Sheet_1_Aleksandar

October 31, 2024

0.1 MNS Computer Practical 1

"No AI tool has been used to produce this solution."

[1]:
```
#importing the libraries and setting the random number generator
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
rng = np.random.default_rng(seed=1023)
```

0.1.1 1. McCulloch-Pits Neurons

a)

[2]:
```
#function to calculate the output of McCulloch-Pits neuron
def mc_culloch_pits_neuron(w, x):
    x1 = np.hstack((-1, x))
    return int(np.sign(w@x1))
```

[3]:
```
#check
w = np.array([-1, 1, 0, -1])
x = np.array([1, 2, 3])
mc_culloch_pits_neuron(w, x)
```

[3]:
```
-1
```

b)

[4]:
```
w = np.array([3, 2, 2])
```

For $x_1 = x_2 = -1$ we get:

[5]:
```
x = np.array([-1, -1])
mc_culloch_pits_neuron(w, x)
```

[5]:
```
-1
```

For $x_1 = -1$ and $x_2 = 1$ we get:

[6]:
```
x = np.array([-1, 1])
mc_culloch_pits_neuron(w, x)
```

[6]:
```
-1
```

For $x_1 = 1$ and $x_2 = -1$ we get:

[7]:
```
x = np.array([1, -1])
mc_culloch_pits_neuron(w, x)
```

[7]:
```
-1
```

For $x_1 = x_2 = 1$ we get:

[8]:
```
x = np.array([1, 1])
mc_culloch_pits_neuron(w, x)
```

[8]:
```
1
```

We have

$$
\begin{aligned}
(-1, -1) &\to -1 \\
(1, -1) &\to -1 \\
(-1, 1) &\to -1 \\
(1, 1) &\to 1
\end{aligned}
$$

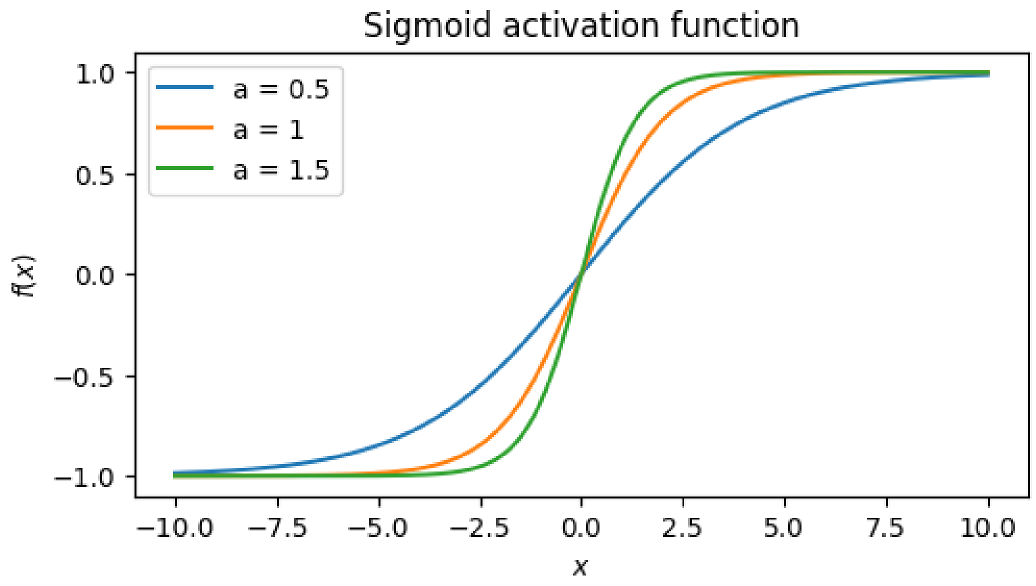
which is a table for AND operation.

