Project 2:

Collaborators:

Talia Seada – 211551601, assigned to the morning group.

Yehudit Brickner – 328601018, assigned to the morning group.

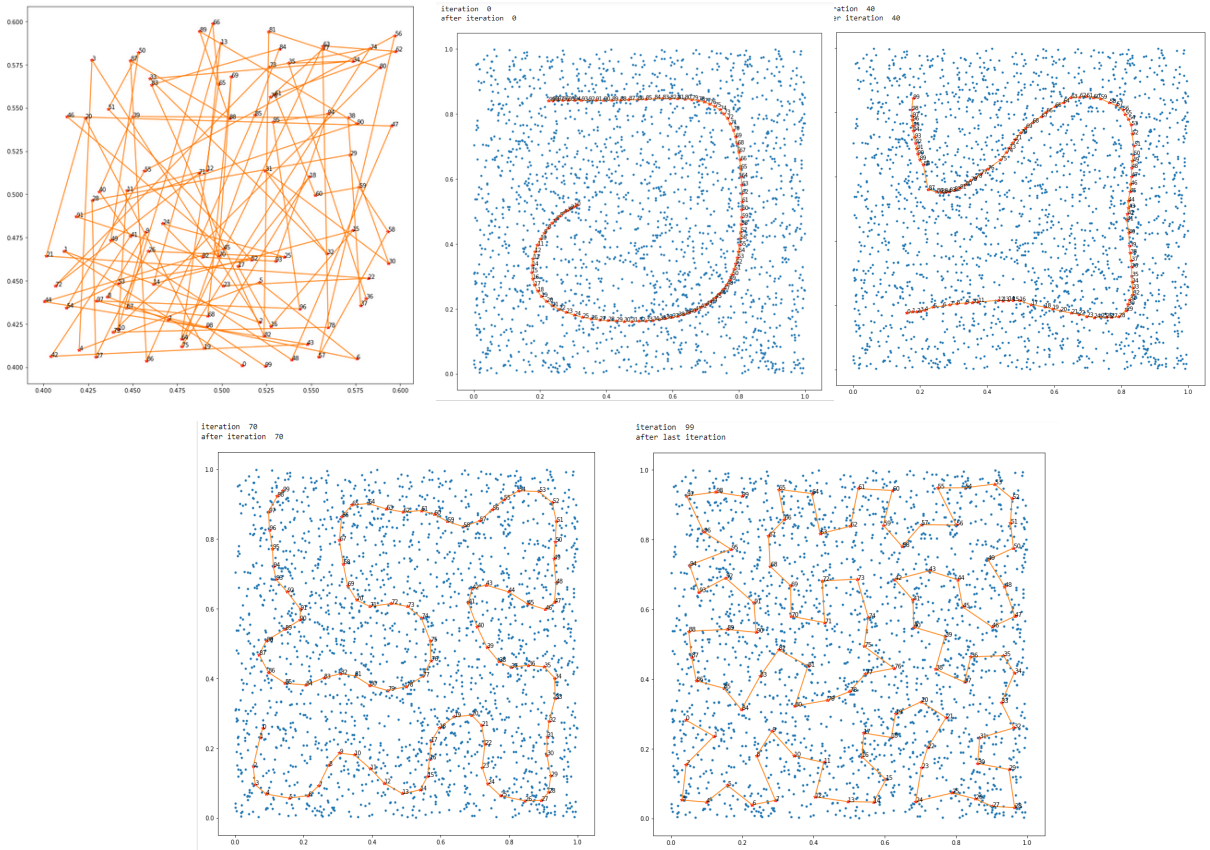
Table of Contents:

