Machine Learning 2023/2024 (2^{nd} semester)

Master in Electrical and Computer Engineering

Department of Electrical and Computer Engineering

A. Pedro Aguiar (pedro.aguiar@fe.up.pt), **Aníbal Matos** (anibal@fe.up.pt), **Andry Pinto** (amgp@fe.up.pt), **Daniel Campos** (dfcampos@fe.up.pt), **Maria Inês Pereira** (maria.ines@fe.up.pt)

FEUP, Mar. 2024

Project #02

Note: This work is to be done in group of **2** elements. Use this notebook to answer all the questions. At the end of the work, you should **upload** the **notebook** and a **pdf file** with a printout of the notebook with all the results in the **moodle** platform. To generate the pdf file we have first to covert the notebook to html using the command `!jupyter nbconvert --to html "ML_project2.ipynb"`, then open the html file and printout to PDF.

Deadlines: Present you work (and answer questions) on the week of **May 20** in your corresponding practical class. Upload the files until 23:59 of **May 31, 2024**.

Identification

- **Group:** A06_B
- **Name:** Bruno Filipe Torres Costa
- **Student Number:** 202004966
- **Name:** André Silva Martins
- **Student Number:** 202006053

Initial setup: To download the data files, run the next cell.

In [ ]:
```
#!wget -O data-setMLproject2.zip https://www.dropbox.com/s/hnyhgqlj5lcqyq
#!unzip data-setMLproject2.zip -d.
```

Main goal

Consider the following scenario: A mobile robot aims to build a map of the environment with **semantics**, meaning that the robot should be capable to classify the objects nearby. The robot is travelling around and carries on-board a 2D LIDAR measurement device that obtains range measurements at each sample time $t = 0, 0.1, 0.2, \ldots$ The following cell shows an example of the type of data:

In [ ]:
```
import pandas as pd
df_test2obs = pd.read_csv('data_test2obs.csv', index_col=0)
df_test2obs.head(5)
#df

# By convention, zero values mean no range measurements.
# The units are:
# [m] for px and py (position of the robot)
# [m] for the LIDAR ranges
```

Out[ ]:

	px	py	angle -179	angle -178	angle -177	angle -176	angle -175	angle -174	angle -173	angle -172	...	angle 171	angl 17
0	-4.00	-2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0
1	-3.98	-2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0
2	-3.96	-2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0
3	-3.94	-2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0
4	-3.92	-2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0

5 rows × 362 columns

Note that the LIDAR measurements consist of range (distance) from the robot to a possible obstacle for each degree of direction, that is,

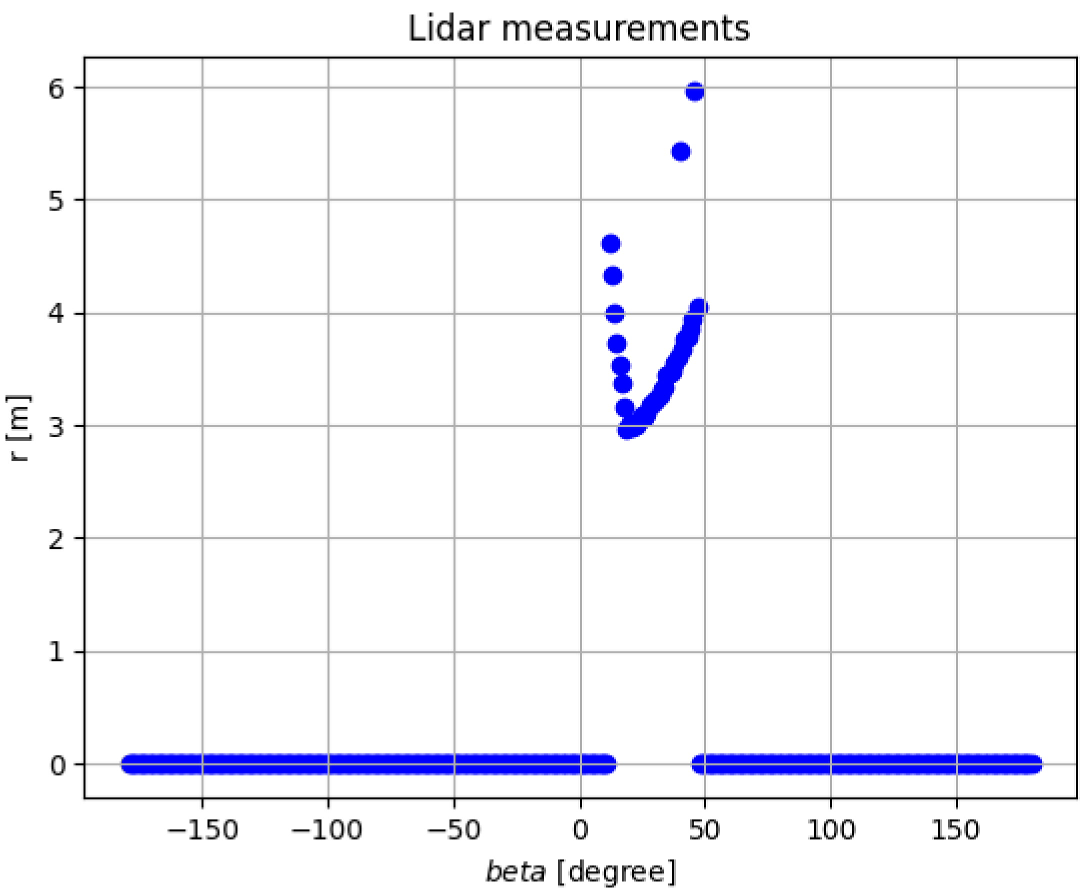
$$r_t = \{r_\beta + \eta_r : \beta = -179^o, -178^o, \ldots, 0^o, \ldots, 180^o\}$$

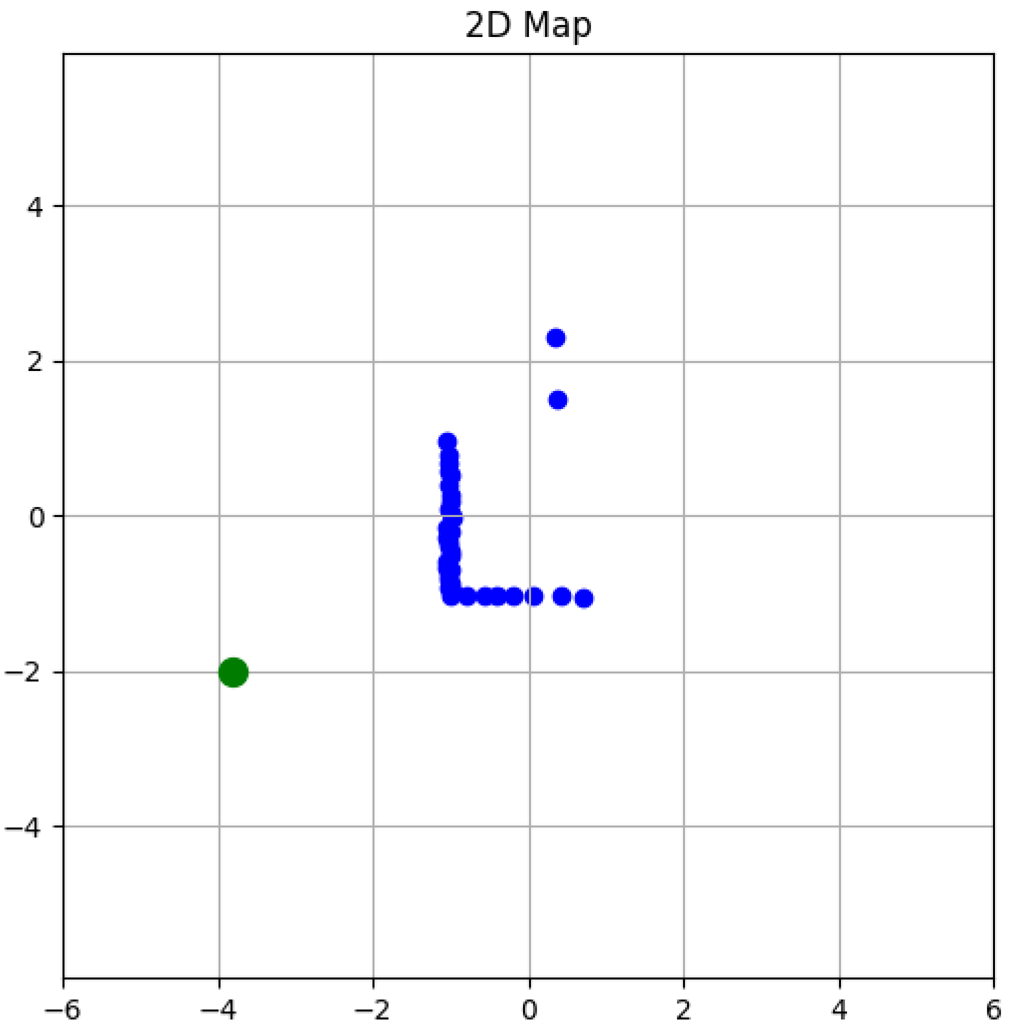
where η_r is assumed to be Gaussian noise. If there is no obstacle within the direction of the laser range or if it is far away, that is, if the distance is greater than $5\,m$, by convention the range measurement is set to zero. Moreover, with a small probability, the range measurements could be corrupted with *outliers*.

The next figure shows r_t as a function of the angle β taken at time $t = 1.0\,s$.

In [ ]:
```
import numpy as np
from numpy import *
import matplotlib.pyplot as plt

Lidar_range = df_test2obs.iloc[:, np.arange(2,362,1)].values
px = df_test2obs["px"].values
```

```
py = df_test2obs["py"].values

t=1*10 #1sec times number of samples/second
angle = np.linspace(-179, 180, num=360)

plt.figure()
plt.scatter(angle, Lidar_range[t], color='b')
plt.title('Lidar measurements')
plt.ylabel('r [m]')
plt.xlabel('$beta$ [degree]')
plt.grid()
plt.show()

#Build the cloud points in 2D plan
x_o, y_o = [], []
for i in range(len(Lidar_range[t])):
  if Lidar_range[t][i] > 0:
    x_o.append(px[t]+Lidar_range[t][i]*np.cos(angle[i]/180*np.pi))
    y_o.append(py[t]+Lidar_range[t][i]*np.sin(angle[i]/180*np.pi))

fig, ax = plt.subplots(figsize=(6,6))
ax.axis('equal')
xdim, ydim = 5, 5
plt.xlim(-xdim-1,xdim+1)
plt.ylim(-ydim-1,ydim+1)
plt.plot(px[t], py[t], 'g.', ms=20)  #position of the robot
plt.grid()

plt.scatter(x_o, y_o, color='b')
plt.title('2D Map')
plt.show()
```

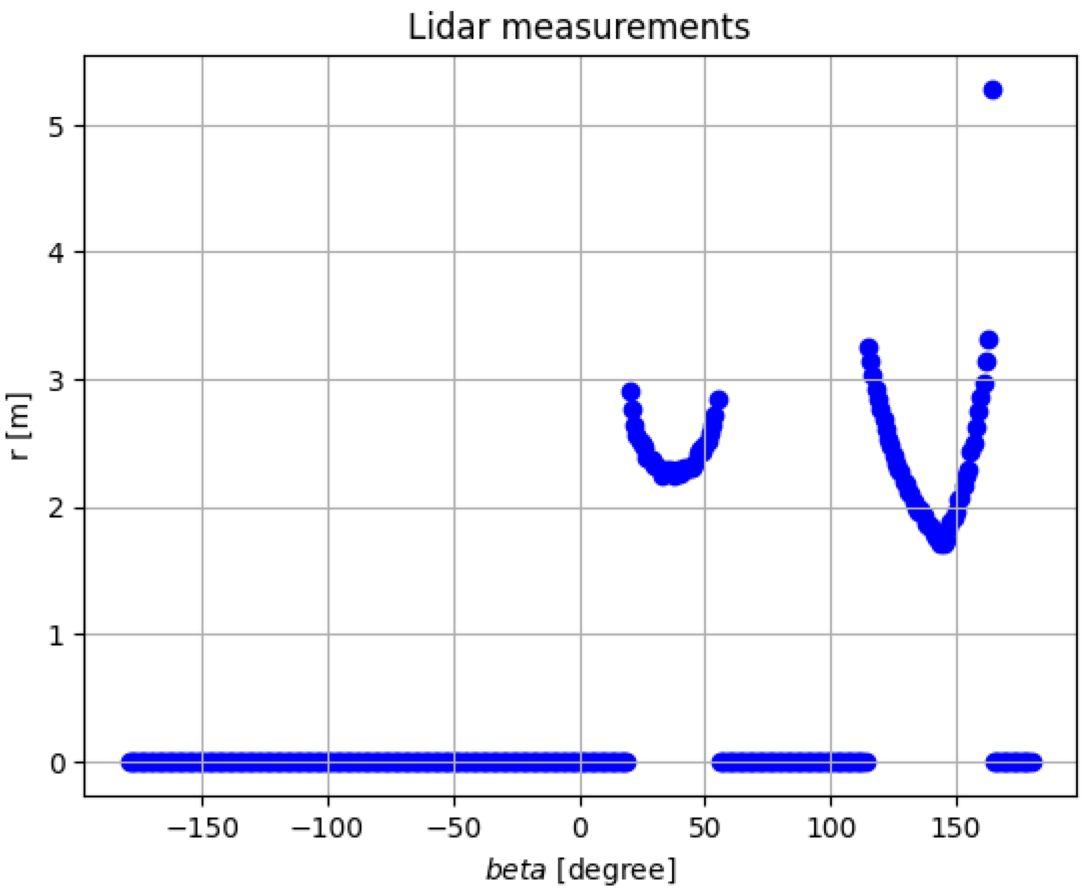


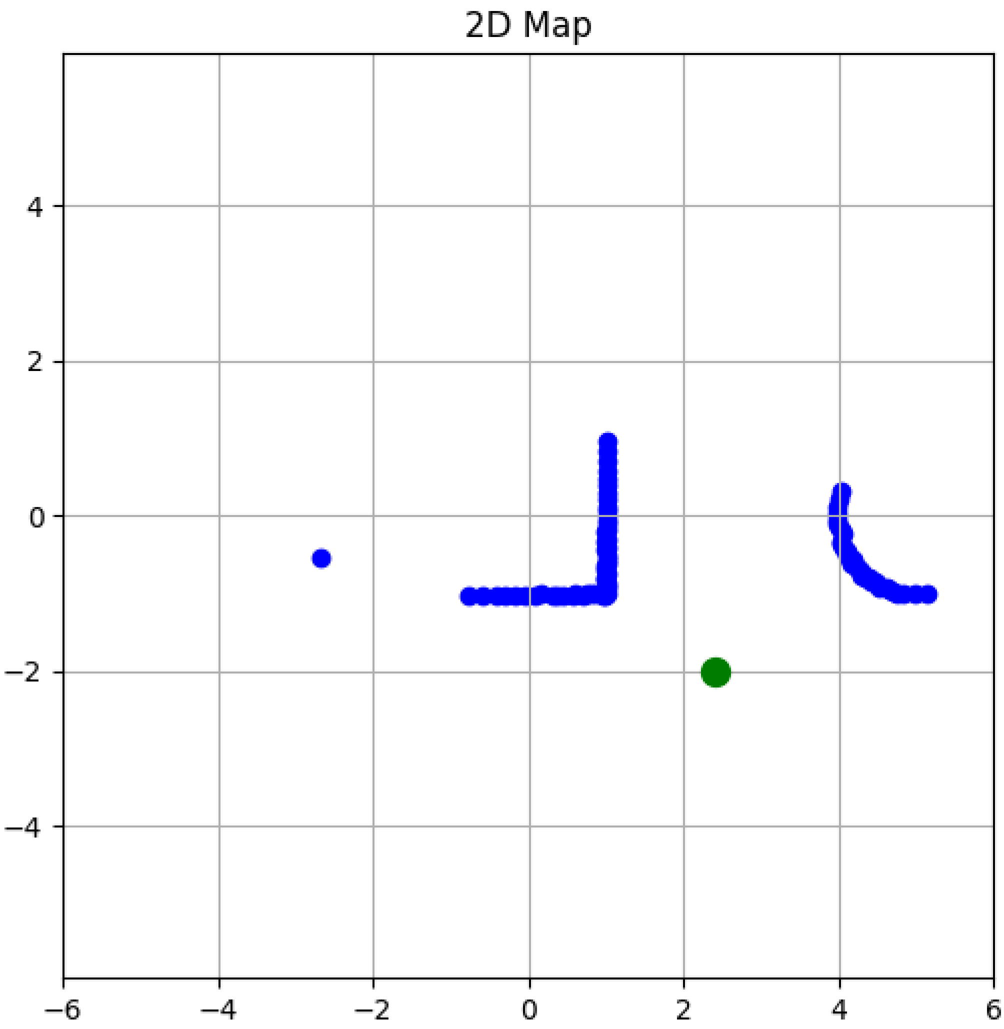


Note that it may be possible to have more than one object in the range of the LIDAR. Here goes an example when $t = 32\,s$:

In [ ]:
```
t=32*10 #5sec times number of samples/second
angle = np.linspace(-179, 180, num=360)

plt.figure()
plt.scatter(angle, Lidar_range[t], color='b')
plt.title('Lidar measurements')
plt.ylabel('r [m]')
plt.xlabel('$beta$ [degree]')
plt.grid()
plt.show()

#Build the cloud points in 2D plan
x_o, y_o = [], []
for i in range(len(Lidar_range[t])):
  if Lidar_range[t][i] > 0:
    x_o.append(px[t]+Lidar_range[t][i]*np.cos(angle[i]/180*np.pi))
    y_o.append(py[t]+Lidar_range[t][i]*np.sin(angle[i]/180*np.pi))

fig, ax = plt.subplots(figsize=(6,6))
ax.axis('equal')
xdim, ydim = 5, 5
plt.xlim(-xdim-1,xdim+1)
plt.ylim(-ydim-1,ydim+1)
plt.plot(px[t], py[t], 'g.', ms=20)  #position of the robot
```

```
plt.grid()

plt.scatter(x_o, y_o, color='b')
plt.title('2D Map')
plt.show()
```





Part 1: Classification of one object

At this point, the goal is to classify only one object that could be a square or a circle at each LIDAR snapshot.