0.1.2 2. Activation functions

a) Sigmoid function

[9]:
```
def plot_sigmoid(a1, a2, a3):

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

    x = np.linspace(-10, 10, 1000)
    y1 = sigmoid(a1, x)
    y2 = sigmoid(a2, x)
    y3 = sigmoid(a3, x)

    fig = plt.figure(figsize=(6, 3))

    plt.plot(x, y1)
    plt.plot(x, y2)
    plt.plot(x, y3)

    plt.xlabel("$x$")
    plt.ylabel("$f(x)$")

    plt.legend([f'a = {a1}', f'a = {a2}', f'a = {a3}'])
    plt.title("Sigmoid activation function")
```

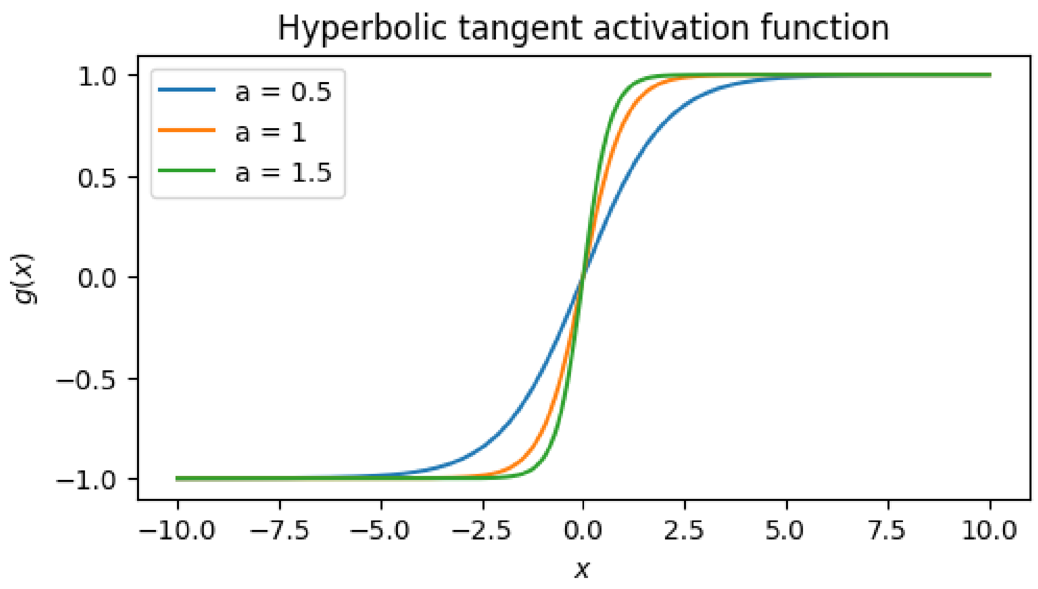
[10]:
```
plot_sigmoid(.5, 1, 1.5)
```



b) Hyperbolic tangent function

[11]:
```
def plot_tanh(a1, a2, a3):

    x = np.linspace(-10, 10, 1000)
    y1 = np.tanh(a1*x)
    y2 = np.tanh(a2*x)
    y3 = np.tanh(a3*x)

    fig = plt.figure(figsize=(6, 3))

    plt.plot(x, y1)
    plt.plot(x, y2)
    plt.plot(x, y3)

    plt.xlabel("$x$")
    plt.ylabel("$g(x)$")

    plt.legend([f'a = {a1}', f'a = {a2}', f'a = {a3}'])
    plt.title("Hyperbolic tangent activation function")
```