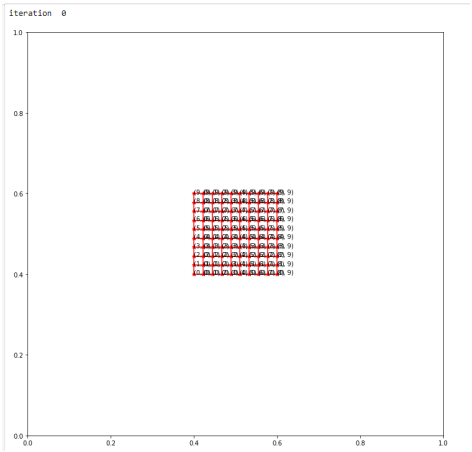
Part A:
Discussions and Illustrations

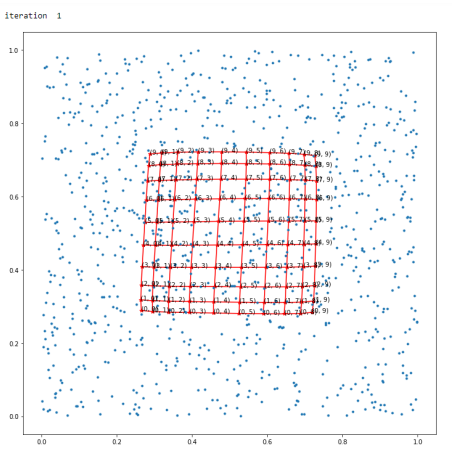
For Part A of this assignment, we implemented the Kohonen algorithm and used it to fit a set of 100 neurons in a topology of a line to a disk.
Then we did the same when the topology of the 100 neurons is arranged in a two dimensional array of 10 x 10.

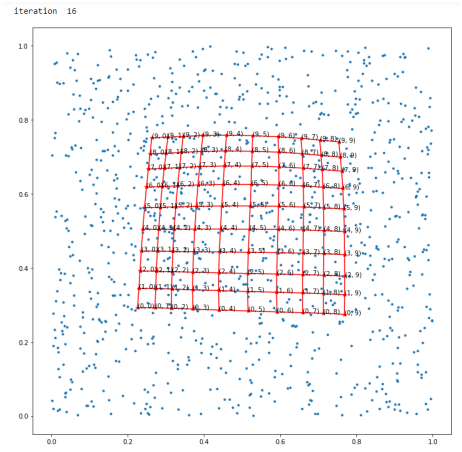
For each implementation we ran it on 3 different data sets of points, with different distributions.
The first data set has data that is uniformly distributed over the range of 0 to 1 (0<=x<=1, 0<=y<=1).

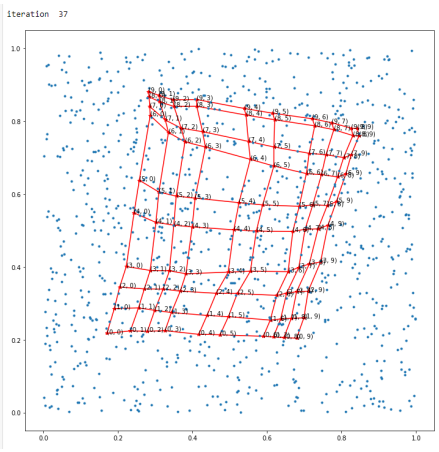
As the number of iterations of the algorithm increased we noticed that both the grid and the line spreaded over the space to match the areas where there are more sampled points.
When the algorithm finishes running we can see that the line is very twisty because it is trying to cluster the data the best way, with each neuron having close to the same amount of data points in its cluster.




iteration 0


iteration 1


iteration 16


iteration 37

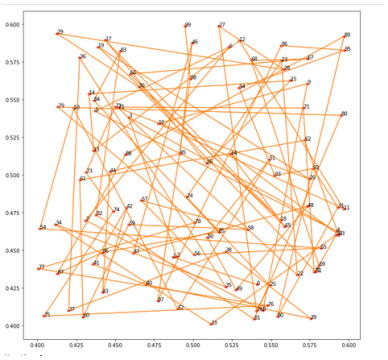
We did the same with at least two non-uniform distributions on the disk:
The second data set has data that is in the corners of the range of 0 to 1,
(0<=x<=0.25, (0.5^0.5)<=x<=1 , 0<=y<=0.25, (0.5^0.5)<=y<=1)

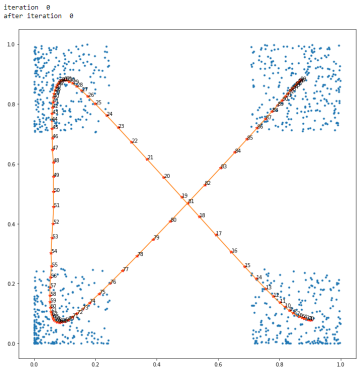
The third data set has data between 0 and 1 where the x and y values are cubed.