To this end, it was performed a set of 4 experiments for each obstacle (alone) where in each experiment the robot travelled during $40\ s$ with a constant speed and constant direction (horizontal line segment from left to right) from the initial position $(p_x, p_y) = (-4, \bar{y})$ to the final position $(p_x, p_y) = (4, \bar{y})$, where $\bar{y} = -4, -3, -2, -1\ m$ The obstacle (circle and square) were placed at the center of the origin $(0, 0)$.

All the experiments were combined in a unique dataset and then randomly split into two datasets: the trainning data set (70%) and the testing data set (30%). The content of each data set are displayed next.

In [ ]:
```
import pandas as pd
df_train = pd.read_csv('data_train.csv', index_col=0)
df_train
```

Out[ ]:

	px	py	angle -179	angle -178	angle -177	angle -176	angle -175	angle -174	angle -173	angle -172	...	angle 172
0	-3.06	-4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0
1	-1.48	-4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0
2	1.58	-2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0
3	-3.10	-3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0
4	-1.48	-3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...
2235	2.60	-3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0
2236	-2.12	-4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0
2237	-2.80	-3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0
2238	1.18	-4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0
2239	2.52	-3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0

2240 rows × 363 columns

In [ ]:

```python
import pandas as pd
df_test = pd.read_csv('data_test.csv', index_col=0)
df_test
```

Out[ ]:

	px	py	angle -179	angle -178	angle -177	angle -176	angle -175	angle -174	angle -173	angle -172	...	angle 172
0	-3.46	-2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000
1	0.38	-2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000
2	2.58	-4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000
3	-2.84	-4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000
4	-2.56	-4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000
...	...	...	...	...	...	...	...	...	...	...	...	...
955	3.32	-1.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	2.752938
956	3.62	-4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000
957	2.20	-1.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	1.381054
958	-2.98	-4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000
959	2.78	-1.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	2.279001

960 rows × 363 columns

Note that there is an extra column (the label column) that indicates if the obstacle is a **circle (label 1)** or a **square (label 2)**.

1.1 Implement a **k-nearest neighbor (k-NN)** classifier that receives the parameter k, the sample to classify (that are the range measurements at one snapshot), and a set of labeled training data.

Do not use sklearn or similar packages (use the results of notebook #7).

In [ ]:
```
# To complete
X_train = df_train.iloc[:, np.arange(2,362,1)].values
Y_train = df_train["label"].values
data_train = df_train.iloc[:, np.arange(2,363,1)].values  #it also inclua

# KNN
def vector2norm(x, data):
  npoints = data.shape[0]
  distances = np.zeros(npoints)
  for i in range(npoints) :
    distances[i] = np.linalg.norm(x - data[i, :-1])
  return distances

def kNN_classifier(k, x, data):
  npoints = data.shape[0]
  # compute distance to training points
  dist = vector2norm(x, data)
  # sort along increasing distances
  ind = np.argsort(dist, axis=0)
  classes = data[:, -1]
  classes_sorted = classes[ind]
  # determine class with more elements in the k neighborhood
  c1 = 0
  c2 = 0
  for i in range(k):
    if classes_sorted[i]==1:
      c1 +=1
    else:
      c2 +=1
  if c1>c2:
    return 1
  else:
    return 2
```

1.2 Test the k-NN classifier for the `data_train.csv` set and for the `data_test.csv` set and obtain the respectively accuracy for $k = 1, 3, 5, 7, 9$

Note that accuracy is defined as

$$acc = \frac{\#\text{correct predictions}}{\#\text{all predictions}} = \frac{TP + TN}{TP + TN + FP + FN}$$

where TP are the true positives, TN true negatives, FP false positives, and FN the fale negatives.

In [ ]:
```
# Dict with test results
train = []

# Let's check for all trainning data
data_train = np.append(X_train, np.reshape(Y_train, (len(Y_train), 1)), a

# Compute KNN
classification = []
K = 1
for t in range(len(X_train)):
```

```
    if kNN_classifier(K, X_train[t], data_train) == 1:
        classification.append(1)
    else:
        classification.append(2)

# Classification of the training data (expected the same as the data_trai
correct = 0
for i in range(len(Y_train)):
    if classification[i] == Y_train[i]:
        correct +=1

acc = correct/len(Y_train)
print(f"Accuracy of the train model with k = {K} : {100.0*acc:4.2f}%")
train.append(100.0*acc)
```

```
Accuracy of the train model with k = 1 : 100.00%
```

In [ ]:
```
# Let's check for all trainning data
data_train = np.append(X_train, np.reshape(Y_train, (len(Y_train), 1)), a

# Compute KNN
classification = []
K = 3
for t in range(len(X_train)):
    if kNN_classifier(K, X_train[t], data_train) == 1:
        classification.append(1)
    else:
        classification.append(2)

# Classification of the training data (expected the same as the data_trai
correct = 0
for i in range(len(Y_train)):
    if classification[i] == Y_train[i]:
        correct +=1

acc = correct/len(Y_train)
print(f"Accuracy of the train model with k = {K} : {100.0*acc:4.2f}%")
train.append(100.0*acc)
```

```
Accuracy of the train model with k = 3 : 99.87%
```

In [ ]:
```
# Let's check for all trainning data
data_train = np.append(X_train, np.reshape(Y_train, (len(Y_train), 1)), a

# Compute KNN
classification = []
K = 5
for t in range(len(X_train)):
    if kNN_classifier(K, X_train[t], data_train) == 1:
        classification.append(1)
    else:
        classification.append(2)

# Classification of the training data (expected the same as the data_trai
correct = 0
for i in range(len(Y_train)):
    if classification[i] == Y_train[i]:
        correct +=1

acc = correct/len(Y_train)
```

```python
print(f"Accuracy of the train model with k = {K} : {100.0*acc:4.2f}%")
train.append(100.0*acc)
```

Accuracy of the train model with k = 5 : 99.64%

In [ ]:
```python
# Let's check for all trainning data
data_train = np.append(X_train, np.reshape(Y_train, (len(Y_train), 1)), a

# Compute KNN
classification = []
K = 7
for t in range(len(X_train)):
    if kNN_classifier(K, X_train[t], data_train) == 1:
        classification.append(1)
    else:
        classification.append(2)

# Classification of the training data (expected the same as the data_trai
correct = 0
for i in range(len(Y_train)):
    if classification[i] == Y_train[i]:
        correct +=1

acc = correct/len(Y_train)
print(f"Accuracy of the train model with k = {K} : {100.0*acc:4.2f}%")
train.append(100.0*acc)
```

Accuracy of the train model with k = 7 : 99.82%

In [ ]:
```python
# Let's check for all trainning data
data_train = np.append(X_train, np.reshape(Y_train, (len(Y_train), 1)), a

# Compute KNN
classification = []
K = 9
for t in range(len(X_train)):
    if kNN_classifier(K, X_train[t], data_train) == 1:
        classification.append(1)
    else:
        classification.append(2)

# Classification of the training data (expected the same as the data_trai
correct = 0
for i in range(len(Y_train)):
    if classification[i] == Y_train[i]:
        correct +=1

acc = correct/len(Y_train)
print(f"Accuracy of the train model with k = {K} : {100.0*acc:4.2f}%")
train.append(100.0*acc)
```

Accuracy of the train model with k = 9 : 99.78%

In [ ]:
```python
# Dict with test results
test = []

# Let's check for the testing data
X_test = df_test.iloc[:, np.arange(2,362,1)].values
Y_test = df_test["label"].values

# Let's check for all test data
```

```
data_test = np.append(X_test, np.reshape(Y_test, (len(Y_test), 1)), axis=

# Compute KNN
classification = []
K = 1
for t in range(len(X_test)):
    if kNN_classifier(K, X_test[t], data_train) == 1:
        classification.append(1)
    else:
        classification.append(2)

# Classification of the test data
#print(classification)

# printing the results
correct = 0
for i in range(len(Y_test)):
    if classification[i] == Y_test[i]:
        correct +=1

acc = correct/len(Y_test)
print(f"Accuracy of the test model with k = {K} : {100.0*acc:4.2f}%")
test.append(100.0*acc)
```

```
Accuracy of the test model with k = 1 : 100.00%
```

In [ ]:

```
# Let's check for the testing data
X_test = df_test.iloc[:, np.arange(2,362,1)].values
Y_test = df_test["label"].values

# Let's check for all test data
data_test = np.append(X_test, np.reshape(Y_test, (len(Y_test), 1)), axis=

# Compute KNN
classification = []
K = 3
for t in range(len(X_test)):
    if kNN_classifier(K, X_test[t], data_train) == 1:
        classification.append(1)
    else:
        classification.append(2)

# Classification of the test data
#print(classification)

# printing the results
correct = 0
for i in range(len(Y_test)):
    if classification[i] == Y_test[i]:
        correct +=1

acc = correct/len(Y_test)
print(f"Accuracy of the test model with k = {K} : {100.0*acc:4.2f}%")
test.append(100.0*acc)
```

```
Accuracy of the test model with k = 3 : 99.48%
```

In [ ]:

```
# Let's check for the testing data
X_test = df_test.iloc[:, np.arange(2,362,1)].values
Y_test = df_test["label"].values
```

```
# Let's check for all test data
data_test = np.append(X_test, np.reshape(Y_test, (len(Y_test), 1)), axis=

# Compute KNN
classification = []
K = 5
for t in range(len(X_test)):
    if kNN_classifier(K, X_test[t], data_train) == 1:
        classification.append(1)
    else:
        classification.append(2)

# Classification of the test data
#print(classification)

# printing the results
correct = 0
for i in range(len(Y_test)):
    if classification[i] == Y_test[i]:
        correct +=1

acc = correct/len(Y_test)
print(f"Accuracy of the test model with k = {K} : {100.0*acc:4.2f}%")
test.append(100.0*acc)
```

```
Accuracy of the test model with k = 5 : 99.27%
```

In [ ]:

```
# Let's check for the testing data
X_test = df_test.iloc[:, np.arange(2,362,1)].values
Y_test = df_test["label"].values

# Let's check for all test data
data_test = np.append(X_test, np.reshape(Y_test, (len(Y_test), 1)), axis=

# Compute KNN
classification = []
K = 7
for t in range(len(X_test)):
    if kNN_classifier(K, X_test[t], data_train) == 1:
        classification.append(1)
    else:
        classification.append(2)

# Classification of the test data
#print(classification)

# printing the results
correct = 0
for i in range(len(Y_test)):
    if classification[i] == Y_test[i]:
        correct +=1

acc = correct/len(Y_test)
print(f"Accuracy of the test model with k = {K} : {100.0*acc:4.2f}%")
test.append(100.0*acc)
```

```
Accuracy of the test model with k = 7 : 99.69%
```

In [ ]:

```
# Let's check for the testing data
X_test = df_test.iloc[:, np.arange(2,362,1)].values
Y_test = df_test["label"].values
```

```
# Let's check for all test data
data_test = np.append(X_test, np.reshape(Y_test, (len(Y_test), 1)), axis=

# Compute KNN
classification = []
K = 9
for t in range(len(X_test)):
    if kNN_classifier(K, X_test[t], data_train) == 1:
        classification.append(1)
    else:
        classification.append(2)

# Classification of the test data
#print(classification)

# printing the results
correct = 0
for i in range(len(Y_test)):
    if classification[i] == Y_test[i]:
        correct +=1

acc = correct/len(Y_test)
print(f"Accuracy of the test model with k = {K} : {100.0*acc:4.2f}%")
test.append(100.0*acc)
```

```
Accuracy of the test model with k = 9 : 99.58%
```

In [ ]:

```
# Results Obtained

# Sample data
k_values = [1, 3, 5, 7, 9]

# Create a dictionary with the data
data = {
    'K Value': k_values,
    'Train Accuracy': train,
    'Test Accuracy': test
}

# Create a DataFrame
df = pd.DataFrame(data)

# Set 'K Value' as the index
df.set_index('K Value', inplace=True)

# Round the accuracy values to 2 decimal places
df = df.round(2)

# Display the table
df
```

Out[ ]:

K Value	Train Accuracy	Test Accuracy
1	100.00	100.00
3	99.87	99.48
5	99.64	99.27
7	99.82	99.69
9	99.78	99.58

1.3 Implement an Artificial Neural Network (ANN) of the type multi-layer perceptron (MLP) with

1. an input layer that receives the first 10 nonzero range measurements (for each snapshot);
2. one hidden layer with 5 neurons with activation functions of the type ReLU (rectified linear unit);
3. an output layer with 1 neuron with a sigmoid activation function;
4. a loss function of the type mean square error.

Train the ANN using the `data_train.csv` set.

Do not use PyTorch, TensorFlow or similar packages (check notebook #8).

Tip: It is important to shuffle the training data. You may get better results with non constant learning rate. A final loss below 0.05 is good!