[12]:
```
plot_tanh(.5, 1, 1.5)
```

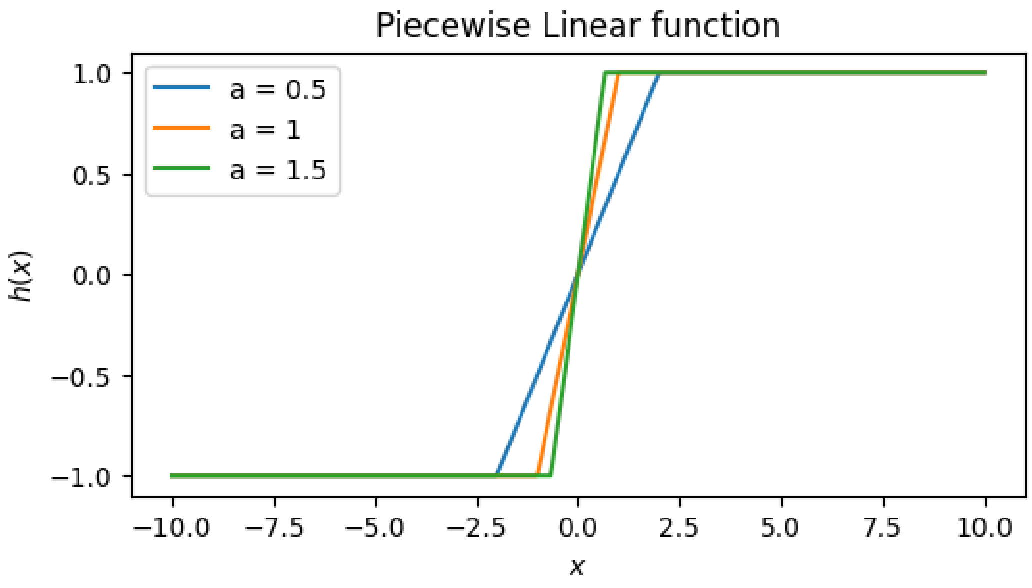


0.1.3 c) Piecewise linear function

[13]:
```
def piecewise_linear(a, x):
    if x <= -1/a:
        return -1
    elif -1/a < x < 1/a:
        return a*x
    else:
        return 1
```

[14]:
```
def plot_piecewise_linear(a1, a2, a3):

    def piecewise_linear(a, x):
        if x <= -1/a:
            return -1
        elif -1/a < x < 1/a:
            return a*x
        else:
            return 1

    x = np.linspace(-10, 10, 1000)
    y1 = [piecewise_linear(a1, xi) for xi in x]
    y2 = [piecewise_linear(a2, xi) for xi in x]
    y3 = [piecewise_linear(a3, xi) for xi in x]
    fig = plt.figure(figsize=(6, 3))

    plt.plot(x, y1)
    plt.plot(x, y2)
    plt.plot(x, y3)

    plt.xlabel("$x$")
    plt.ylabel("$h(x)$")

    plt.legend([f'a = {a1}', f'a = {a2}', f'a = {a3}'])
    plt.title("Piecewise Linear function")
```

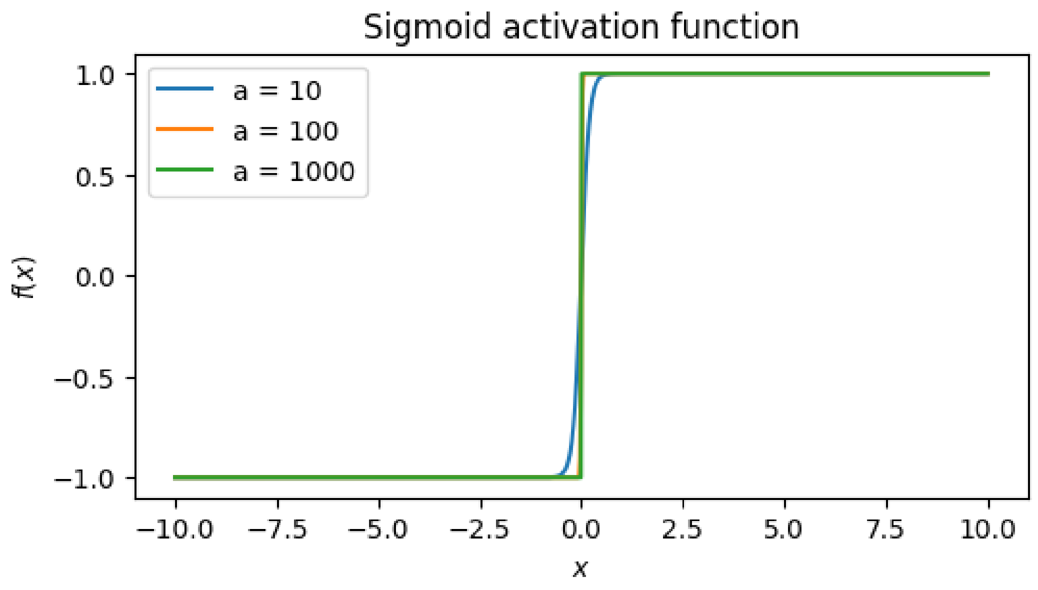
[15]:
```
plot_piecewise_linear(.5, 1, 1.5)
```