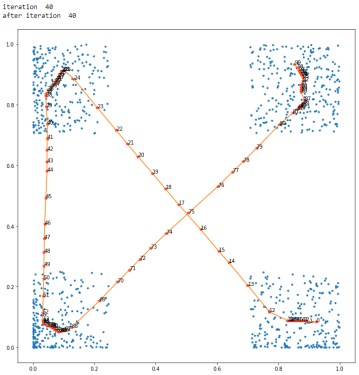
Lastly, we did the same experiments as above for fitting a circle of neurons on a "doughnut" shape.
We also ran the topology of a line where the first and last neuron are connected for a data set of a doughnut with values in the range 2 <= x^2 + y^2 <= 4. This self-organizing map has 30 neurons.

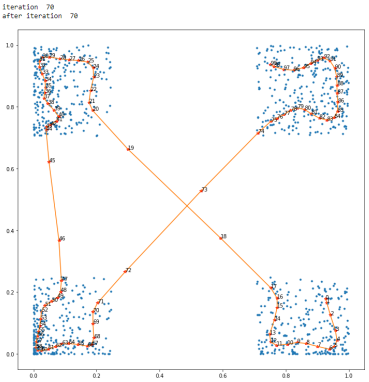
As the number of iterations of the algorithm increased we noticed that the neurons spread out to better fit the data, but we can see some neurons have no cluster of points around them because those neurons are being pulled by 2 parts of the graph, so the neuron gets stuck in the middle. We can also see that in the areas with more point sampled we get a twisty line and where the data is parse its less twisty.

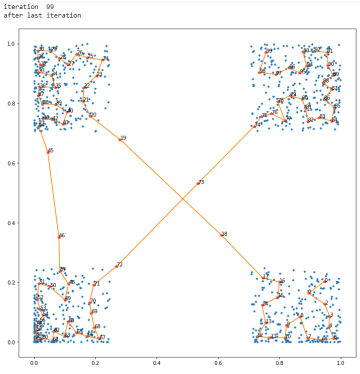
We also noticed that the map of 10 x 10 needed less iterations than the line.