In [ ]:

```
import numpy as np
np.random.seed(42)

N_INPUTS = 10  #Number of inputs

def mse_loss(y_true, y_pred):
  return ((y_true - y_pred) ** 2).mean()

# Sigmoid activation function: f(x) = 1 / (1 + e^(-x))
def sigmoid(x):
  return 1 / (1 + np.exp(-x))

# Derivative of sigmoid: f'(x) = f(x) * (1 - f(x))
def deriv_sigmoid(x):
  fx = sigmoid(x)
  return fx * (1 - fx)

# ReLu activation function:
def relu(x):
  if x > 0:
    return x
  else :
    return 0
```

```
# Derivative of ReLu
def deriv_relu(x):
  if x > 0:
    return 1
  else :
    return 0

class NeuralNetwork:
  '''
  Structure of the neural network:
    - N_INPUTS inputs
    - a hidden layer with 5 neurons (h1, h2, h3, h4, h5)
    - an output layer with 1 neuron (o1)
  '''
  def __init__(self):

    # Biases
    self.b1 = np.random.random()
    self.b2 = np.random.random()
    self.b3 = np.random.random()
    self.b4 = np.random.random()
    self.b5 = np.random.random()
    self.bo = np.random.random()

    # # Weights
    self.w1o, self.w2o, self.w3o, self.w4o, self.w5o = np.random.random(5
    self.wi1 = np.random.random(N_INPUTS)
    self.wi2 = np.random.random(N_INPUTS)
    self.wi3 = np.random.random(N_INPUTS)
    self.wi4 = np.random.random(N_INPUTS)
    self.wi5 = np.random.random(N_INPUTS)


  def feedforward(self, x):
    '''
    - x is a numpy array with N_INPUTS elements.
    '''

    # # Hidden layer
    self.sum_h1 = np.dot(self.wi1, x) + self.b1
    self.h1 = relu(self.sum_h1)
    self.sum_h2 = np.dot(self.wi2, x) + self.b2
    self.h2 = relu(self.sum_h2)
    self.sum_h3 = np.dot(self.wi3, x) + self.b3
    self.h3 = relu(self.sum_h3)
    self.sum_h4 = np.dot(self.wi4, x) + self.b4
    self.h4 = relu(self.sum_h4)
    self.sum_h5 = np.dot(self.wi5, x) + self.b5
    self.h5 = relu(self.sum_h5)

    # Output layer
    self.sum_o1 = self.w1o*self.h1 + self.w2o*self.h2 + self.w3o*self.h3
    self.o1 = sigmoid(self.sum_o1)
    return self.o1


  def train(self, data, y_trues, learn_rate = 0.1, epochs = 500):
    '''
    - data is a (n x N_INPUTS) numpy array, n = # of samples in the datas
    - y_trues is a numpy array with n elements.
```

```
      Elements in y_true correspond to those in data.
    '''
    loss_prev = 10000  #loss_prev is the loss of the previous iteration
    for epoch in range(epochs):
      for x, y_true in zip(data, y_trues):

        # ***************************************************
        # 1. Feedforward Step
        y_pred = self.feedforward(x)

        # ***************************************************
        # 2. Backpropagation Step

        # Partial derivatives.
        d_L_d_ypred = -2 * (y_true - y_pred)

        # Output Layer:  Neuron o1
        d_ypred_d_w1o = self.h1 * deriv_sigmoid(self.sum_o1)
        d_ypred_d_w2o = self.h2 * deriv_sigmoid(self.sum_o1)
        d_ypred_d_w3o = self.h3 * deriv_sigmoid(self.sum_o1)
        d_ypred_d_w4o = self.h4 * deriv_sigmoid(self.sum_o1)
        d_ypred_d_w5o = self.h5 * deriv_sigmoid(self.sum_o1)
        d_ypred_d_bo = deriv_sigmoid(self.sum_o1)

        d_ypred_d_h1 = self.w1o * deriv_sigmoid(self.sum_o1)
        d_ypred_d_h2 = self.w2o * deriv_sigmoid(self.sum_o1)
        d_ypred_d_h3 = self.w3o * deriv_sigmoid(self.sum_o1)
        d_ypred_d_h4 = self.w4o * deriv_sigmoid(self.sum_o1)
        d_ypred_d_h5 = self.w5o * deriv_sigmoid(self.sum_o1)

        # Hidden Layer: Neuron h1
        d_h1_d_wi1 = x * deriv_relu(self.sum_h1)
        d_h1_d_b1 = deriv_relu(self.sum_h1)

        # Hidden Layer: Neuron h2
        d_h2_d_wi2 = x * deriv_relu(self.sum_h2)
        d_h2_d_b2 = deriv_relu(self.sum_h2)

        # Hidden Layer: Neuron h3
        d_h3_d_wi3 = x * deriv_relu(self.sum_h3)
        d_h3_d_b3 = deriv_relu(self.sum_h3)

        # Hidden Layer: Neuron h4
        d_h4_d_wi4 = x * deriv_relu(self.sum_h4)
        d_h4_d_b4 = deriv_relu(self.sum_h4)

        # Hidden Layer: Neuron h5
        d_h5_d_wi5 = x * deriv_relu(self.sum_h5)
        d_h5_d_b5 = deriv_relu(self.sum_h5)

        # ***************************************************
        # 3. Gradient Descent
        # Output Layer:  Neuron o1
        self.w1o -= learn_rate * d_L_d_ypred * d_ypred_d_w1o
        self.w2o -= learn_rate * d_L_d_ypred * d_ypred_d_w2o
        self.w3o -= learn_rate * d_L_d_ypred * d_ypred_d_w3o
        self.w4o -= learn_rate * d_L_d_ypred * d_ypred_d_w4o
        self.w5o -= learn_rate * d_L_d_ypred * d_ypred_d_w5o
        self.bo -= learn_rate * d_L_d_ypred * d_ypred_d_bo
```

```
          # Hidden Layer: Neuron h1
          self.wi1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_wi1
          self.b1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_b1

          # Hidden Layer: Neuron h2
          self.wi2 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_wi2
          self.b2 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_b2

          # Hidden Layer: Neuron h3
          self.wi3 -= learn_rate * d_L_d_ypred * d_ypred_d_h3 * d_h3_d_wi3
          self.b3 -= learn_rate * d_L_d_ypred * d_ypred_d_h3 * d_h3_d_b3

          # Hidden Layer: Neuron h4
          self.wi4 -= learn_rate * d_L_d_ypred * d_ypred_d_h4 * d_h4_d_wi4
          self.b4 -= learn_rate * d_L_d_ypred * d_ypred_d_h4 * d_h4_d_b4

          # Hidden Layer: Neuron h5
          self.wi5 -= learn_rate * d_L_d_ypred * d_ypred_d_h5 * d_h5_d_wi5
          self.b5 -= learn_rate * d_L_d_ypred * d_ypred_d_h5 * d_h5_d_b5

        # *************************************************
        # 4. Performance assessment (per epoch)
        if epoch % 5 == 0:
          y_preds = np.apply_along_axis(self.feedforward, 1, data)
          loss = mse_loss(y_trues, y_preds)
          print("Epoch %d  --> Loss: %.4f" % (epoch, loss))
# Uncomment this part to enable a nonconstant learning rate
          if loss > loss_prev:  #if loss did not decrease, let's decrease t
           if learn_rate > 0.002:
             learn_rate = learn_rate*.9  #decrease 90% of the previous valu
           print("I'm at epoch", epoch, "with new learn_rate: ", learn_rate
          loss_prev = loss


# Create the ANN
model = NeuralNetwork()

# Build the Trainingset (with the first nonzero N_INPUTS ranges)
trainingset_X = np.zeros([len(Y_train), N_INPUTS])
for t in range(len(Y_train)):
  j=0
  for i in range(360):
    if X_train[t][i] > 0:
      if j < N_INPUTS:
        trainingset_X[t][j] = X_train[t][i]
        j +=1

# Trainingset: here the labels are 0 or 1
trainingset_Y = Y_train-1

# Shuffling the set...
from sklearn.utils import shuffle
trainingset_X, trainingset_Y = shuffle(trainingset_X, trainingset_Y, rand

# Train the ANN
model.train(trainingset_X, trainingset_Y, learn_rate = 0.1, epochs = 1000
```

```
Epoch 0  --> Loss: 0.4999
Epoch 5  --> Loss: 0.2116
Epoch 10  --> Loss: 0.1859
Epoch 15  --> Loss: 0.1925
I'm at epoch 15 with new learn_rate:  0.09000000000000001
Epoch 20  --> Loss: 0.1873
Epoch 25  --> Loss: 0.1733
Epoch 30  --> Loss: 0.1711
Epoch 35  --> Loss: 0.1866
I'm at epoch 35 with new learn_rate:  0.08100000000000002
Epoch 40  --> Loss: 0.1702
Epoch 45  --> Loss: 0.1711
I'm at epoch 45 with new learn_rate:  0.07290000000000002
Epoch 50  --> Loss: 0.1647
Epoch 55  --> Loss: 0.1650
I'm at epoch 55 with new learn_rate:  0.06561000000000002
Epoch 60  --> Loss: 0.1618
Epoch 65  --> Loss: 0.1599
Epoch 70  --> Loss: 0.1593
Epoch 75  --> Loss: 0.1579
Epoch 80  --> Loss: 0.1567
Epoch 85  --> Loss: 0.1558
Epoch 90  --> Loss: 0.1547
Epoch 95  --> Loss: 0.1536
Epoch 100  --> Loss: 0.1522
Epoch 105  --> Loss: 0.1510
Epoch 110  --> Loss: 0.1493
Epoch 115  --> Loss: 0.1477
Epoch 120  --> Loss: 0.1469
Epoch 125  --> Loss: 0.1450
Epoch 130  --> Loss: 0.1399
Epoch 135  --> Loss: 0.1391
Epoch 140  --> Loss: 0.1377
Epoch 145  --> Loss: 0.1371
Epoch 150  --> Loss: 0.1389
I'm at epoch 150 with new learn_rate:  0.05904900000000002
Epoch 155  --> Loss: 0.1358
Epoch 160  --> Loss: 0.1057
Epoch 165  --> Loss: 0.1106
I'm at epoch 165 with new learn_rate:  0.05314410000000002
Epoch 170  --> Loss: 0.1190
I'm at epoch 170 with new learn_rate:  0.04782969000000002
Epoch 175  --> Loss: 0.1047
Epoch 180  --> Loss: 0.0612
Epoch 185  --> Loss: 0.0655
I'm at epoch 185 with new learn_rate:  0.043046721000000024
Epoch 190  --> Loss: 0.0488
Epoch 195  --> Loss: 0.0557
I'm at epoch 195 with new learn_rate:  0.03874204890000002
Epoch 200  --> Loss: 0.0637
I'm at epoch 200 with new learn_rate:  0.03486784401000002
Epoch 205  --> Loss: 0.0948
I'm at epoch 205 with new learn_rate:  0.03138105960900001
Epoch 210  --> Loss: 0.1467
I'm at epoch 210 with new learn_rate:  0.028242953648100012
Epoch 215  --> Loss: 0.1152
Epoch 220  --> Loss: 0.1247
I'm at epoch 220 with new learn_rate:  0.025418658283290013
Epoch 225  --> Loss: 0.0625
Epoch 230  --> Loss: 0.0956
```

```
I'm at epoch 230 with new learn_rate:  0.022876792454961013
Epoch 235  --> Loss: 0.0510
Epoch 240  --> Loss: 0.0497
Epoch 245  --> Loss: 0.0483
Epoch 250  --> Loss: 0.0479
Epoch 255  --> Loss: 0.0476
Epoch 260  --> Loss: 0.0473
Epoch 265  --> Loss: 0.0411
Epoch 270  --> Loss: 0.0413
I'm at epoch 270 with new learn_rate:  0.020589113209464913
Epoch 275  --> Loss: 0.0536
I'm at epoch 275 with new learn_rate:  0.01853020188851842
Epoch 280  --> Loss: 0.0473
Epoch 285  --> Loss: 0.0460
Epoch 290  --> Loss: 0.0453
Epoch 295  --> Loss: 0.0452
Epoch 300  --> Loss: 0.0449
Epoch 305  --> Loss: 0.0447
Epoch 310  --> Loss: 0.0446
Epoch 315  --> Loss: 0.0446
I'm at epoch 315 with new learn_rate:  0.01667718169966658
Epoch 320  --> Loss: 0.0450
I'm at epoch 320 with new learn_rate:  0.015009463529699923
Epoch 325  --> Loss: 0.0453
I'm at epoch 325 with new learn_rate:  0.013508517176729932
Epoch 330  --> Loss: 0.0453
I'm at epoch 330 with new learn_rate:  0.01215766545905694
Epoch 335  --> Loss: 0.0453
I'm at epoch 335 with new learn_rate:  0.010941898913151246
Epoch 340  --> Loss: 0.0448
Epoch 345  --> Loss: 0.0448
Epoch 350  --> Loss: 0.0448
Epoch 355  --> Loss: 0.0447
Epoch 360  --> Loss: 0.0447
Epoch 365  --> Loss: 0.0447
I'm at epoch 365 with new learn_rate:  0.009847709021836121
Epoch 370  --> Loss: 0.0443
Epoch 375  --> Loss: 0.0443
I'm at epoch 375 with new learn_rate:  0.00886293811965251
Epoch 380  --> Loss: 0.0438
Epoch 385  --> Loss: 0.0438
I'm at epoch 385 with new learn_rate:  0.007976644307687259
Epoch 390  --> Loss: 0.0432
Epoch 395  --> Loss: 0.0432
Epoch 400  --> Loss: 0.0432
Epoch 405  --> Loss: 0.0432
Epoch 410  --> Loss: 0.0432
Epoch 415  --> Loss: 0.0432
Epoch 420  --> Loss: 0.0431
Epoch 425  --> Loss: 0.0431
Epoch 430  --> Loss: 0.0431
Epoch 435  --> Loss: 0.0431
Epoch 440  --> Loss: 0.0431
Epoch 445  --> Loss: 0.0430
Epoch 450  --> Loss: 0.0430
Epoch 455  --> Loss: 0.0430
Epoch 460  --> Loss: 0.0429
Epoch 465  --> Loss: 0.0429
Epoch 470  --> Loss: 0.0428
Epoch 475  --> Loss: 0.0428
```

```
Epoch 480  --> Loss: 0.0427
Epoch 485  --> Loss: 0.0427
Epoch 490  --> Loss: 0.0426
Epoch 495  --> Loss: 0.0425
Epoch 500  --> Loss: 0.0425
Epoch 505  --> Loss: 0.0423
Epoch 510  --> Loss: 0.0422
Epoch 515  --> Loss: 0.0421
Epoch 520  --> Loss: 0.0419
Epoch 525  --> Loss: 0.0418
Epoch 530  --> Loss: 0.0417
Epoch 535  --> Loss: 0.0416
Epoch 540  --> Loss: 0.0415
Epoch 545  --> Loss: 0.0415
Epoch 550  --> Loss: 0.0413
Epoch 555  --> Loss: 0.0413
Epoch 560  --> Loss: 0.0412
Epoch 565  --> Loss: 0.0412
Epoch 570  --> Loss: 0.0411
Epoch 575  --> Loss: 0.0410
Epoch 580  --> Loss: 0.0410
Epoch 585  --> Loss: 0.0410
Epoch 590  --> Loss: 0.0410
I'm at epoch 590 with new learn_rate:  0.007178979876918534
Epoch 595  --> Loss: 0.0411
I'm at epoch 595 with new learn_rate:  0.006461081889226681
Epoch 600  --> Loss: 0.0405
Epoch 605  --> Loss: 0.0405
I'm at epoch 605 with new learn_rate:  0.005814973700304013
Epoch 610  --> Loss: 0.0396
Epoch 615  --> Loss: 0.0396
Epoch 620  --> Loss: 0.0395
Epoch 625  --> Loss: 0.0395
Epoch 630  --> Loss: 0.0395
Epoch 635  --> Loss: 0.0394
Epoch 640  --> Loss: 0.0394
Epoch 645  --> Loss: 0.0393
Epoch 650  --> Loss: 0.0393
Epoch 655  --> Loss: 0.0393
Epoch 660  --> Loss: 0.0392
Epoch 665  --> Loss: 0.0392
Epoch 670  --> Loss: 0.0391
Epoch 675  --> Loss: 0.0391
Epoch 680  --> Loss: 0.0391
Epoch 685  --> Loss: 0.0391
Epoch 690  --> Loss: 0.0389
Epoch 695  --> Loss: 0.0389
Epoch 700  --> Loss: 0.0389
Epoch 705  --> Loss: 0.0389
Epoch 710  --> Loss: 0.0388
Epoch 715  --> Loss: 0.0388
Epoch 720  --> Loss: 0.0388
Epoch 725  --> Loss: 0.0388
Epoch 730  --> Loss: 0.0388
Epoch 735  --> Loss: 0.0387
Epoch 740  --> Loss: 0.0387
Epoch 745  --> Loss: 0.0387
Epoch 750  --> Loss: 0.0386
Epoch 755  --> Loss: 0.0386
Epoch 760  --> Loss: 0.0386
```

```
Epoch 765  --> Loss: 0.0386
Epoch 770  --> Loss: 0.0385
Epoch 775  --> Loss: 0.0385
Epoch 780  --> Loss: 0.0385
Epoch 785  --> Loss: 0.0384
Epoch 790  --> Loss: 0.0384
Epoch 795  --> Loss: 0.0383
Epoch 800  --> Loss: 0.0382
Epoch 805  --> Loss: 0.0380
Epoch 810  --> Loss: 0.0380
Epoch 815  --> Loss: 0.0379
Epoch 820  --> Loss: 0.0378
Epoch 825  --> Loss: 0.0376
Epoch 830  --> Loss: 0.0376
I'm at epoch 830 with new learn_rate:  0.005233476330273611
Epoch 835  --> Loss: 0.0373
Epoch 840  --> Loss: 0.0372
Epoch 845  --> Loss: 0.0372
Epoch 850  --> Loss: 0.0372
Epoch 855  --> Loss: 0.0371
Epoch 860  --> Loss: 0.0371
Epoch 865  --> Loss: 0.0371
Epoch 870  --> Loss: 0.0371
Epoch 875  --> Loss: 0.0370
Epoch 880  --> Loss: 0.0369
Epoch 885  --> Loss: 0.0369
Epoch 890  --> Loss: 0.0370
I'm at epoch 890 with new learn_rate:  0.00471012869724625
Epoch 895  --> Loss: 0.0367
Epoch 900  --> Loss: 0.0367
Epoch 905  --> Loss: 0.0367
Epoch 910  --> Loss: 0.0366
Epoch 915  --> Loss: 0.0366
Epoch 920  --> Loss: 0.0366
Epoch 925  --> Loss: 0.0365
Epoch 930  --> Loss: 0.0365
Epoch 935  --> Loss: 0.0365
Epoch 940  --> Loss: 0.0364
Epoch 945  --> Loss: 0.0364
Epoch 950  --> Loss: 0.0364
Epoch 955  --> Loss: 0.0364
Epoch 960  --> Loss: 0.0364
Epoch 965  --> Loss: 0.0364
Epoch 970  --> Loss: 0.0364
Epoch 975  --> Loss: 0.0363
Epoch 980  --> Loss: 0.0363
Epoch 985  --> Loss: 0.0363
Epoch 990  --> Loss: 0.0363
Epoch 995  --> Loss: 0.0363
```

1.4 Test the ANN classifier for the `data_train.csv` set and for the `data_test.csv` set and obtain the respectively accuracy. Write in a brief sentence of the main conclusions about the classifiers (k-NN and ANN) until this point.