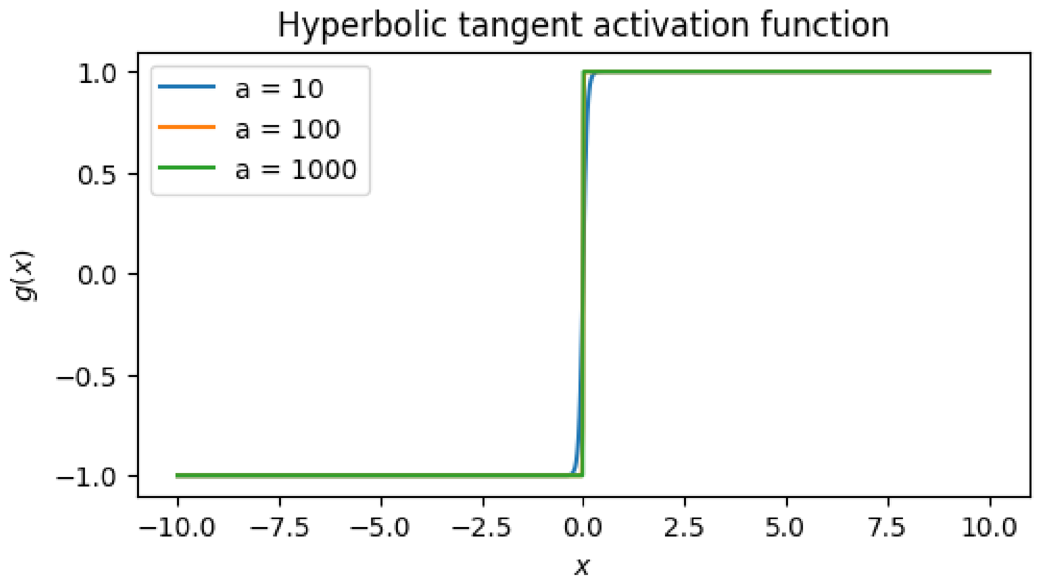
We just need to set the parameter a big enough. For example:

[16]:
```
plot_sigmoid(10, 100, 1000)
```

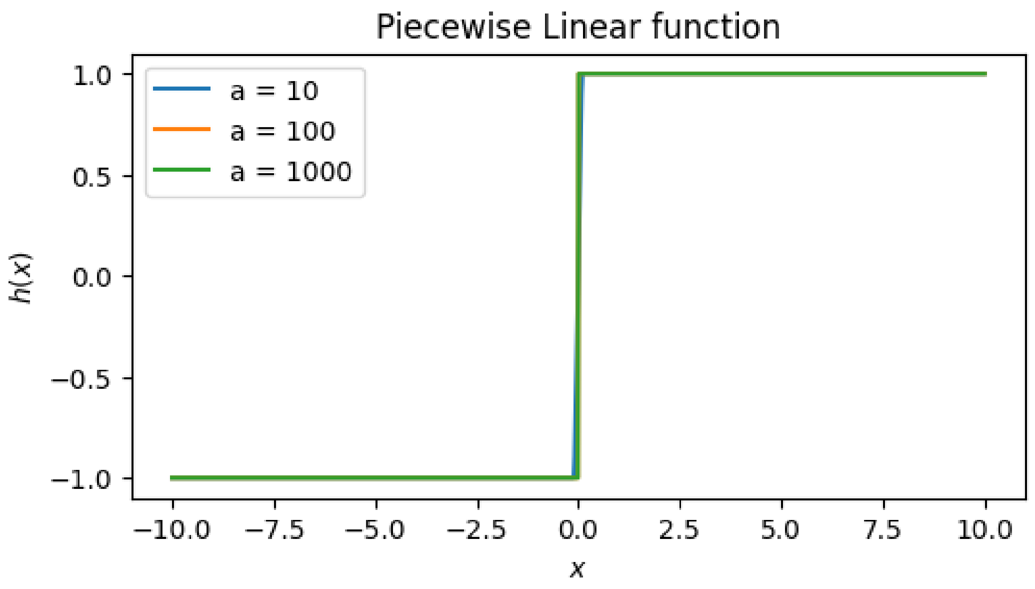
```
C:\Users\alegza\AppData\Local\Temp\ipykernel_9784\2794667910.py:4:
RuntimeWarning: overflow encountered in exp
  return 2/(1 + np.exp(-a*x)) - 1
```