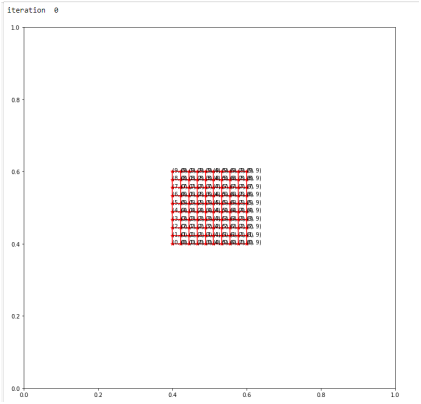


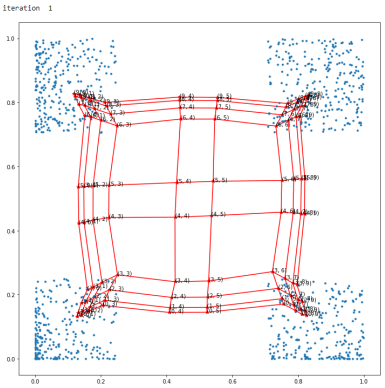

iteration 0
after iteration 0

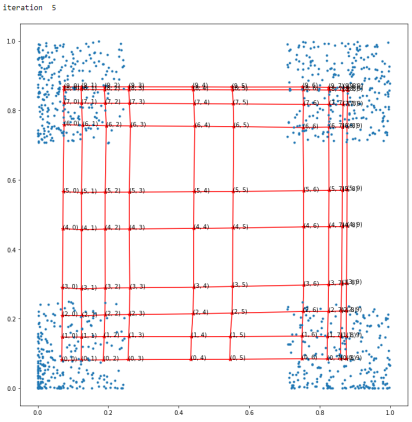

iteration 40
after iteration 40

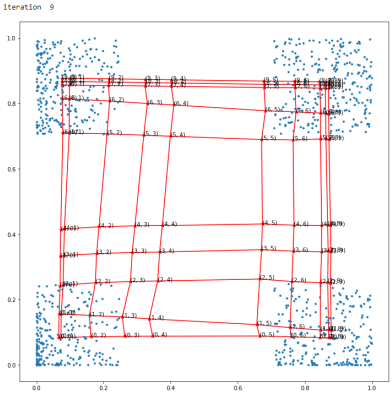

iteration 70
after iteration 70

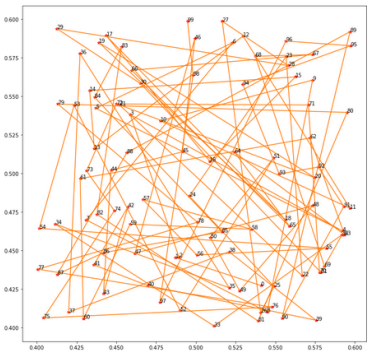

iteration 99
after last iteration

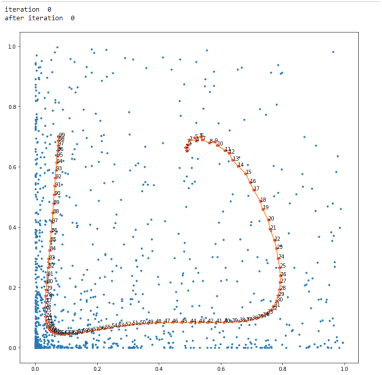

iteration 0

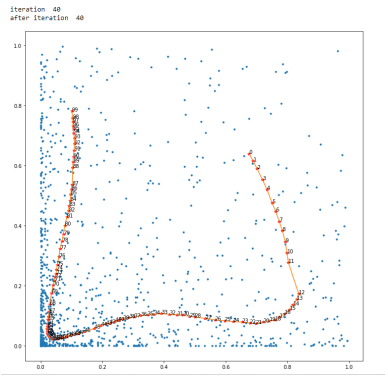

iteration 1

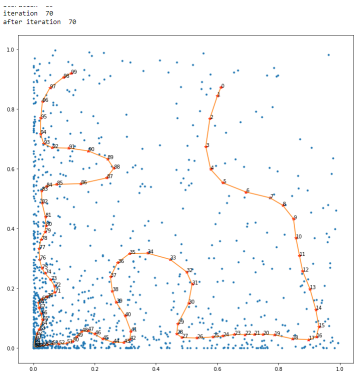

iteration 5

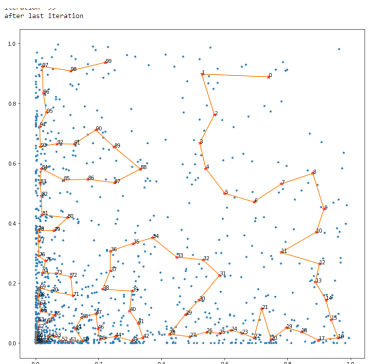

iteration 9

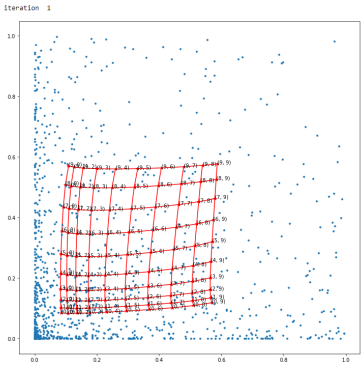


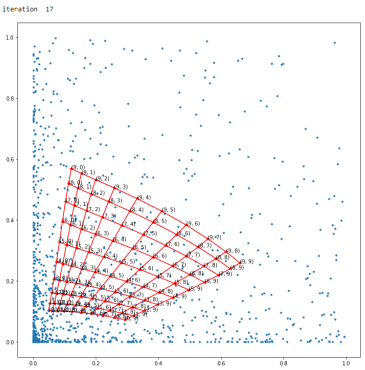

iteration 0
after iteration 0

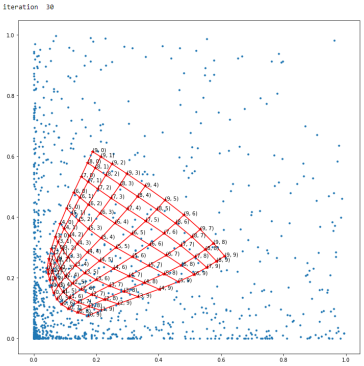

iteration 40
after iteration 40

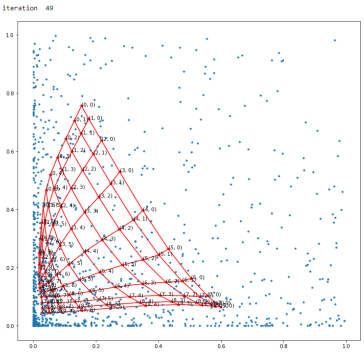

iteration 70
after iteration 70

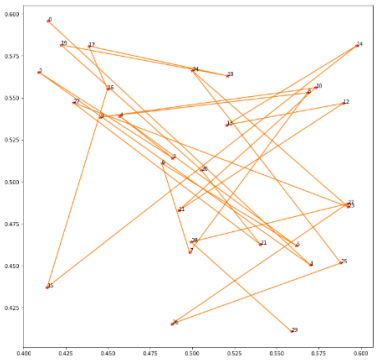

after last iteration

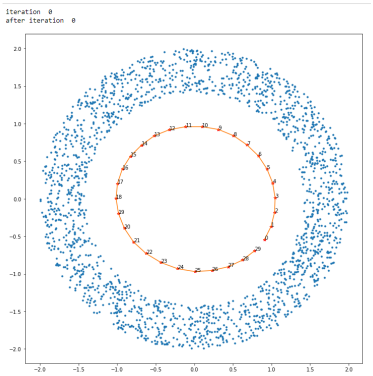

iteration 1

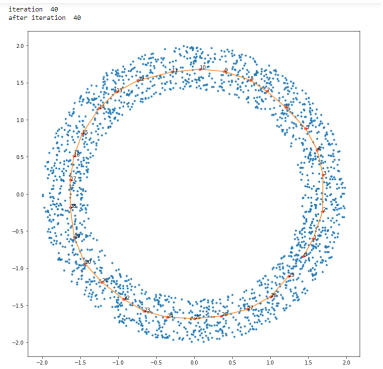