In [ ]:
```python
# To complete

# Evaluation with the Training set
classification = []
for t in range(len(Y_train)):
```

```
  classification.append( model.feedforward(trainingset_X[t]) )
classification = np.array(classification)
error_clas = 0

for t in range(len(classification)):
  if (classification[t] >= 0.5) and trainingset_Y[t]  == 0:
    error_clas +=1
  if (classification[t] < 0.5) and trainingset_Y[t]  == 1:
    error_clas +=1

print("Number of misclassified samples in the training data: ", error_cla
acc = 1 - (error_clas/len(Y_train))
print(f"Accuracy of the model: {100.0*acc:4.2f}%")


# Evaluation with Test_set

# Build the Testset (with the first nonzero N_INPUTS ranges)
testset_X = np.zeros([len(Y_test), N_INPUTS])
for t in range(len(Y_test)):
  j=0
  for i in range(360):
    if X_test[t][i] > 0:
      if j < N_INPUTS:
        testset_X[t][j] = X_test[t][i]
        j +=1

# Testset: here the labels are 0 or 1
testset_Y = Y_test-1

classification = []
for t in range(len(Y_test)):
  classification.append( model.feedforward(testset_X[t]) )
classification = np.array(classification)
error_clas = 0

for t in range(len(classification)):
  if (classification[t] >= 0.5) and testset_Y[t]  == 0:
    error_clas +=1
  if (classification[t] < 0.5) and testset_Y[t]  == 1:
    error_clas +=1

print("Number of misclassified samples in the testing data: ", error_clas
acc = 1 - (error_clas/len(Y_test))
print(f"Accuracy of the model: {100.0*acc:4.2f}%")
```

```
Number of misclassified samples in the training data:  89 in 2240
Accuracy of the model: 96.03%
Number of misclassified samples in the testing data:  43 in 960
Accuracy of the model: 95.52%
```

Main Conclusion

KNN vs ANN

K-Nearest Neighbors (KNN) and Artificial Neural Networks (ANN) showcase distinctive strengths and weaknesses depending on the complexity and nature of the dataset. In simpler scenarios, like distinguishing between circles and squares in point cloud classification, KNN's reliance on stored instances allows it to excel due to its simplicity and intuitive nature. Conversely, ANN, with its intricate architecture capable of capturing complex relationships, might struggle to demonstrate its full potential in such straightforward tasks. However, as the complexity of the data increases, the adaptive nature of ANN becomes advantageous, enabling it to model intricate patterns and nuances that KNN might overlook. Thus, while KNN might outperform ANN in simpler tasks, the latter often shines in more complex scenarios, leveraging its adaptability and ability to discern intricate relationships within data.

Part 2: Classification of two objects

We would like now to use the previous ANN classifier to the data in `data_test2obs.csv` that may have two objects at the same snapshot. The idea is to before send the range measurements to the classifier, apply first a k-means at each snapshot to separate the data into two sub-sets such that each sub-set only contains data of one object. Then, send each subset of data to the ANN classifier.

2.1 Implement the k-means algorithm and test it for two snapshots **converted to the 2D map** (that is, the input data for the k-means is the 2D map) of the dataset `data_test2obs.csv` for

1. $t = 1\ s$ (which has only one object) and
2. for $t = 32\ s$ (which has 2 objects).

What can you conclude? Do not use sklearn or similar packages (use the results of notebook #10).

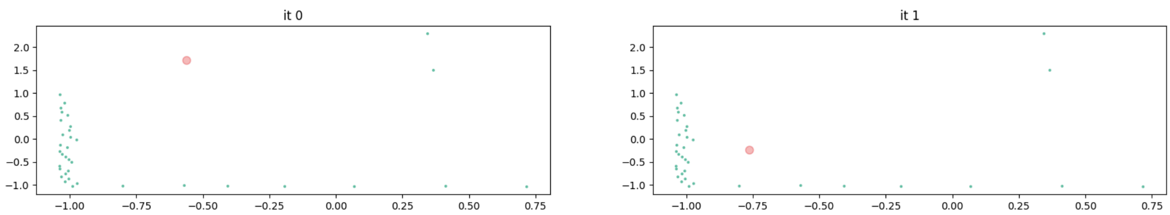
In [ ]:
```
import pandas as pd
df_test2obs = pd.read_csv('data_test2obs.csv', index_col=0)

Lidar_range = df_test2obs.iloc[:, np.arange(2,362,1)].values
px = df_test2obs["px"].values
py = df_test2obs["py"].values
```

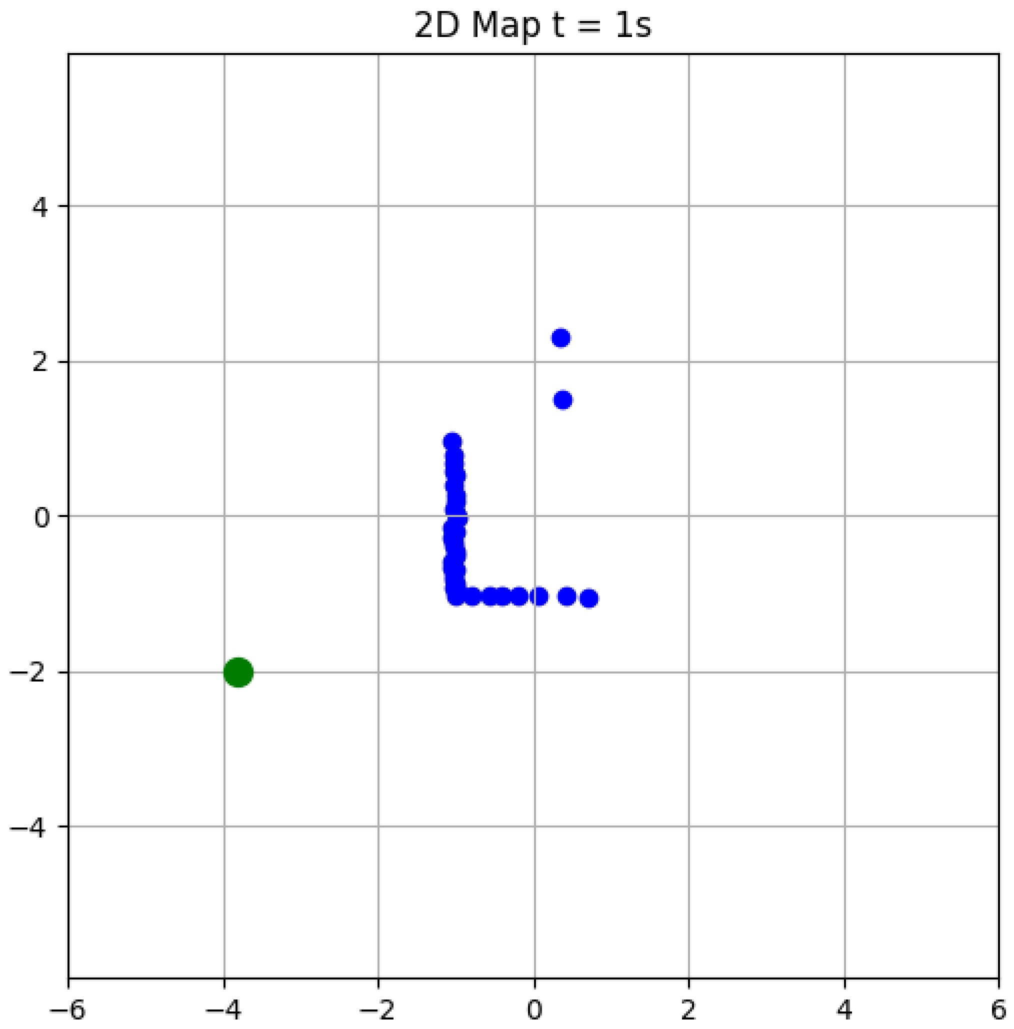
In [ ]:
```
def kmeans_func(X_func, K_func, show_plot=0, ini_method=0):
  '''
    X_func are sample points;
    K_func is the number of clusters;
    showplot 0/1 - not / show centroid evolution over iterations;
    ini_method 0/1 - ini centroids 0=> with random within data bounderies
  '''
  cluster = np.zeros(X_func.shape[0],dtype=int)
  centr_ini = []
  if ini_method:  # 1 is points, 0 is random
    # Initial centroids are sample points (risky!)
    for i in range(K_func):
      centr_ini.append(X_func[i])
  else:
    # Non absurd Random Initial Centroids
    max0 = max(X_func[:,0]);
    min0 = min(X_func[:,0]);
```

```
    max1 = max(X_func[:,1]);
    min1 = min(X_func[:,1]);

    for i in range(K_func):
      centr_ini.append([random.uniform(min0,max0),random.uniform(min1,max

  iter = 0
  diff = 1
  centroids = centr_ini
  centr_list = [centroids]

  while diff:
    # for each sample
    for sample_i, sample_pt in enumerate(X_func):
      min_dist = float('inf')
      # dist of the point from all centroids
      for centroid_i, centroid in enumerate(centroids):
        dist =  np.sqrt( (sample_pt[0] - centroid[0])**2 + (sample_pt[1]
        # store closest centroid
        if dist < min_dist:
          min_dist = dist
          cluster[sample_i] = centroid_i


    sum = np.zeros((K_func,2))
    cnt = np.zeros(K_func)
    for sample_i, sample_pt in enumerate(X_func):
      sum[cluster[sample_i]] += X_func[sample_i]
      cnt[cluster[sample_i]] += 1
    new_centroids = np.zeros((K_func,2))
    for k in range(K_func):
      if (cnt[k]>0):
        new_centroids[k] = sum[k] / cnt[k]
      else:
        # hopefully, the next iteration will fix uninteresting centroid
        new_centroids[k] = [random.uniform(min0,max0),random.uniform(min1


    # if centroids are same then leave
    if np.count_nonzero(centroids-new_centroids) == 0:
      diff = 0
    else:
      centroids = new_centroids
      centr_list.append(new_centroids)
    iter = iter+1

  #print("Number of iterations", iter)
  #print(centr_list)

  if show_plot: # pretty subplotting
    cols = min(iter,6) # max 6 plots
    fig, ax = plt.subplots(nrows=1, ncols=cols, figsize=(20,3))
    for col in range(min(cols,iter)):
      ax[col].scatter(X_func[:, 0], X_func[:, 1], s=3, c=np.zeros(X_func.
      i = round(iter/cols*col)
      if (col==cols-1):
        i = iter-1
      ## print("centr %s", i,"=>", centr_list[i])
      ## devia funcionar ### ax[row, col].scatter(centr_list[i][:,0], cen
      for j in range(K_func):
```

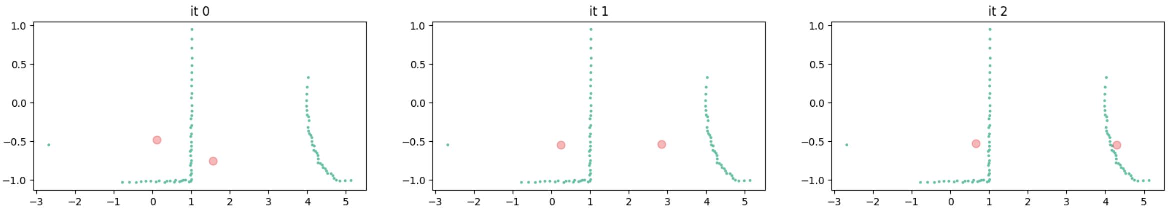
```
            centr_plot_x = centr_list[i][j][0]
            centr_plot_y = centr_list[i][j][1]
            ax[col].scatter(centr_plot_x, centr_plot_y, c=j, s=60, alpha=0.3,
            ax[col].title.set_text("it " + str(i) )
        plt.show()

    return centroids, cluster
```

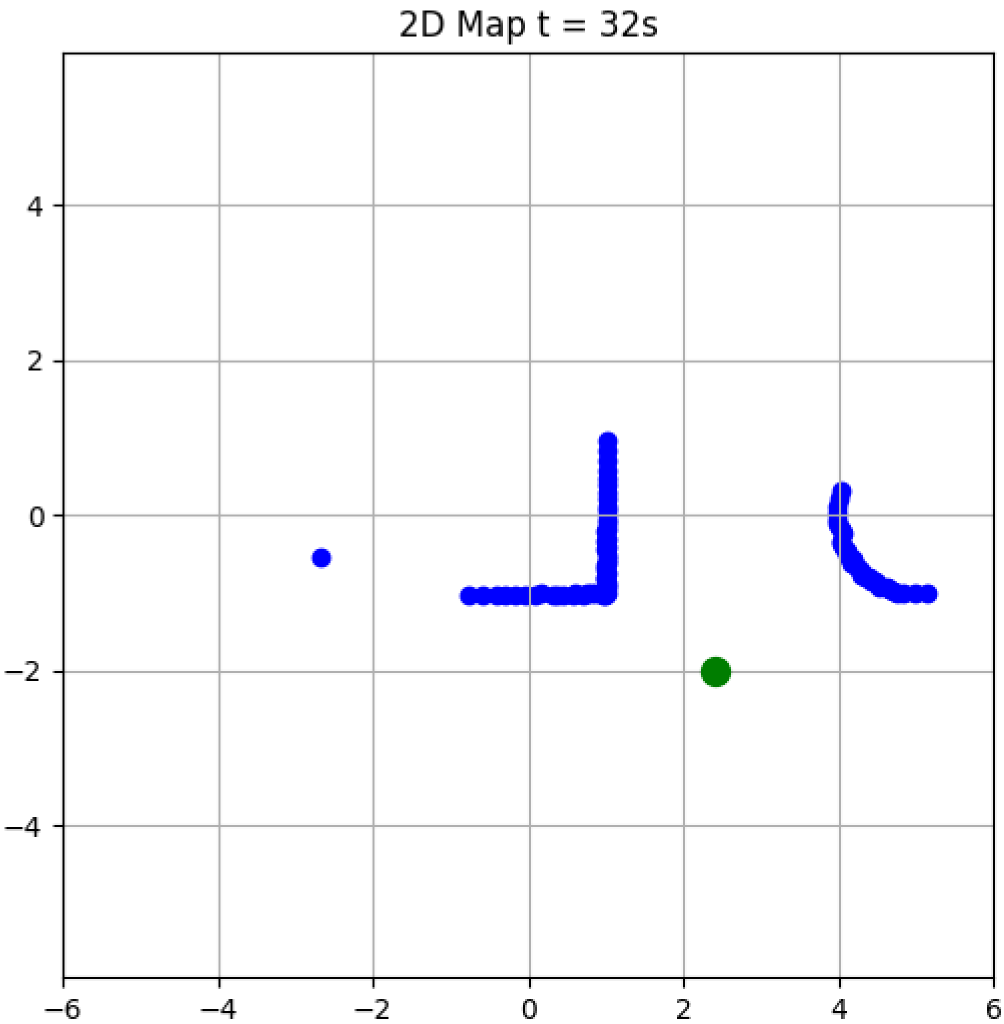
In [ ]:
```
# Build the cloud points in 2D map (t = 1s)
x_o, y_o = [], []

t=10*1
for i in range(len(Lidar_range[t])):
  if Lidar_range[t][i] > 0:
    x_o.append(px[t]+Lidar_range[t][i]*np.cos(angle[i]/180*np.pi))
    y_o.append(py[t]+Lidar_range[t][i]*np.sin(angle[i]/180*np.pi))

X = np.array([x_o, y_o]).T
centr, clust = kmeans_func(X, 1, show_plot=1, ini_method=0)

fig, ax = plt.subplots(figsize=(6,6))
ax.axis('equal')
xdim, ydim = 5, 5
plt.xlim(-xdim-1,xdim+1)
plt.ylim(-ydim-1,ydim+1)
plt.plot(px[t], py[t], 'g.', ms=20)  #position of the robot
plt.grid()

plt.scatter(x_o, y_o, color='b')
plt.title('2D Map t = 1s')
plt.show()
```





In [ ]:
```
# Build the cloud points in 2D map (t = 1s)
x_o, y_o = [], []

t=10*32
for i in range(len(Lidar_range[t])):
  if Lidar_range[t][i] > 0:
    x_o.append(px[t]+Lidar_range[t][i]*np.cos(angle[i]/180*np.pi))
    y_o.append(py[t]+Lidar_range[t][i]*np.sin(angle[i]/180*np.pi))

X = np.array([x_o, y_o]).T
centr, clust = kmeans_func(X, 2, show_plot=1, ini_method=0)

fig, ax = plt.subplots(figsize=(6,6))
ax.axis('equal')
xdim, ydim = 5, 5
plt.xlim(-xdim-1,xdim+1)
plt.ylim(-ydim-1,ydim+1)
plt.plot(px[t], py[t], 'g.', ms=20)  #position of the robot
plt.grid()

plt.scatter(x_o, y_o, color='b')
plt.title('2D Map t = 32s')
plt.show()
```





Main Conclusion

K-Means

Based on the results of clustering the LIDAR data at t = 1s and t = 32s using k-means, we conclude that k-means is robust in detecting and differentiating objects. At t = 1s, it identified a single object, and at t = 32s, it distinguished between two objects, demonstrating its adaptability and accuracy.