[17]:
```
plot_tanh(10, 100, 1000)
```



[18]:
```
plot_piecewise_linear(10, 100, 1000)
```



0.1.4 3. Rosenblatt's Perceptron

a)

```
[19]: #defining the training set
N = 1000
X_train = rng.standard_normal((N, 2))
X_train = np.hstack((-np.ones((N, 1)), X_train))
X_train[:10]
```

```
[19]: array([[-1.        ,  1.59777721,  0.44287552],
       [-1.        ,  0.40821543, -0.96506678],
       [-1.        , -0.88602812, -0.44414343],
       [-1.        , -1.54370735, -0.15083739],
       [-1.        , -1.19275067,  0.2686129 ],
       [-1.        ,  1.79503243,  0.47229044],
       [-1.        ,  0.6565819 , -0.45283222],
       [-1.        ,  1.07657657, -0.39174439],
       [-1.        , -0.62435265, -0.43551258],
       [-1.        , -0.42197685, -0.11799472]])
```

```
[20]: X_train.shape
```

```
[20]: (1000, 3)
```

```
[21]: #defining the target vector for the training set
d_train = np.array([1 if x[2]-x[1] >= .5 else -1 for x in X_train])
d_train = d_train.reshape(1000, 1)
```

```
d_train[:10]
```

[21]:
```
array([[-1],
       [-1],
       [-1],
       [ 1],
       [ 1],
       [-1],
       [-1],
       [-1],
       [-1],
       [-1]])
```

b)

[22]:
```
#initialize the weight vector
w = np.zeros((1, 3))

#set the learning rate
lr = .1
epoch = 0

#training the neuron
while epoch < 10**5:
    epoch += 1
    w_old = w.copy()

    #updating the weights
    for k, x in enumerate(X_train):
        w += lr*(d_train[k] - np.sign(w@x))*x

    #comparing if the weights changed from the previous epoch
    if (w == w_old).all():
        print(f'The training finished in {epoch} epochs.')
        print(f'The final weight vector: {w}')
        break

else:
    print(f'The algorithm did not converge after {epoch} epochs.')
```

```
The training finished in 9 epochs.
The final weight vector: [[ 1.1        -2.19504749  2.17278969]]
```

[23]:
```
#checking how our model performs on the training set
acc = (d_train ==  np.sign(X_train@w.T)).sum()/N

print(f'The model performs with {acc*100}% accuracy on the training set.')
```

```
The model performs with 100.0% accuracy on the training set.
```

We decided to use small (but not too small!) value for η, so that the values of the final weight vector $\mathbf{w}$ don't blow-up.

c)

```
[24]: #defining the validation set
      X_valid = rng.standard_normal((N, 2))
      X_valid = np.hstack((-np.ones((N, 1)), X_valid))

      #defining  the target vector for the validation set
      d_valid = np.array([1 if x[2]-x[1] >= .5 else -1 for x in X_valid])
      d_valid = d_valid.reshape(1000, 1)
```

```
[25]: #checking how our model performs on the validation set
      acc = (d_valid ==  np.sign(X_valid@w.T)).sum()/N

      print(f'The model performs with {acc*100}% accuracy on the validation set.')
```

```
The model performs with 99.9% accuracy on the validation set.
```