iteration 17

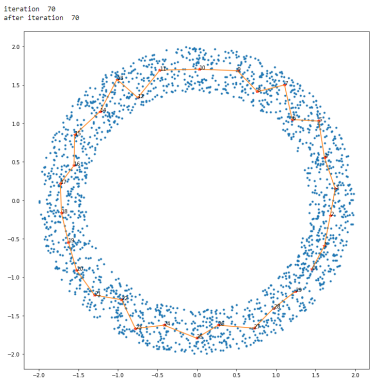

iteration 30

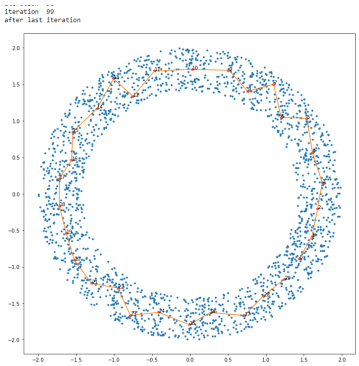

iteration 49

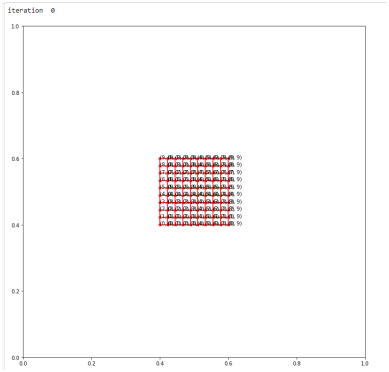


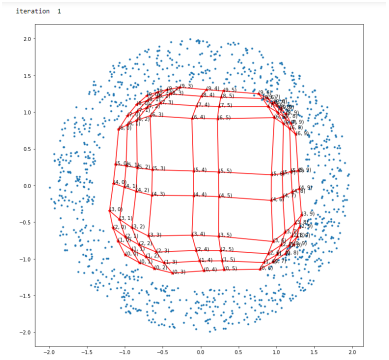

iteration 0
after iteration 0

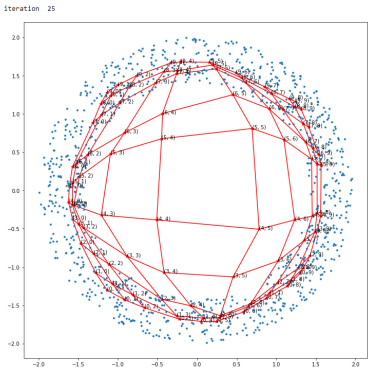

iteration 40
after iteration 40

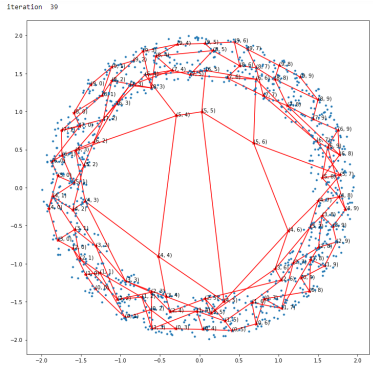

iteration 70
after iteration 70


iteration 99
after last iteration


iteration 0


iteration 1


iteration 25


iteration 39

Code

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import random
import warnings
warnings.filterwarnings('ignore')
from matplotlib.pyplot import figure
import cv2
```

SelfOrginizingMap_line

In [7]:
```
class SelfOrginizingMap_line:
    def __init__(self, learning_rate, data, neighborhood, n, epoch):
        self.learning_rate = learning_rate
        self.data = data
        self.neighborhood = neighborhood
        self.n = n
        self.epoch = epoch