To determine the best number of clusters (K), we used the Sum of Squared Errors (SSE) and its derivative. SSE measures the compactness of clusters, decreasing as K increases. By examining the SSE derivative, we identified the point where increasing K no longer significantly reduces SSE, thus preventing overfitting. This method ensures optimal clustering performance by balancing compactness and generalization.

2.2 Using the previous results,

1. implement a method to automatically identify for each snapshot if it has 1 or 2 objects;
2. build a new test set with all the data in `data_test2obs.csv`, but now the new test set only has 1 object in each snapshot (and therefore this data set has more lines);
3. test this new data set using the ANN classifier.

Plot the position of the robot and the classified objects for some snapshots. What are the main conclusions?

In [ ]:
```
def plot_snapshot(t, X, px, py, num_objects, classification):
    '''
      Plot the result obtained at each snapshot
    '''

    def get_String(classification):
      '''
        Classification number to string
      '''
      if classification == 1:
        return "Square"
      else:
        return "Circle"

    plt.figure(figsize=(8, 6))

    # Highlight classified objects
    if num_objects == 1:
        plt.scatter(X[:, 0], X[:, 1], color='blue', label=f'Classified Ob
    else:
        plt.scatter(X[:, 0], X[:, 1], color='blue', label=f'Classified Ob
        plt.scatter(X[:, 0], X[:, 1], color='blue', label=f'Classified Ob

    # Plot robot position
    plt.scatter(px[t], py[t], c='green', marker='x', s=100, label='Robot
    plt.title(f'Point Cloud at t = {t/10}')
    plt.xlabel('X Position')
    plt.ylabel('Y Position')
    plt.legend()
    plt.show()

def SSE(X, centroids, cluster):
  '''
    returns the Sum of Squared Error
    X are the 2D points
    centroids are the cluster centers
    cluster is the cluster that each data point belongs to
  '''
  sum = 0
  for i, val in enumerate(X):
    sum += np.sqrt((centroids[cluster[i], 0]-val[0])**2 +(centroids[clust
  return sum
```

In [ ]:
```
Lidar_range = df_test2obs.iloc[:, np.arange(2,362,1)].values
px = df_test2obs["px"].values
py = df_test2obs["py"].values
```

```
# Building the data set for ANN testing using k-means to split objects
test_set_X = np.array([])
# Keep track of time (duplicated time means that we detected 2 objects at
time = []
classification = []
# Measurements for each t
for t in range(len(Lidar_range)):
  x_o, y_o = [], []
  for i in range(len(Lidar_range[t])):
    if Lidar_range[t][i] > 0:
      x_o.append(px[t]+Lidar_range[t][i]*np.cos(angle[i]/180*np.pi))
      y_o.append(py[t]+Lidar_range[t][i]*np.sin(angle[i]/180*np.pi))
  X = np.array([x_o, y_o]).T

  # Let's now decide if this snapshot has 1 or 2 obstacles
  # 2 objects
  Threshold = -20  # to decide if k=1 or k=2,
  cost_list = []
  k_range = range(1, 3)
  for k in k_range:
    centr, clust = kmeans_func(X, k, show_plot=0, ini_method=0)
    # Calculate SSE
    cost = SSE(X, centr, clust)
    cost_list.append(cost)

  # Check best K by evaluating SSE Derivative
  der_list=[]
  der_range=range(1,max(k_range))
  for i in der_range:
    der_list.append( cost_list[i]-cost_list[i-1] )

  # Find lower derivative
  for i in reversed(der_range):
    if der_list[i-1] < Threshold:
      bestK = i+1
      break
    else:
      bestK = 1

  if bestK == 2:
    time.append(t)
    time.append(t)
    X_feature1, X_feature2 = np.zeros([N_INPUTS]), np.zeros([N_INPUTS])
    j1,j2=0,0
    for i in range(len(x_o)):
      if j1 < N_INPUTS:
        if clust[i] == 0:
          X_feature1[j1]=np.sqrt((x_o[i]-px[t])**2 + (y_o[i]-py[t])**2)
          j1=j1+1
      if j2 < N_INPUTS:
        if clust[i] == 1:
          X_feature2[j2]=np.sqrt((x_o[i]-px[t])**2 + (y_o[i]-py[t])**2)
          j2=j2+1
    if len(test_set_X) == 0:
      test_set_X = np.array([X_feature1, X_feature2])
    else:
      test_set_X = np.concatenate((test_set_X,np.array([X_feature1, X_fea
  else:
  # 1 object
    time.append(t)
```

```
    X_feature1 = np.zeros([N_INPUTS])
    j1=0
    for i in range(len(x_o)):
      if j1 < N_INPUTS:
        if clust[i] == 0:
          X_feature1[j1]=np.sqrt((x_o[i]-px[t])**2 + (y_o[i]-py[t])**2)
          j1=j1+1
    if len(test_set_X) == 0:
      test_set_X = np.array([X_feature1])
    else:
      test_set_X = np.concatenate((test_set_X,np.array([X_feature1])))

for i in range(len(test_set_X)):
  classification.append(model.feedforward(test_set_X[i]))

# From probabilities to 0 or 1 (Binary classification)
classification = np.array(classification)
for t in range(len(classification)):
  if ((classification[t] >= 0.5)):
    classification[t] = 1
  if ((classification[t] < 0.5)):
    classification[t] = 0
```

In [ ]:

```
# Results Obtained

# Create the 'Classification' column based on the boolean values
classification_labels = ["Square" if c else "Circle" for c in classificat

# Create a dictionary with the data
data = {
    'Time': time,
    'Classification': classification_labels,
}

# Create a DataFrame
df = pd.DataFrame(data)

# Set 'Time' as the index
df.set_index('Time', inplace=True)

# Round the accuracy values to 2 decimal places
df = df.round(2)

# Display the table (Adjust the range to analyse the results obtained)
# If the time is duplicated means we have 2 objects
df[470:500]
```

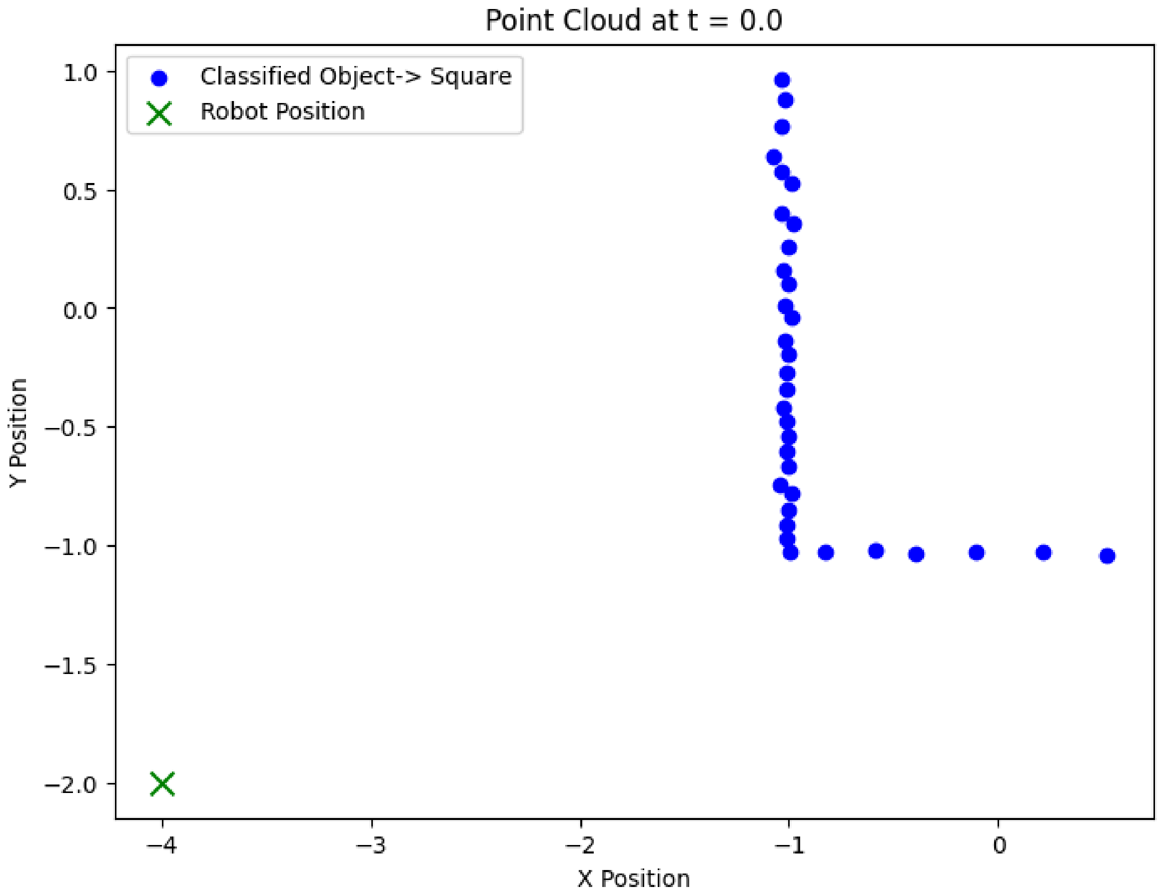
Out[ ]:

	Classification
Time	
319	Circle
320	Square
320	Circle
321	Circle
321	Square
322	Square
322	Circle
323	Square
323	Square
324	Square
324	Circle
325	Square
325	Circle
326	Square
326	Circle
327	Circle
327	Square
328	Circle
328	Square
329	Square
329	Circle
330	Square
330	Circle
331	Circle
331	Square
332	Square
332	Circle
333	Circle
333	Square
334	Circle

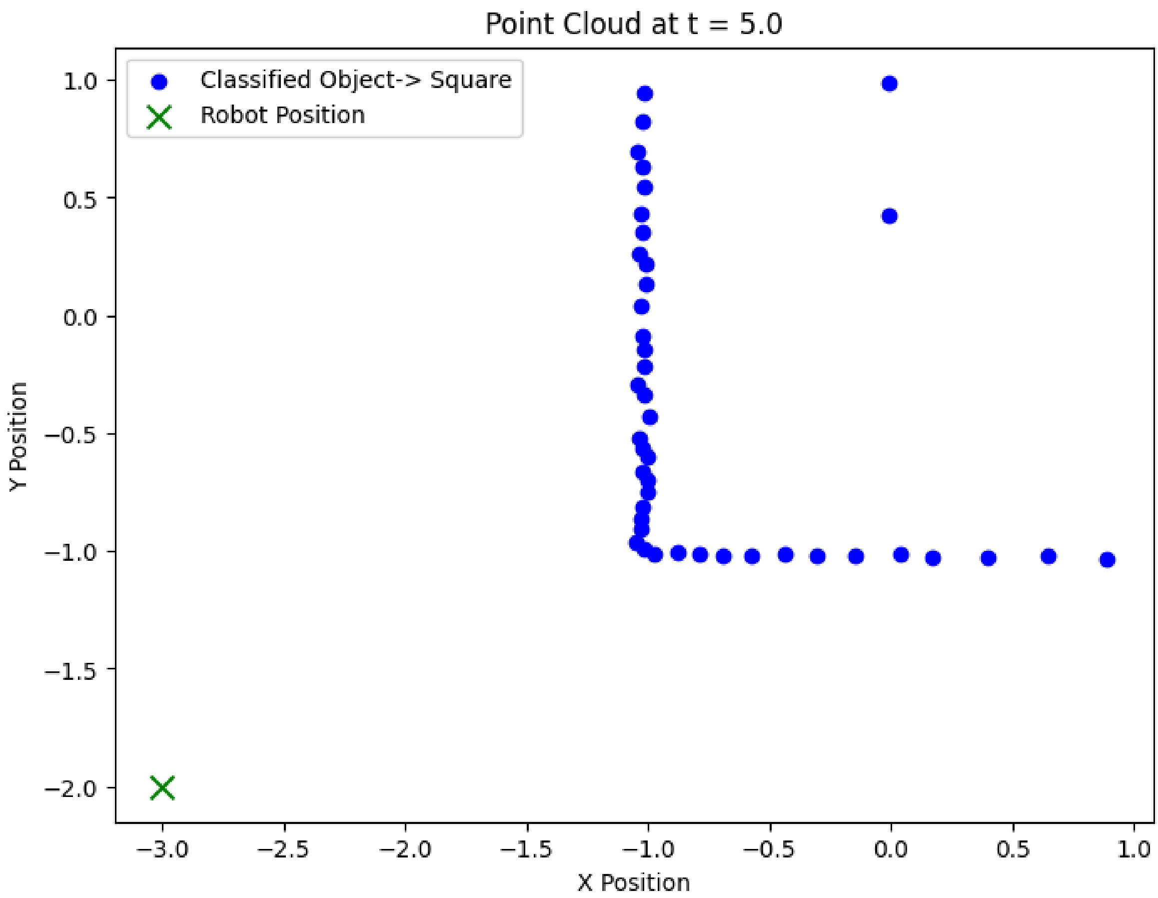
In [ ]:

```
# Plot some results
for t in range(0, 10*50, 50):
  x_o, y_o = [], []
  for i in range(len(Lidar_range[t])):
    if Lidar_range[t][i] > 0:
      x_o.append(px[t]+Lidar_range[t][i]*np.cos(angle[i]/180*np.pi))
      y_o.append(py[t]+Lidar_range[t][i]*np.sin(angle[i]/180*np.pi))
```