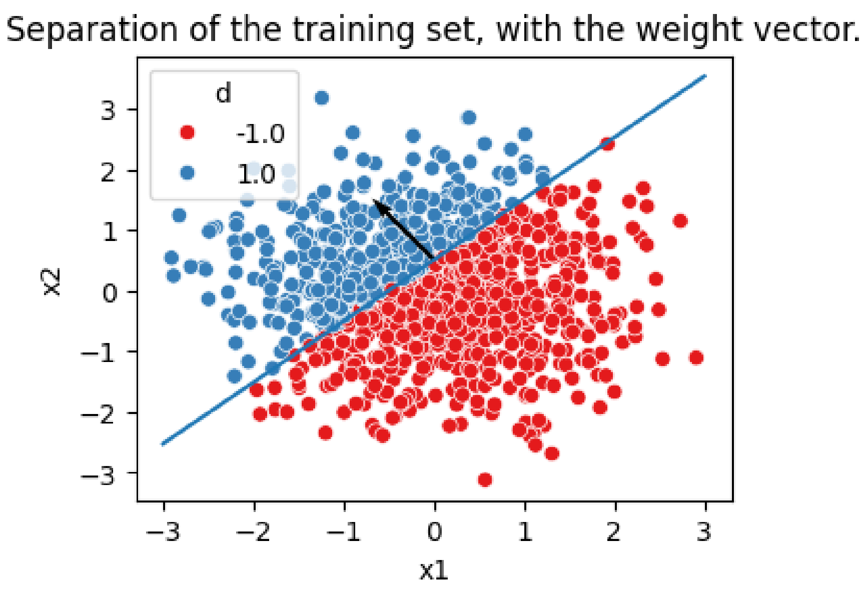
d)

```
[26]: #making dataframe out of the training set, to make plotting easier
      train_set = np.hstack((X_train[:, 1:], d_train))
      df_train_set = pd.DataFrame(train_set, columns=['x1', 'x2', 'd'])
      df_train_set.head()
```

```
[26]:          x1        x2    d
      0  1.597777  0.442876 -1.0
      1  0.408215 -0.965067 -1.0
      2 -0.886028 -0.444143 -1.0
      3 -1.543707 -0.150837  1.0
      4 -1.192751  0.268613  1.0
```

```
[27]: w = w.ravel()

      #the decision boundary equation
      x1 = np.linspace(-3, 3, 1000)
      x2 = -x1*(w[1]/w[2]) + (w[0]/w[2])

      fig = plt.figure(figsize=(4, 3))

      #the training set
      sns.scatterplot(data=df_train_set, x='x1', y='x2', hue='d', palette='Set1');

      #the decision boundary
      plt.plot(x1, x2)

      #the weight vector
```

```
plt.quiver(x1[500], x2[500], w[1], w[2], scale=21)

plt.title('Separation of the training set, with the weight vector.');
```



Decision boundary is a line orthogonal to vector (w_1, w_2) and given by the equation

$$x_1 w_1 + x_2 w_2 - w_0 = 0.$$

The weight vector is optimal in a sence that determines the decision boundary which perfectly separates two classes.

0.1.5 4. Linear Separability

a)

[28]:
```
#defining the training set
N = 1000
X_train = rng.standard_normal((N, 2))
X_train = np.hstack((-np.ones((N, 1)), X_train))
X_train[:10]
```

[28]:
```
array([[-1.        , -0.90286305, -2.17742402],
       [-1.        ,  1.88914445, -0.14877927],
       [-1.        ,  0.4073816 , -0.4328634 ],
       [-1.        , -0.27149695, -0.84523149],
       [-1.        , -1.12027308, -0.67852874],
       [-1.        ,  0.56469782, -1.40579299],
       [-1.        ,  0.37134317,  0.27071827],
       [-1.        , -0.47015459, -1.95674652],
       [-1.        ,  1.14711798, -1.03218562],
       [-1.        , -0.01734606, -2.15867101]])
```

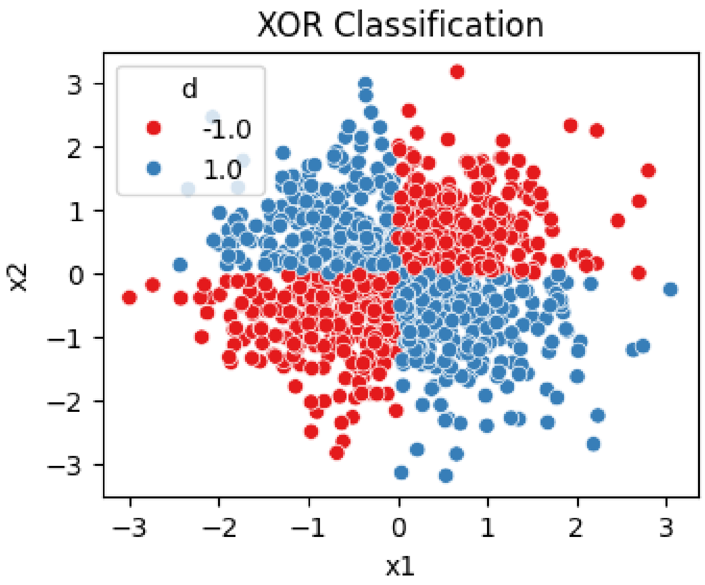
[29]:
```
#defining the target vector for the training set
y_train = -np.sign(X_train[:, 1])*np.sign(X_train[:, 2])
y_train = y_train.reshape(1000, 1)
y_train[:10]
```

