1000)
d2_values = get_dvalues(x2)

validation_set = np.hstack((x2, d2_values))
print(validation_set[:5,:])
print(validation_set.shape)
```

```
[[-1.         -0.03689532 -0.29526786 -1.        ]
 [-1.         -2.40205862 -0.18523177 -1.        ]
 [-1.         -0.34697509 -0.63012558 -1.        ]
 [-1.          0.5647252   0.25953423  1.        ]
 [-1.          0.91589326  1.11392727  1.        ]]
(1000, 4)
```

In [87]:

```
w2 = epoch(validation_set, wi[-1, :], 0.01)
#print(w2)

print("Norm of difference array between weights in training set and validation set:"
print(LA.norm(np.array(w2)-np.array(wi)))

def error(value_set, weight_vector):
    error = []
    output = []
    for x in range(0,value_set.shape[0]):
        output.append(mcneuron(value_set[x, 0:3],weight_vector))
        error.append(value_set[x,3] - output[x])
    error = np.array(error)
    output = np.array(output)
    return error

e = error(validation_set, wi[-1, :])
percentage_correct = (1000 - np.count_nonzero(e))/1000 * 100
print("Percentage of examples with neuron response equal to the desired response:")
print(percentage_correct)
```

```
Norm of difference array between weights in training set and validatio
n set:
0.0
Percentage of examples with neuron response equal to the desired respo
nse:
100.0
```

The difference between the final weights from the training set and the ones used in one epoch of the validation set is close to 0. That means that the weights have not been updated significantly during the presentation of the examples in the validation set, and that, therefore, the weights are already suitable to perform the function.

More importantly, computing the error function (difference between the desired response and the output) for all examples in the validation set shows that, for most examples, the output is equal to the desired response. This can be easily observed by the percentage of correct classifications (very close to 100%). Thus, the neuron can indeed perform the trained comparison function.

d)

Plot the training set and label each input vector according to its response class. Superimpose the weight vector on the same plot(think about the bias term w_0). Explain in what sense the weight vector is optimal:

In [88]:

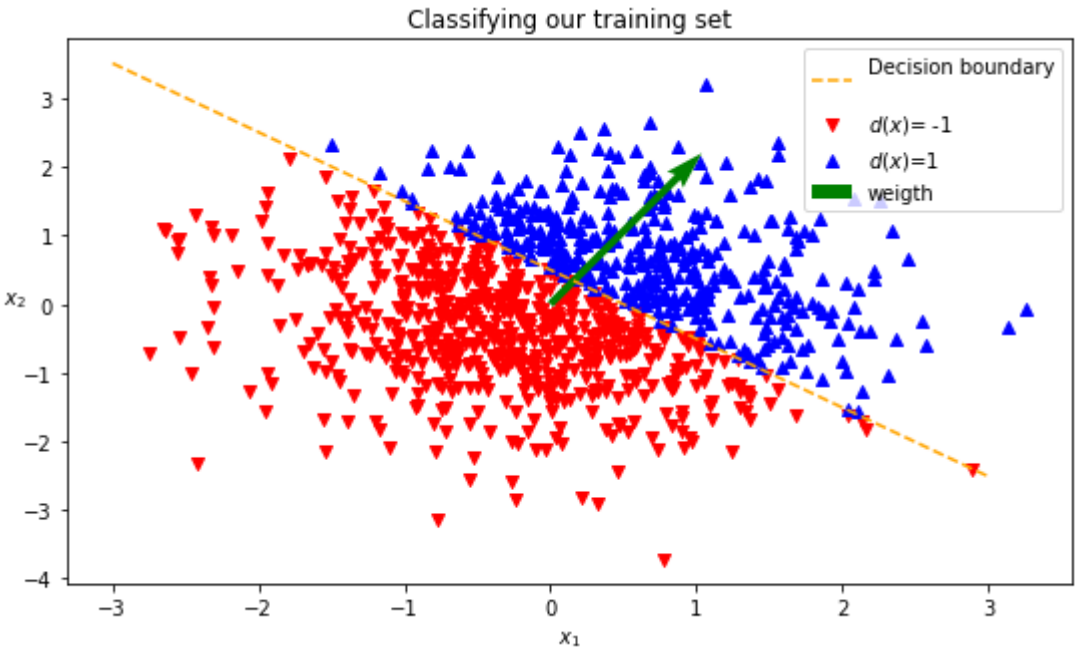
```
def decisionboun(x, weigths):
    y = -(weigths[-1,1]/weigths[-1,2])*x + (1.5/weigths[-1,2])
    return y
```

In [89]:

```
xplot = training_set[:,1:3]
yplot = training_set[:,-1]
xi = np.linspace(-3,3,1000)

plt.scatter(xplot[yplot==-1,0], xplot[yplot==-1,1], marker='v', c='red', label ='$d(
plt.scatter(xplot[yplot==1,0], xplot[yplot==1,1], marker='^', c='blue', label='$d(x)
plt.plot(xi, decisionboun(xi, wi), ls='--', c='orange', label='Decision boundary\n')
plt.quiver(0, 0 , wi[-1,1], wi[-1,2], color='green', label='weigth', scale = 20)  #w
legend = plt.legend()
plt.xlabel('$x_1$')  # putting the $ let u use latex inside, it only works in the ma
plt.ylabel('$x_2$', rotation='horizontal')
plt.title("Classifying our training set")
plt.show()
```

Out[89]:



The weigth vector is optimal because it is perpendicular to the decision boundary ($y = -\frac{w_1}{w2} * x_1 + \frac{\theta}{w2}$)

PTK: with

```
plt.axis('equal')
```
*you'll get a square format plot without the stretching in **x**-direction*

Also, the origing of the weigth vector (being orthogonal to the decision boundary) is not at (0,0) - please see the tutorial tonigth for the detailed interpretation.

4. Linear separability

a)

Plot the XOR classification problem on a Euclidian plane. Explain why the problem is not linearly separable.

1^{st} We define an array with the truth table of XOR

In [90]:

```
pq0=[0,0,0]
pq01=[0,1,1]
pq10=[1,0,1]
pq11=[1,1,0]
truthtableXOR=np.row_stack((pq0,pq01,pq10,pq11))
```

Now we proceed to plot 4 different boundaries where we can see the XOR logical operation can not be computed by a single linear operation.

In [91]:

```
pqval=truthtableXOR[:,:2]
pqsol=truthtableXOR[:,-1]
```

In [92]:

```
xi = np.linspace(0,1,50)
yplus=lambda x : x + 0.5
yminus=lambda x : x - 0.5
diagupbound=[yplus(a) for a in xi]
diaglowbound=[yminus(a) for a in xi]
```

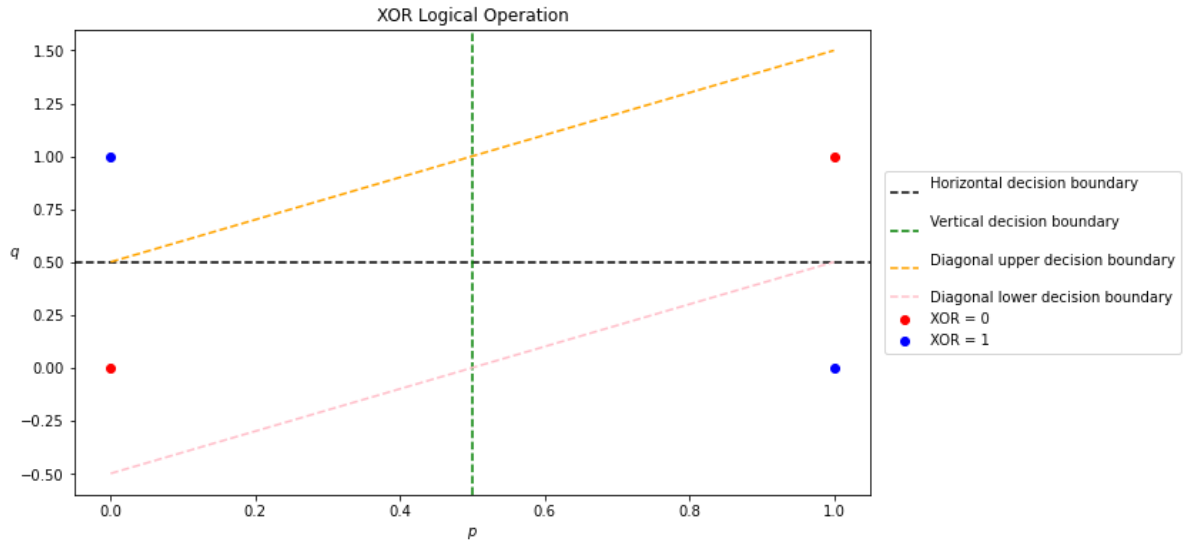
In [93]:

```
plt.figure(figsize=(10,6))
plt.scatter(pqval[pqsol==0,0],pqval[pqsol==0,1], c = 'r', label = "XOR = 0")
plt.scatter(pqval[pqsol==1,0],pqval[pqsol==1,1], c = 'b', label = "XOR = 1")
plt.axhline(y= 0.5, ls='--', c='k', label='Horizontal decision boundary\n')
plt.axvline(x= 0.5, ls='--', c='green', label='Vertical decision boundary\n')
plt.plot(xi, diagupbound, ls='--', c='orange', label='Diagonal upper decision bounda
plt.plot(xi,diaglowbound, ls='--', c='pink', label='Diagonal lower decision boundary
plt.legend(loc="center right", bbox_to_anchor=(1.4, 0.5), ncol=1)
plt.gca().set(title="XOR Logical Operation")
plt.xlabel('$p$')
plt.ylabel('$q$', rotation='horizontal')
plt.show
```

Out[93]:

```
<function matplotlib.pyplot.show(close=None, block=None)>
```

Out[93]:



As we can see, only one decision boundary can't compute the logical operation XOR [(x_1 AND not x_2) OR (not x_1 AND x_2)]. For that we need 2 neurons on the hidden layer (for both AND functions individually) and a final neuron (for the OR function between the two). Then, for example, these neurons could compute the XOR using the two diagonal boundaries shown in the plot.

b) (bonus)

In order to train this single perceptron on binary inputs, it is necessary to redefine the function to obtain the x values as well as define the desired output based on the truth table:

In [94]:

```
def get_xXORvalues(n):
    x = []
    for i in range(1,1001):
        xi1 = np.random.randint(0,2)
        xi2 = np.random.randint(0,2)
        xi = np.array([-1, xi1, xi2])
        x.append(xi)
    return np.array(x)

x3 = get_xXORvalues(1000)
```

In [95]:

```
def d_XOR(x):
    y = []
    if x[1] == 1 and x[2] == 0:
        y.append(1)
    elif x[1] == 0 and x[2] == 1:
        y.append(1)
    else:
        y.append(0)
    return np.array(y)
```

In [96]:

```
def get_dXORvalues(x):
    d_XORvalues = []
    for i in x:
        d_XORvalues.append(d_XOR(i))
    return np.array(d_XORvalues)

d_XORvalues = get_dXORvalues(x3)
```

In [97]:

```
XORtraining_set = np.hstack((x3, d_XORvalues))
print(XORtraining_set[:5,:])
print(XORtraining_set.shape)
```

```
[[-1  0  0  0]
 [-1  1  1  0]
 [-1  0  1  1]
 [-1  1  1  0]
 [-1  1  0  1]]
(1000, 4)
```

Then, taking the new training set and computing the epochs in a way similar to before, we printed some of the weights from the last epoch:

In [98]:

```
w = np.array([3.,3.,2.])
w3 = epoch(XORtraining_set, w, 0.01)
#print(w3)
```

In [101]:

```
wiXOR_minus1 = w3
wiXOR = epoch(XORtraining_set, wiXOR_minus1[-1,:], 0.01)
while stopping_criterium(wiXOR, wiXOR_minus1) is not True:
    wiXOR_minus1 = wiXOR
    wiXOR = epoch(XORtraining_set, wiXOR_minus1[-1,:], 0.01)
    print(LA.norm(np.array(wiXOR_minus1)-np.array(wiXOR)))


print("Final weights (from final epoch):")
print(wiXOR[-8:])

e3 = error(XORtraining_set, wiXOR[-1, :])
percentage_correct = (1000 - np.count_nonzero(e3))/1000 * 100
print("Percentage of examples with neuron response equal to the desired response:")
print(percentage_correct)
```

```
26.05099614218245
0.0
Final weights (from final epoch):
[[-6.26929064e-15  2.13613849e-14  2.00000000e-02]
 [ 1.00000000e-02 -1.00000000e-02  1.00000000e-02]
 [-1.00000000e-02  1.00000000e-02  1.00000000e-02]
 [-6.26929064e-15  2.13613849e-14  2.01401396e-14]
 [-6.26929064e-15  2.13613849e-14  2.01401396e-14]
 [ 1.00000000e-02 -1.00000000e-02 -1.00000000e-02]
 [-1.00000000e-02 -1.00000000e-02  1.00000000e-02]
 [-6.26929064e-15 -2.00000000e-02  2.01401396e-14]]
Percentage of examples with neuron response equal to the desired respo
nse:
26.6
```

-> It becomes clear that the weight vectors keep changing between the examples and do not settle on fixed values, that is, the learning algorithm does not converge. Taking, for example, the weight vector from the final example and calculating the percentage of correct responses, one can tell that it is very far away from 100%.

PTK: very good solution overall, with a lot of discussion, making it very easy to follow your train of thought! 100% for the CP #1

In [ ]:

```
```