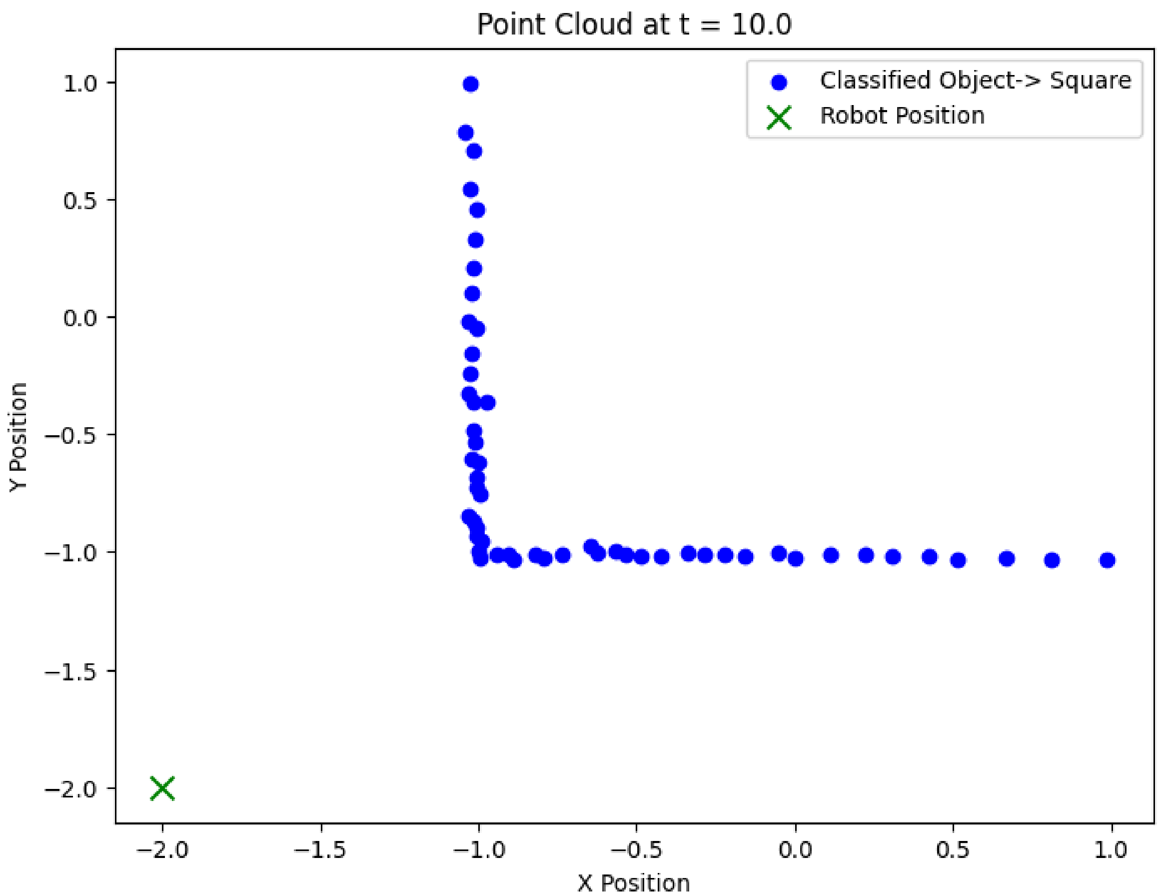
```
X = np.array([x_o, y_o]).T

all_idx = [i for i, x in enumerate(time) if x == t]
num_objects = len(all_idx)
plot_snapshot(t, X, px, py, num_objects, classification[all_idx])
print(classification[all_idx])
print(f"Number of objects found {num_objects}")
```