[29]:
```
array([[-1.],
       [ 1.],
       [ 1.],
       [-1.],
       [-1.],
       [ 1.],
       [-1.],
       [-1.],
       [ 1.],
       [-1.]])
```

[30]:
```
#making dataframe out of the training set, to make plotting easier
train_set = np.hstack((X_train[:, 1:], y_train))
df_train_set = pd.DataFrame(train_set, columns=['x1', 'x2', 'd'])
df_train_set.head()
```

[30]:
```
         x1        x2    d
0 -0.902863 -2.177424 -1.0
1  1.889144 -0.148779  1.0
2  0.407382 -0.432863  1.0
3 -0.271497 -0.845231 -1.0
4 -1.120273 -0.678529 -1.0
```

[31]:
```
#plotting the training set
fig = plt.figure(figsize=(4, 3))
sns.scatterplot(data=df_train_set, x='x1', y='x2', hue='d', palette='Set1')
plt.title("XOR Classification");
```



b)

[32]:
```python
#atteming the train the neuron
w = np.zeros((1, 3))
lr = .1
epoch = 0
while epoch < 10**2:
    epoch += 1
    w_old = w.copy()
    print(w)
    for k, x in enumerate(X_train):
        w += lr*(y_train[k] - np.sign(w@x))*x
    if (w == w_old).all():
        break
        print(f'The training finished in {epoch} epochs.')
        print(f'The final weight vector: {w}')

else:
    print(f'The algorithm did not converge after {epoch} epochs.')
```

```
[[0. 0. 0.]]
[[-0.1        -0.36113428  0.37871278]]
[[-0.1        -0.32059642  0.02643883]]
[[-0.3        -0.10912557  0.23269366]]
[[-0.1        -0.32641435  0.02778306]]
[[-0.1        -0.26291053  0.12922434]]
[[-0.3        -0.10830084  0.2369243 ]]
[[-0.1        -0.27730443  0.08081046]]
```

```
[[-0.1         -0.2707802   0.09722561]]
[[-0.1         -0.3591578   0.37765732]]
[[-0.3         -0.19255613  0.22430578]]
[[-0.1         -0.37881031  0.11250769]]
[[-0.1         -0.40505542  0.10849025]]
[[-0.1         -0.27152968  0.06973953]]
[[-0.1         -0.37891263  0.09370029]]
[[-0.1         -0.42679278  0.07195341]]
[[-0.1         -0.41143162  0.13555927]]
[[-0.1         -0.3283267   0.08297041]]
[[-0.1         -0.2665218   0.09577366]]
[[-0.1         -0.32073765  0.02349497]]
[[-0.3         -0.21346791  0.21570151]]
[[-0.3         -0.16664429  0.24799538]]
[[-0.3         -0.11793247  0.2351103 ]]
[[-0.1         -0.39797483  0.08591889]]
[[-0.1         -0.42934833  0.37009947]]
[[-0.1         -0.32847798  0.15368426]]
[[-0.1         -0.40919704  0.10744458]]
[[-0.1         -0.40994897  0.13431086]]
[[-0.1         -0.24282795  0.05853285]]
[[-0.3         -0.09436965  0.23327954]]
[[-0.1         -0.31179774  0.00856691]]
[[-0.1         -0.41825659  0.3924228 ]]
[[-0.1         -0.43108217  0.06831852]]
[[-0.3        -0.1078663  0.2319434]]
[[-0.1         -0.25079087  0.08124438]]
[[ 0.1         -0.19301714  0.215316  ]]
[[-0.1         -0.23284219  0.06799358]]
[[-0.1         -0.31011329  0.11916136]]
[[-0.1         -0.26247602  0.12565463]]
[[-0.1         -0.21826124  0.1276359 ]]
[[-0.1         -0.34084565  0.15327437]]
[[-0.1         -0.39116963  0.10335167]]