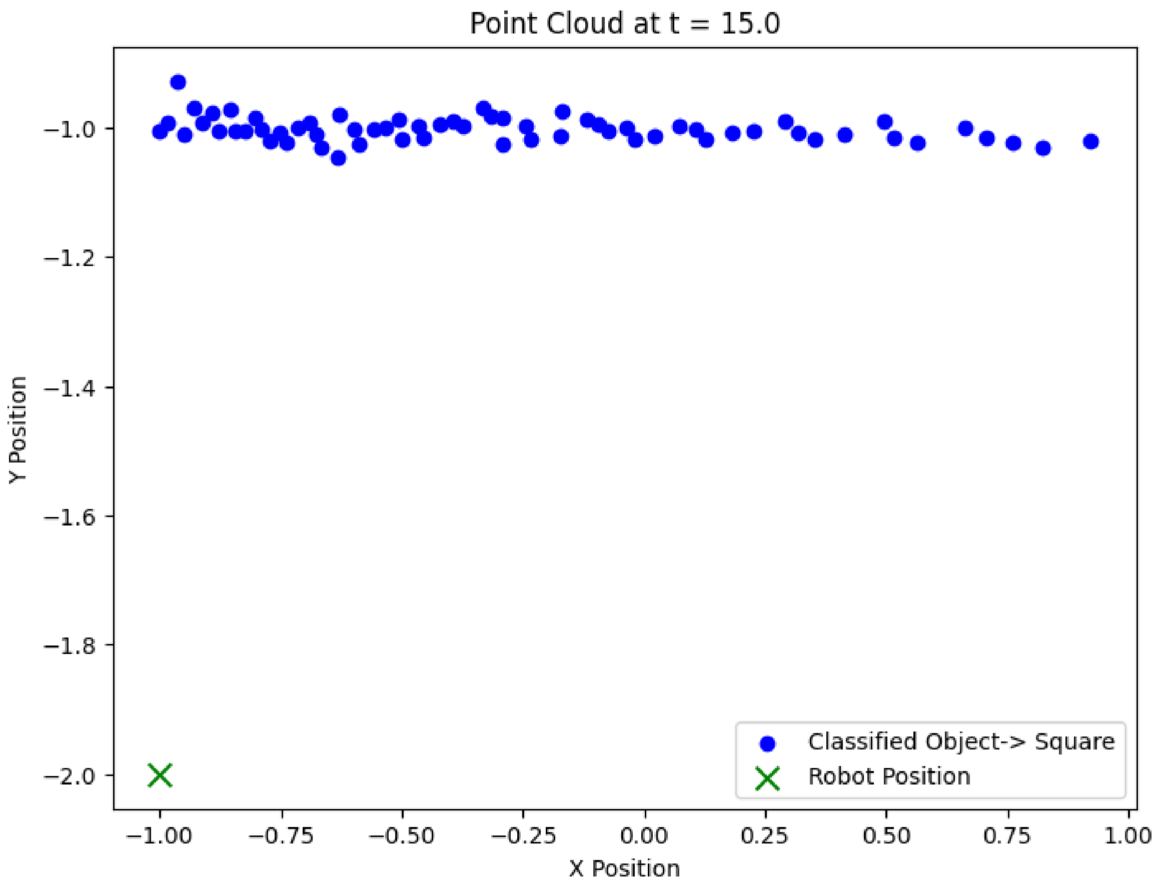

```
[1.]
Number of objects found 1
```



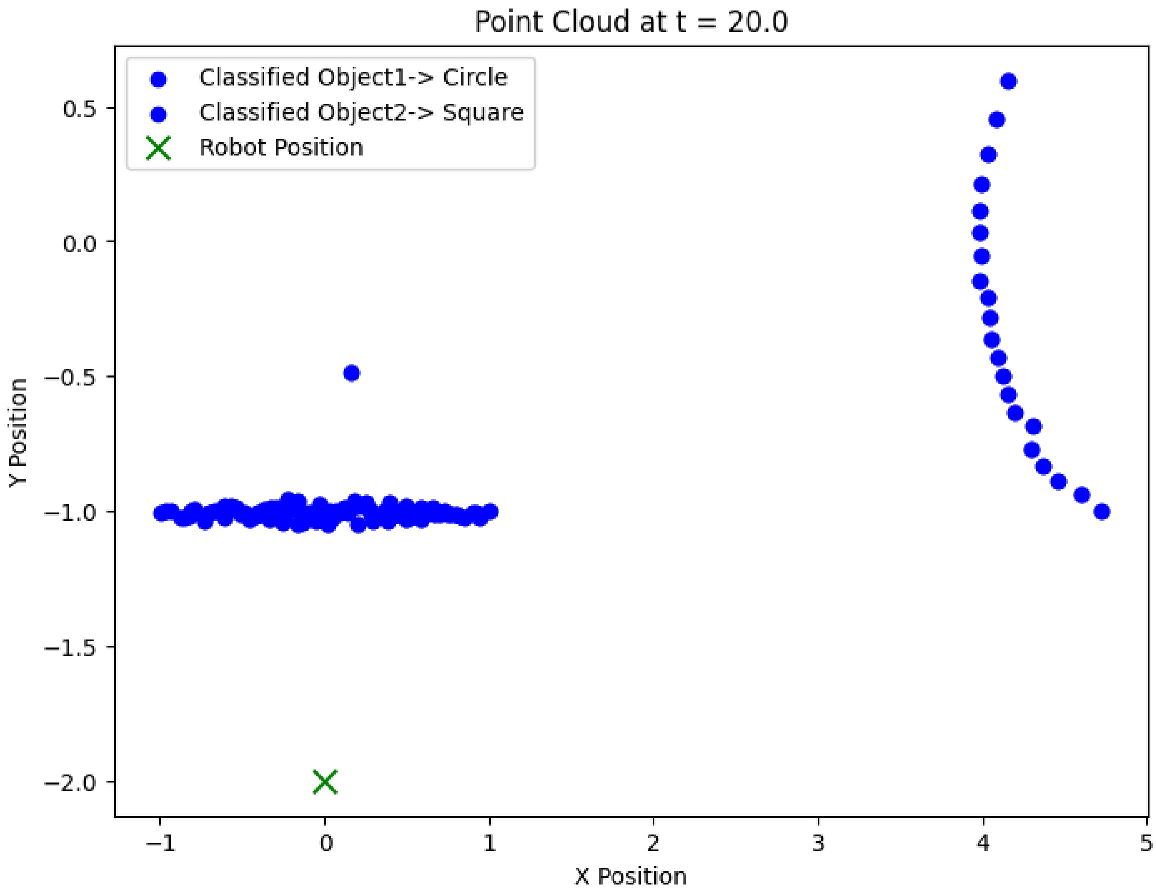

```
[1.]
Number of objects found 1
```



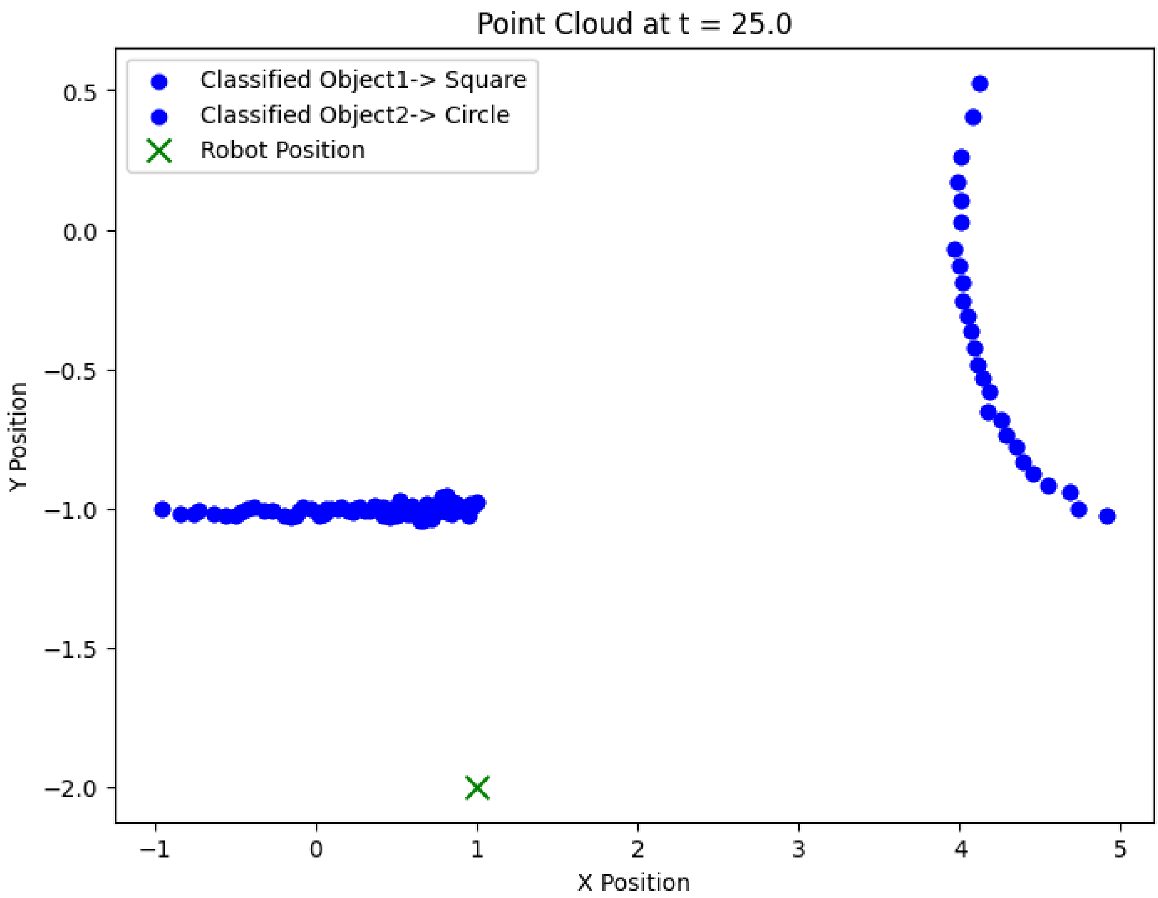

```
[1.]
Number of objects found 1
```



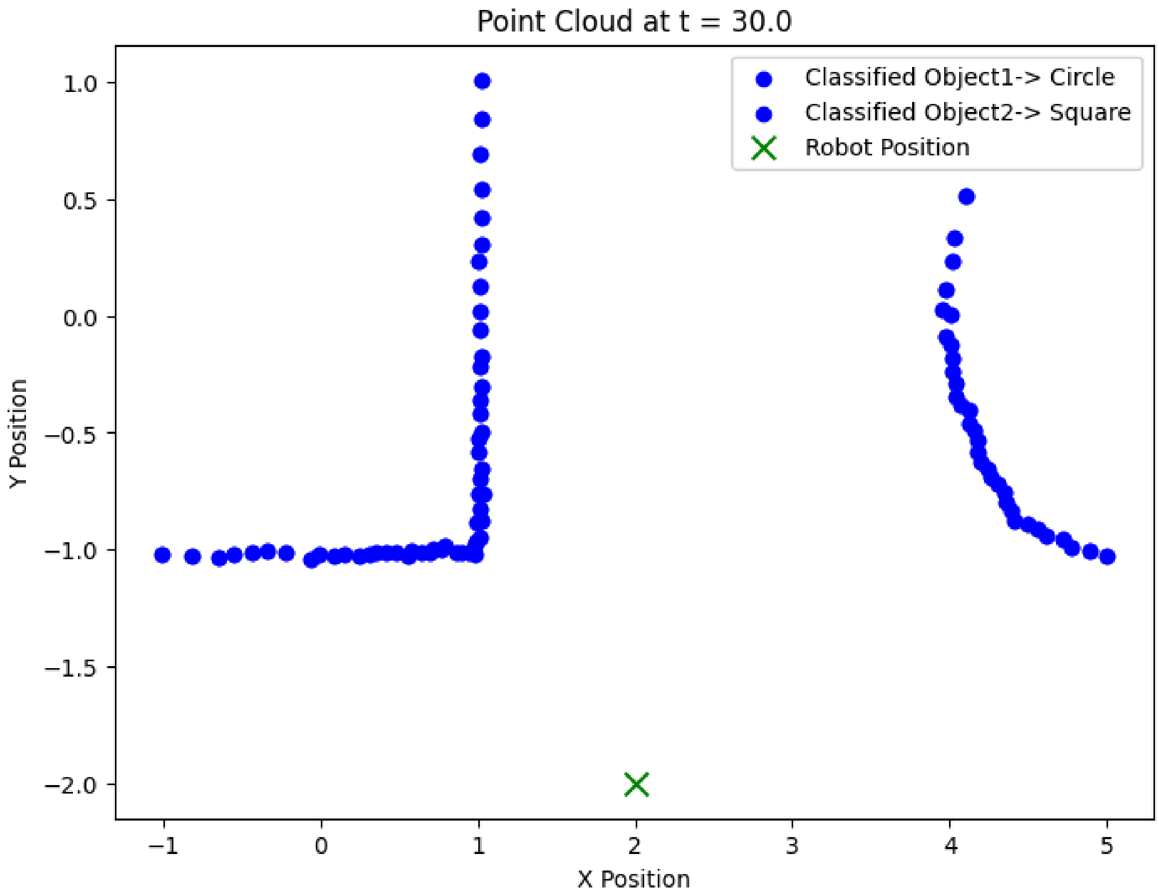

```
[1.]
Number of objects found 1
```



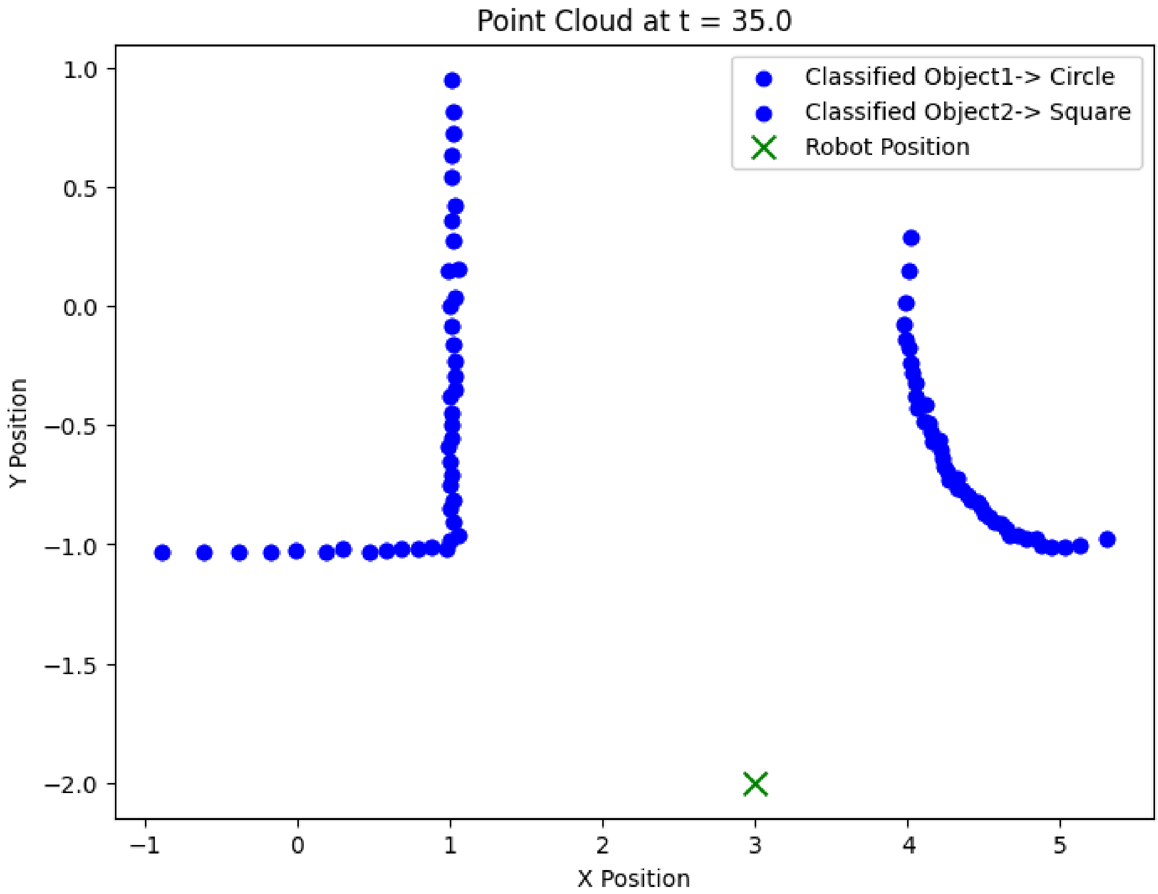

```
[0. 1.]
Number of objects found 2
```



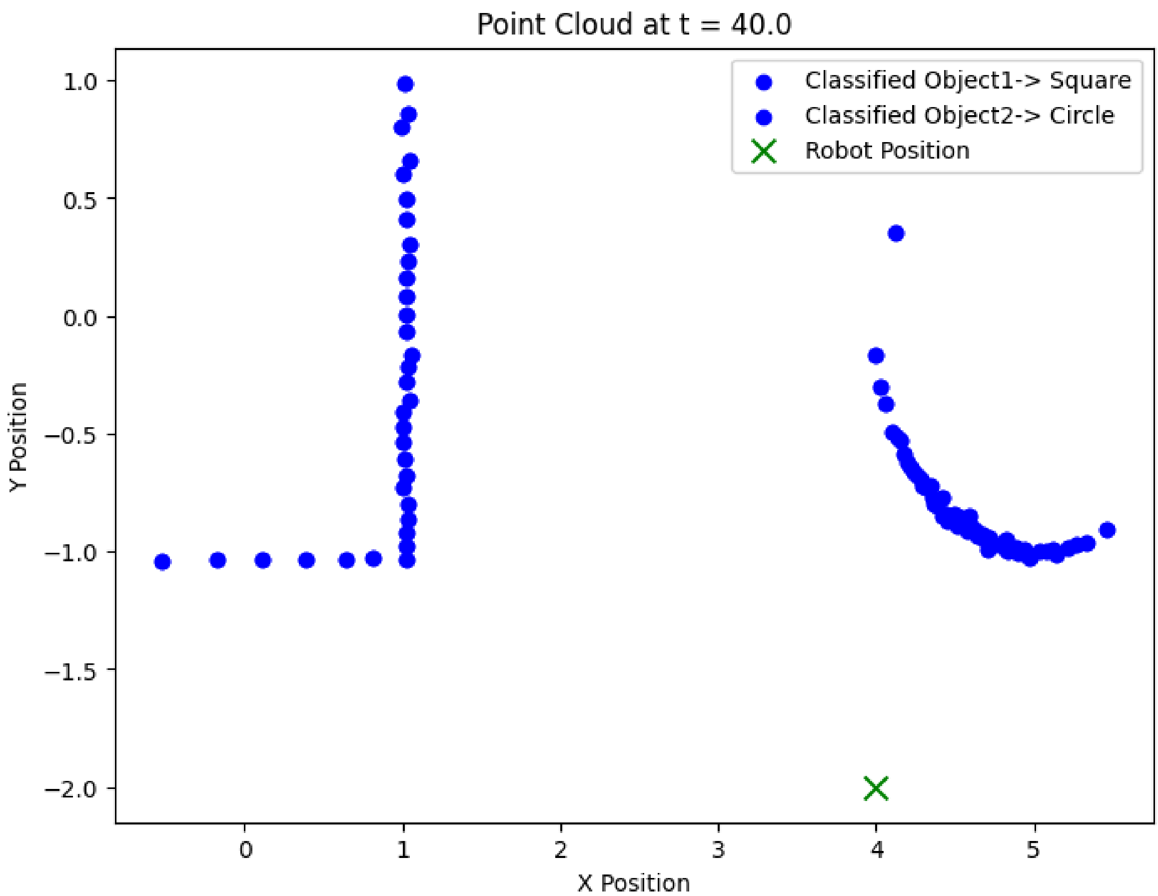

```
[1. 0.]
Number of objects found 2
```



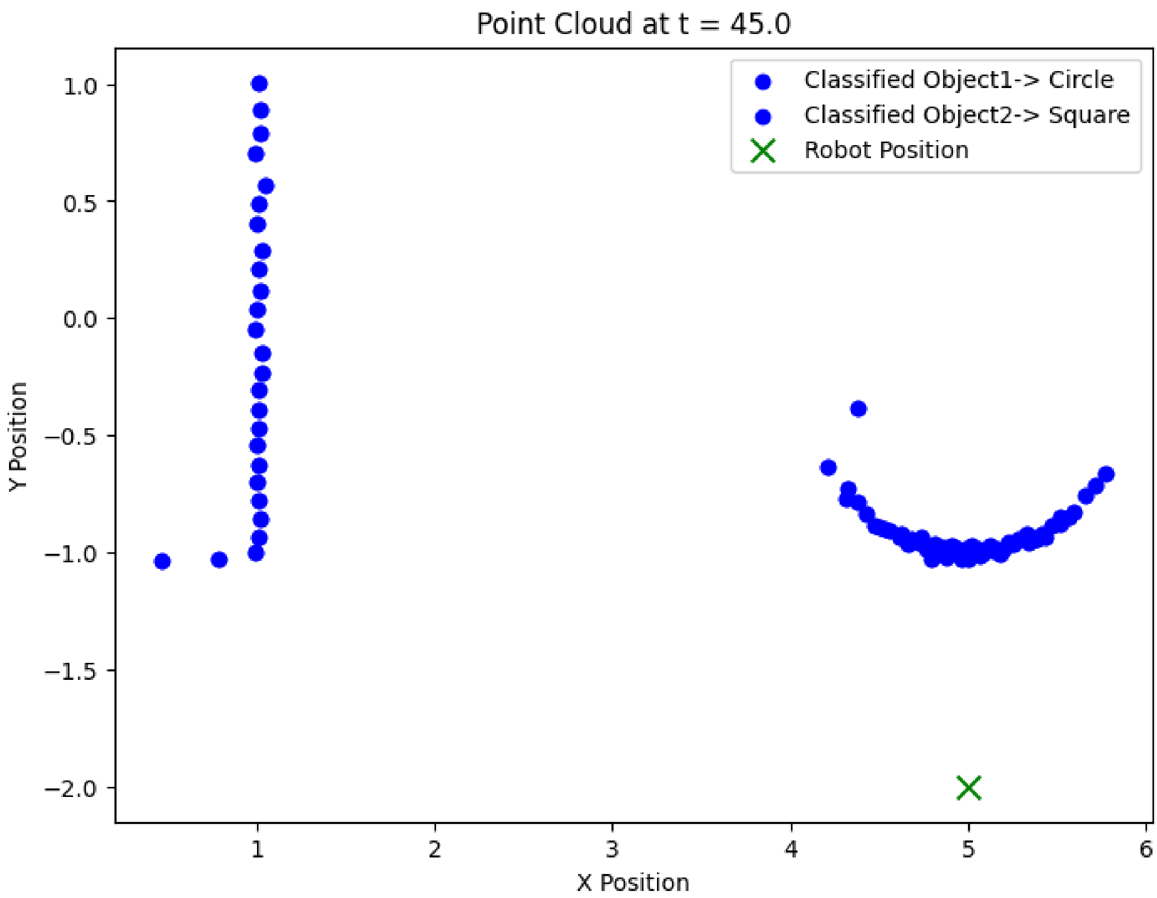

```
[0. 1.]
Number of objects found 2
```




```
[0. 1.]
Number of objects found 2
```




```
[1. 0.]
Number of objects found 2
```




```
[0. 1.]
Number of objects found 2
```

Main Conclusion

K-Means and ANN

The snapshots presented demonstrate the robustness of employing k-means for data clustering and artificial neural networks (ANN) for identifying whether the clustered data represents a circle or a square. This combined approach exhibits remarkable resilience, effectively discerning and categorizing objects with accuracy across different scenarios and datasets. The synergy between k-means clustering and ANN classification showcases a promising methodology for object detection and classification tasks, highlighting its potential for real-world applications requiring robust and adaptable solutions.

2.3 (Extra) Using now PyTorch or other similar package, implement a better ANN (meaning with a better accuracy) and test it.

Note: This question is optional. If you solve it, you get extra 15 points (in 100).

The Shape Classifier Module using PyTorch

In [ ]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
```

```python
import numpy as np
import pandas as pd
```

In [ ]:
```python
class Shape_Classifier_Module(nn.Module):

    def __init__(self, num_inputs, num_hidden, num_outputs):
        super().__init__()
        # Initialize the modules we need to build the network
        self.linear1         = nn.Linear(num_inputs, num_hidden)
        self.activation_f1   = nn.ReLU()
        self.linear2         = nn.Linear(num_hidden , num_hidden)
        self.activation_f2   = nn.ReLU()
        self.linear3         = nn.Linear(num_hidden , num_hidden - 10)
        self.activation_f3   = nn.ReLU()
        self.linear4         = nn.Linear(num_hidden - 10, num_outputs)
        self.activation_sig  = nn.Sigmoid()           #  COMMENT if nn.BC

    def forward(self, x):
        # Perform the calculation of the model to determine the predictio
        x = self.linear1(x)
        x = self.activation_f1(x)
        x = self.linear2(x)
        x = self.activation_f2(x)
        x = self.linear3(x)
        x = self.activation_f3(x)
        x = self.linear4(x)
        x = self.activation_sig(x)                   #  COMMENT if nn.BCE
        return x
```

In [ ]:
```python
N_INPUTS = 10
model = Shape_Classifier_Module(num_inputs=N_INPUTS, num_hidden=15, num_o
# Printing a module shows all its submodules
print(model)
```

```
Shape_Classifier_Module(
  (linear1): Linear(in_features=10, out_features=15, bias=True)
  (activation_f1): ReLU()
  (linear2): Linear(in_features=15, out_features=15, bias=True)
  (activation_f2): ReLU()
  (linear3): Linear(in_features=15, out_features=5, bias=True)
  (activation_f3): ReLU()
  (linear4): Linear(in_features=5, out_features=1, bias=True)
  (activation_sig): Sigmoid()
)
```

In [ ]:
```python
# Specific model parameters
for name, param in model.named_parameters():
    print(f"Parameter {name}, shape {param.shape}")

# All model parameter
#model.state_dict()   #PyTorch assigns random values to these weights and
```

```
Parameter linear1.weight, shape torch.Size([15, 10])
Parameter linear1.bias, shape torch.Size([15])
Parameter linear2.weight, shape torch.Size([15, 15])
Parameter linear2.bias, shape torch.Size([15])
Parameter linear3.weight, shape torch.Size([5, 15])
Parameter linear3.bias, shape torch.Size([5])
Parameter linear4.weight, shape torch.Size([1, 5])
Parameter linear4.bias, shape torch.Size([1])
```

Prepare data

In [ ]:
```
import torch.utils.data as data

SEED = 42
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

In [ ]:
```
# Run cells that load the dataframes

# Training Data
X_train = df_train.iloc[:, np.arange(2,362,1)].values
Y_train = df_train["label"].values
data_train = df_train.iloc[:, np.arange(2,363,1)].values  #it also inclua

# Testing Data
X_test = df_test.iloc[:, np.arange(2,362,1)].values
Y_test = df_test["label"].values

# Between 0 and 1
Y_train = Y_train - 1
Y_test = Y_test - 1

# Build the Trainingset (with the first nonzero N_INPUTS ranges)
trainingset_X = np.zeros([len(Y_train), N_INPUTS])
for t in range(len(Y_train)):
  j=0
  for i in range(360):
    if X_train[t][i] > 0:
      if j < N_INPUTS:
        trainingset_X[t][j] = X_train[t][i]
        j +=1

# Build the Testset (with the first nonzero N_INPUTS ranges)
testset_X = np.zeros([len(Y_test), N_INPUTS])
for t in range(len(Y_test)):
  j=0
  for i in range(360):
    if X_test[t][i] > 0:
      if j < N_INPUTS:
        testset_X[t][j] = X_test[t][i]
        j +=1
```

In [ ]:
```
print(len(trainingset_X))
print(len(testset_X))
```

```
2240
960
```

In [ ]:
```
class ShapeDataset(Dataset):
    def __init__(self, data, labels):
        self.data = torch.tensor(data, dtype=torch.float32)
        self.labels = torch.tensor(labels, dtype=torch.int16)

    def __len__(self):
        return len(self.labels)
```

```
    def __getitem__(self, idx):
        x = self.data[idx]
        y = self.labels[idx]
        return x, y
```

In [ ]:
```
# Create dataset instances
train_dataset = ShapeDataset(trainingset_X, Y_train)
test_dataset = ShapeDataset(testset_X, Y_test)

# Create DataLoader instances
batch_size = 32  # Example value, set this according to your needs
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=T
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=Fal
```

In [ ]:
```
# next(iter(...)) catches the first batch of the data loader
# If shuffle is True, this will return a different batch every time we ru
# For iterating over the whole dataset, we can simple use "for batch in d
data_inputs, data_labels = next(iter(train_loader))

# The shape of the outputs are [batch_size, d_1,...,d_N] where d_1,...,d_
# dimensions of the data point returned from the dataset class
print("Data inputs", data_inputs.shape, "\n", data_inputs)
print("Data labels", data_labels.shape, "\n", data_labels)
```

```
Data inputs torch.Size([32, 10])
 tensor([[2.5559, 2.3538, 2.2307, 2.1920, 2.0859, 2.0850, 2.0426, 1.9988,
1.9858,
         1.9341],
        [4.4423, 4.2429, 4.1084, 4.0230, 3.9771, 3.9437, 3.9176, 5.7930,
3.8191,
         3.8035],
        [4.9255, 4.7673, 4.6048, 4.5244, 4.4841, 4.4674, 4.4185, 4.4084,
4.3188,
         4.3644],
        [4.5768, 4.0766, 3.6260, 3.2956, 3.0443, 3.0449, 3.0808, 4.6165,
3.1102,
         3.1319],
        [3.9776, 3.6727, 3.5683, 3.4706, 3.4014, 3.3420, 3.3344, 3.2622,
3.2743,
         3.1980],
        [4.9203, 4.8188, 4.7566, 4.6099, 4.5437, 4.4309, 4.3185, 4.2677,
4.2629,
         4.1449],
        [1.9963, 1.9000, 1.8220, 1.7850, 1.7457, 1.6944, 1.6323, 1.6098,
1.6421,
         1.6091],
        [4.2168, 3.9047, 3.8007, 3.6114, 3.6094, 3.5353, 3.5260, 3.4617,
3.4324,
         3.4056],
        [2.7934, 2.6065, 2.5550, 2.4500, 2.3996, 2.3780, 2.2988, 2.2976,
2.2490,
         2.2596],
        [4.5315, 4.4558, 4.3822, 4.2781, 4.1781, 4.1575, 4.0887, 4.0326,
3.9369,
         3.8837],
        [3.8051, 3.7752, 3.7253, 3.6743, 3.6208, 3.6105, 3.5641, 3.5260,
3.4947,
         3.4680],
        [1.5460, 1.4362, 1.3878, 1.3111, 1.2842, 1.2261, 1.2263, 1.1657,
1.1542,
         1.1309],
        [4.5724, 3.2704, 2.8903, 2.9113, 2.9127, 2.9465, 2.9443, 2.9652,
2.9617,
         2.9694],
        [1.0319, 1.0870, 1.0611, 1.0345, 1.0622, 0.9859, 1.0459, 1.0516,
1.0168,
         0.9781],
        [3.9415, 3.5972, 3.3494, 3.0950, 2.9277, 2.6944, 2.5442, 2.3909,
2.3073,
         2.1362],
        [0.9869, 0.9373, 0.8991, 0.8736, 0.8041, 0.8068, 0.7723, 0.7303,
0.7508,
         0.7266],
        [3.8639, 3.7319, 3.6312, 3.5023, 3.5036, 5.1244, 3.3364, 3.3354,
3.3229,
         3.3094],
        [2.0607, 2.0230, 2.0175, 2.0340, 2.0547, 2.0517, 2.0560, 1.9934,
2.0103,
         1.9878],
        [2.3728, 2.2758, 2.1923, 2.1536, 2.0480, 1.9912, 1.9686, 1.8635,
1.8105,
         1.7870],
        [1.2759, 1.2585, 1.2517, 1.2492, 1.2293, 1.1592, 1.1828, 1.2025,
1.1885,
```

```
         1.1405],
        [4.2039, 4.0522, 3.8690, 3.7476, 3.5794, 3.4383, 3.3614, 3.2268,
3.0942,
         2.9916],
        [4.8524, 4.7889, 4.6307, 4.5830, 4.4623, 4.3805, 4.3136, 4.2382,
4.1299,
         4.0635],
        [3.6254, 3.4553, 3.3479, 3.3079, 3.2679, 3.1947, 3.1913, 3.1204,
3.0813,
         3.0426],
        [2.3186, 2.0152, 2.0072, 2.0019, 2.0113, 2.0160, 2.0115, 2.0461,
2.0249,
         2.0244],
        [4.3744, 4.1844, 4.0652, 3.9340, 3.8185, 3.7339, 3.6121, 3.5690,
3.4447,
         3.3751],
        [4.4775, 4.3491, 4.2407, 4.1282, 3.9834, 3.8559, 3.7527, 3.6676,
3.6211,
         3.4679],
        [1.0183, 1.0217, 1.0588, 1.0085, 1.0143, 1.0387, 1.0504, 1.0226,
0.9914,
         0.9839],
        [1.7621, 1.6878, 1.6068, 1.5408, 1.5135, 1.4651, 1.4420, 1.4234,
1.4333,
         1.3483],
        [1.0294, 0.9289, 0.8977, 0.8444, 0.8106, 0.7602, 0.7226, 0.7169,
0.6686,
         0.6907],
        [3.8145, 3.5158, 3.4138, 3.3621, 3.2784, 3.2538, 3.2204, 3.1615,
3.1324,
         3.0817],
        [4.2755, 4.2070, 4.1224, 4.0461, 3.9762, 3.9696, 3.8832, 3.8305,
3.7866,
         3.8070],
        [4.0443, 3.8226, 3.7476, 3.6387, 3.6156, 3.5522, 3.4680, 3.4796,
3.4449,
         3.4341]])
Data labels torch.Size([32])
 tensor([0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1,
0, 1,
        1, 1, 1, 0, 0, 0, 1, 0], dtype=torch.int16)
```

Train Model

In [ ]:
```
#loss_module =  nn.BCELoss() # or nn.BCEWithLogitsLoss()  nn.BCELoss()
#loss_module = nn.BCEWithLogitsLoss()
loss_module = nn.MSELoss()
```

In [ ]:
```
# Input to the optimizer are the parameters of the model: model.parameter
# Example usage with SGD
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

# Example usage with Adam optimizer
# optimizer = torch.optim.Adam(model.parameters(), lr=0.1)
```

In [ ]:
```
# Push model to device. Has to be only done once
# Define your execution device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
print("The model will be running on", device, "device")

# Convert model parameters and buffers to CPU or CUDA
model.to(device)
```

```
The model will be running on cpu device
```

Out[ ]:
```
Shape_Classifier_Module(
  (linear1): Linear(in_features=10, out_features=15, bias=True)
  (activation_f1): ReLU()
  (linear2): Linear(in_features=15, out_features=15, bias=True)
  (activation_f2): ReLU()
  (linear3): Linear(in_features=15, out_features=5, bias=True)
  (activation_f3): ReLU()
  (linear4): Linear(in_features=5, out_features=1, bias=True)
  (activation_sig): Sigmoid()
)
```

In [ ]:
```
from tqdm.notebook import trange, tqdm

def train_model(model, optimizer, dataloader, loss_criteria, num_epochs=1
    # Set model to train mode
    model.train()

    # Training loop
    for epoch in tqdm(range(num_epochs)):
        epoch_loss = 0.0
        for data_inputs, data_labels in dataloader:

            ## Step 0 (needed in case of GPU): Move input data to device
            data_inputs = data_inputs.to(device)
            data_labels = data_labels.to(device)

            # Step 1: setting gradients to zero. The gradients would not
            optimizer.zero_grad()

            ## Step 2: Run the model on the input data
            preds = model(data_inputs)
            preds = preds.squeeze(dim=1) # Output is [Batch size, 1], but
            #preds = torch.sigmoid(preds)  # UNCOMMNET if nn.BCEWithLogit

            ## Step 3: Calculate the loss
            loss = loss_criteria(preds, data_labels.float())

            ## Step 4: Perform backpropagation
            loss.backward()

            ## Step 5: Update parameters based on the calculated gradient
            optimizer.step()

            ## Step 6: Take the running average of the loss
            epoch_loss += loss.item()

        # Add average loss to TensorBoard
        epoch_loss /= len(dataloader)

        if epoch % 5 == 0:
            print('[%d] loss: %.3f' % (epoch + 1, epoch_loss,))
```

In [ ]:
```
train_model(model, optimizer, train_loader, loss_module)
```

```
0%|          | 0/1500 [00:00<?, ?it/s]
```

```
[1] loss: 0.250
[6] loss: 0.250
[11] loss: 0.249
[16] loss: 0.248
[21] loss: 0.245
[26] loss: 0.227
[31] loss: 0.219
[36] loss: 0.218
[41] loss: 0.215
[46] loss: 0.203
[51] loss: 0.200
[56] loss: 0.198
[61] loss: 0.200
[66] loss: 0.199
[71] loss: 0.195
[76] loss: 0.176
[81] loss: 0.180
[86] loss: 0.165
[91] loss: 0.170
[96] loss: 0.160
[101] loss: 0.162
[106] loss: 0.155
[111] loss: 0.143
[116] loss: 0.143
[121] loss: 0.148
[126] loss: 0.147
[131] loss: 0.133
[136] loss: 0.142
[141] loss: 0.158
[146] loss: 0.146
[151] loss: 0.137
[156] loss: 0.130
[161] loss: 0.139
[166] loss: 0.135
[171] loss: 0.134
[176] loss: 0.130
[181] loss: 0.122
[186] loss: 0.123
[191] loss: 0.123
[196] loss: 0.128
[201] loss: 0.122
[206] loss: 0.126
[211] loss: 0.130
[216] loss: 0.116
[221] loss: 0.120
[226] loss: 0.128
[231] loss: 0.122
[236] loss: 0.116
[241] loss: 0.126
[246] loss: 0.121
[251] loss: 0.121
[256] loss: 0.121
[261] loss: 0.115
[266] loss: 0.112
[271] loss: 0.109
[276] loss: 0.112
[281] loss: 0.119
[286] loss: 0.102
[291] loss: 0.114
[296] loss: 0.118
```

```
[301] loss: 0.110
[306] loss: 0.108
[311] loss: 0.108
[316] loss: 0.109
[321] loss: 0.107
[326] loss: 0.111
[331] loss: 0.098
[336] loss: 0.101
[341] loss: 0.099
[346] loss: 0.096
[351] loss: 0.109
[356] loss: 0.096
[361] loss: 0.091
[366] loss: 0.090
[371] loss: 0.090
[376] loss: 0.096
[381] loss: 0.087
[386] loss: 0.087
[391] loss: 0.092
[396] loss: 0.083
[401] loss: 0.086
[406] loss: 0.079
[411] loss: 0.074
[416] loss: 0.084
[421] loss: 0.077
[426] loss: 0.075
[431] loss: 0.105
[436] loss: 0.078
[441] loss: 0.085
[446] loss: 0.077
[451] loss: 0.073
[456] loss: 0.104
[461] loss: 0.089
[466] loss: 0.085
[471] loss: 0.072
[476] loss: 0.071
[481] loss: 0.072
[486] loss: 0.059
[491] loss: 0.082
[496] loss: 0.128
[501] loss: 0.119
[506] loss: 0.101
[511] loss: 0.106
[516] loss: 0.081
[521] loss: 0.077
[526] loss: 0.086
[531] loss: 0.080
[536] loss: 0.077
[541] loss: 0.075
[546] loss: 0.077
[551] loss: 0.054
[556] loss: 0.075
[561] loss: 0.065
[566] loss: 0.053
[571] loss: 0.053
[576] loss: 0.066
[581] loss: 0.067
[586] loss: 0.057
[591] loss: 0.069
[596] loss: 0.061
```

```
[601] loss: 0.079
[606] loss: 0.072
[611] loss: 0.049
[616] loss: 0.065
[621] loss: 0.059
[626] loss: 0.062
[631] loss: 0.090
[636] loss: 0.057
[641] loss: 0.058
[646] loss: 0.063
[651] loss: 0.080
[656] loss: 0.051
[661] loss: 0.055
[666] loss: 0.053
[671] loss: 0.049
[676] loss: 0.068
[681] loss: 0.059
[686] loss: 0.094
[691] loss: 0.093
[696] loss: 0.039
[701] loss: 0.043
[706] loss: 0.047
[711] loss: 0.034
[716] loss: 0.070
[721] loss: 0.028
[726] loss: 0.059
[731] loss: 0.079
[736] loss: 0.044
[741] loss: 0.051
[746] loss: 0.058
[751] loss: 0.059
[756] loss: 0.033
[761] loss: 0.054
[766] loss: 0.041
[771] loss: 0.041
[776] loss: 0.051
[781] loss: 0.069
[786] loss: 0.048
[791] loss: 0.053
[796] loss: 0.055
[801] loss: 0.037
[806] loss: 0.044
[811] loss: 0.035
[816] loss: 0.029
[821] loss: 0.023
[826] loss: 0.034
[831] loss: 0.023
[836] loss: 0.039
[841] loss: 0.084
[846] loss: 0.139
[851] loss: 0.027
[856] loss: 0.031
[861] loss: 0.047
[866] loss: 0.063
[871] loss: 0.027
[876] loss: 0.030
[881] loss: 0.029
[886] loss: 0.036
[891] loss: 0.026
[896] loss: 0.036
```

```
[901] loss: 0.043
[906] loss: 0.022
[911] loss: 0.029
[916] loss: 0.038
[921] loss: 0.024
[926] loss: 0.045
[931] loss: 0.034
[936] loss: 0.072
[941] loss: 0.040
[946] loss: 0.029
[951] loss: 0.046
[956] loss: 0.036
[961] loss: 0.040
[966] loss: 0.028
[971] loss: 0.044
[976] loss: 0.025
[981] loss: 0.028
[986] loss: 0.022
[991] loss: 0.043
[996] loss: 0.031
[1001] loss: 0.022
[1006] loss: 0.045
[1011] loss: 0.033
[1016] loss: 0.022
[1021] loss: 0.022
[1026] loss: 0.019
[1031] loss: 0.039
[1036] loss: 0.023
[1041] loss: 0.031
[1046] loss: 0.022
[1051] loss: 0.030
[1056] loss: 0.054
[1061] loss: 0.020
[1066] loss: 0.041
[1071] loss: 0.047
[1076] loss: 0.041
[1081] loss: 0.028
[1086] loss: 0.020
[1091] loss: 0.022
[1096] loss: 0.015
[1101] loss: 0.034
[1106] loss: 0.043
[1111] loss: 0.035
[1116] loss: 0.021
[1121] loss: 0.033
[1126] loss: 0.026
[1131] loss: 0.027
[1136] loss: 0.022
[1141] loss: 0.023
[1146] loss: 0.030
[1151] loss: 0.023
[1156] loss: 0.022
[1161] loss: 0.030
[1166] loss: 0.021
[1171] loss: 0.028
[1176] loss: 0.027
[1181] loss: 0.020
[1186] loss: 0.017
[1191] loss: 0.017
[1196] loss: 0.028
```

```
[1201] loss: 0.016
[1206] loss: 0.036
[1211] loss: 0.028
[1216] loss: 0.021
[1221] loss: 0.029
[1226] loss: 0.022
[1231] loss: 0.019
[1236] loss: 0.034
[1241] loss: 0.030
[1246] loss: 0.017
[1251] loss: 0.021
[1256] loss: 0.023
[1261] loss: 0.024
[1266] loss: 0.022
[1271] loss: 0.030
[1276] loss: 0.023
[1281] loss: 0.019
[1286] loss: 0.037
[1291] loss: 0.017
[1296] loss: 0.024
[1301] loss: 0.019
[1306] loss: 0.016
[1311] loss: 0.035
[1316] loss: 0.024
[1321] loss: 0.020
[1326] loss: 0.026
[1331] loss: 0.018
[1336] loss: 0.027
[1341] loss: 0.043
[1346] loss: 0.028
[1351] loss: 0.051
[1356] loss: 0.023
[1361] loss: 0.016
[1366] loss: 0.095
[1371] loss: 0.027
[1376] loss: 0.018
[1381] loss: 0.019
[1386] loss: 0.020
[1391] loss: 0.021
[1396] loss: 0.028
[1401] loss: 0.035
[1406] loss: 0.026
[1411] loss: 0.033
[1416] loss: 0.040
[1421] loss: 0.023
[1426] loss: 0.020
[1431] loss: 0.023
[1436] loss: 0.107
[1441] loss: 0.048
[1446] loss: 0.046
[1451] loss: 0.018
[1456] loss: 0.016
[1461] loss: 0.015
[1466] loss: 0.017
[1471] loss: 0.022
[1476] loss: 0.029
[1481] loss: 0.016
[1486] loss: 0.026
[1491] loss: 0.016
[1496] loss: 0.019
```

Test the model

In [ ]:
```
def eval_model(model, data_loader):
    # Set model to eval mode
    model.eval()

    true_preds, num_preds = 0., 0.

    # Deactivate gradients for the following code
    with torch.no_grad():

        # get batch of images from the test DataLoader
        for data_inputs, data_labels in data_loader:
            ## Step 0 (needed in case of GPU): Move input data to device
            data_inputs, data_labels = data_inputs.to(device), data_label

            # Step 1: determine prediction of model
            preds = model(data_inputs)
            preds = preds.squeeze(dim=1)  # Output is [Batch size, 1], bu
            #preds = torch.sigmoid(preds)  # UNCOMMNET if nn.BCEWithLogit

            # Step 2: Binarize predictions to 0 and 1
            pred_labels = (preds >= 0.5).long()

            # Step 3: Keep records of predictions for the accuracy metric
            true_preds += (pred_labels == data_labels).sum()
            num_preds += data_labels.shape[0]

    acc = true_preds / num_preds
    print("Number of misclassified samples in the data: ", int(num_preds)
    print(f"Accuracy of the model: {100.0*acc:4.2f}%")
```

In [ ]:
```
# Evaluate Training Dataset
print("\n----------------------------Train--------------------------")
eval_model(model, train_loader)

print("\n----------------------------Test---------------------------")
# Evaluate Testing Dataset
eval_model(model, test_loader)
```

```
----------------------------Train-------------------------
Number of misclassified samples in the data:  37 in 2240
Accuracy of the model: 98.35%

----------------------------Test--------------------------
Number of misclassified samples in the data:  27 in 960
Accuracy of the model: 97.19%
```