[[-0.1         -0.44995821  0.06576283]]
[[-0.1         -0.30187415  0.07950521]]
[[-0.1         -0.31353961  0.12471335]]
[[-0.1         -0.31847346  0.01065871]]
[[-0.3         -0.21124006  0.20635932]]
[[-0.1         -0.2359668   0.10986781]]
[[ 0.1         -0.15803493  0.26022472]]
[[-0.1         -0.4043879   0.13625019]]
[[-0.1         -0.34794285  0.40675296]]
[[-0.1         -0.26474823  0.09474735]]
[[-0.1         -0.26104255  0.12936143]]
[[-0.1         -0.42999523  0.11614501]]
[[-0.1         -0.27195394  0.10936891]]
[[-0.1         -0.42863772  0.07267583]]
```

```
[[-0.1        -0.2298624   0.12500961]]
[[-0.1        -0.30990222  0.12645433]]
[[-0.1        -0.30336352  0.1322313 ]]
[[-0.1        -0.43398803  0.07238734]]
[[-0.1        -0.42607337  0.07150928]]
[[-0.1        -0.41071222  0.13511514]]
[[-0.1        -0.33661527  0.13340455]]
[[-0.1        -0.36124165  0.1184222 ]]
[[-0.1        -0.31346365  0.14065608]]
[[-0.1        -0.3618186   0.37530394]]
[[-0.3        -0.10789233  0.23155631]]
[[-0.1        -0.25081691  0.08085729]]
[[ 0.1        -0.19304318  0.21492891]]
[[-0.1        -0.23286823  0.06760649]]
[[-0.1        -0.31013933  0.11877427]]
[[-0.1        -0.26250206  0.12526754]]
[[-0.1        -0.21828728  0.12724881]]
[[-0.1        -0.34087169  0.15288728]]
[[-0.1        -0.23131064  0.121572  ]]
[[-0.1        -0.40251719  0.08400141]]
[[-0.1        -0.38210233  0.09629993]]
[[-0.1        -0.42998247  0.07455305]]
[[-0.1        -0.23120715  0.12688683]]
[[-0.1        -0.31304647  0.16806475]]
[[-0.1        -0.27508319  0.07962135]]
[[-0.1        -0.41551868  0.39507941]]
[[-0.1        -0.3214123   0.08387335]]
[[-0.1        -0.25997032  0.13060863]]
[[-0.1        -0.42394604  0.34334628]]
[[-0.1        -0.2752342   0.08081803]]
[[-0.1        -0.41554543  0.11224003]]
[[-0.1        -0.43096017  0.36813424]]
[[-0.3        -0.20502645  0.21905246]]
[[-0.3        -0.10869068  0.23163465]]
[[ 0.1        -0.17254008  0.23108127]]
[[ 0.1        -0.15505085  0.25838898]]
[[-0.1        -0.42117013  0.11794117]]
[[-0.1        -0.45115228  0.0692527 ]]
[[-0.1        -0.24582367  0.05860267]]
[[-0.1        -0.26879095  0.10923657]]
[[-0.1        -0.2683512   0.07583324]]
[[-0.1        -0.28493171  0.0665633 ]]
[[-0.1        -0.30834814  0.11995113]]
[[-0.1        -0.26071087  0.1264444 ]]
The algorithm did not converge after 100 epochs.
```

For the sake of demonstration we limited the algorithm to 100 epochs, but it wouldn't have converged for any preset number of epochs. We see that the entries of the weight vector keep changing signs with almost every iteration.

d) (speculation) Maybe 14 of 16 operations could be learn by perceptron, since they form (piecewise) linearly separable data, i.e. could be transformed to linearly separate data. Only two operations that do not form linearly separable data are XOR and equivalence (XNOR).

```
[ ]:
```