    def print_graph(self, nlist, slist):
        figure(figsize=(12, 12))
        X = []
        Y = []
        for i in range(len(nlist)):
            X.append(nlist[i][0])
            Y.append(nlist[i][1])
            plt.annotate(i, (nlist[i][0], nlist[i][1]))
            plt.plot(nlist[i][0], nlist[i][1], '*r')

        Xp = []
        Yp = []
        for n in slist:
            Xp.append(n[0])
            Yp.append(n[1])
        plt.plot(Xp, Yp,'.')
        plt.plot(X,  Y)
        plt.show()

    def check_new(self, new):
        if new[0] > 1:
            new[0] = 1
        if new[0] < 0:
            new[0] = 0
        if new[1] > 1:
            new[1] = 1
        if new[1] < 0:
            new[1] = 0
        return new

    def make_nlist(self):
        nueron_lst = []
        i = 0
        while i < self.n:
            randX = random.uniform(0, 1)
            randY = random.uniform(0, 1)
            if 0.4 <= randX <= 0.6:
                if 0.4 <= randY <= 0.6:
                    nueron_lst.append(np.array([randX, randY]))
                    i += 1
        return nueron_lst
```

```
def SOM_d(self):
    nueron_lst = self.make_nlist()
    mid = 15

    ker = cv2.getGaussianKernel(mid * 2 + 1,-1)

    sampled_points = []
    for i in range(self.epoch):
        print("iteration ",i)
        if i > 0 and i % 3 == 0:
            self.neighborhood -= 1
            if self.neighborhood < 1:
                self.neighborhood = 1
        if i > 0 and i % 25 == 0:
            self.learning_rate /= 2

        sampled_points = []
        data = self.data.sample(frac=1)
        for index, row in data.iterrows():
            sampled_points.append(np.array([row[0], row[1]]))
            small_dist = np.inf

            # find closest nueron
            for n in range(len(nueron_lst)):
                dist = np.sqrt((row[0] - nueron_lst[n][0]) ** 2 + (row[1] - nueron_lst[n][1]) ** 2)
                if dist < small_dist:
                    small_dist=dist
                    spot = n

            # update weights of nuerons
            new = nueron_lst[spot] + self.learning_rate * ker[mid] * (sampled_points[-1] - nueron_lst[spot])
            nueron_lst[spot] = new

            # update weights
            for j in range(1, self.neighborhood):
                new = nueron_lst[(spot+j) % len(nueron_lst)] + self.learning_rate * ker[mid+j] *
                (sampled_points[-1] - nueron_lst[(spot+j) % len(nueron_lst)])
                nueron_lst[(spot+j) % len(nueron_lst)] = new
                new = nueron_lst[(spot-j) % len(nueron_lst)] + self.learning_rate * ker[mid-j] *
                (sampled_points[-1] - nueron_lst[(spot-j) % len(nueron_lst)])
                nueron_lst[(spot-j) % len(nueron_lst)] = new

        if i%10==0:
            print("after iteration ",i )
            self.print_graph(nueron_lst,sampled_points)
    print("after last iteration " )
    self.print_graph(nueron_lst, sampled_points)
```

```
def SOM(self):
    nueron_lst = self.make_nlist()
    mid = 25

    ker = cv2.getGaussianKernel(mid * 2 + 1,-1)

    sampled_points = []
    for i in range(self.epoch):
        print("iteration ",i)
        if i > 0 and i % 3 == 0:
            self.neighborhood -= 1
            if self.neighborhood < 1:
                self.neighborhood = 1
        if i > 0 and i % 25 == 0:
            self.learning_rate /= 2

        sampled_points = []
        data = self.data.sample(frac=1)
        for index, row in data.iterrows():
            sampled_points.append(np.array([row[0], row[1]]))
            small_dist = np.inf

            # find closest nueron
            for n in range(len(nueron_lst)):
                dist = np.sqrt((row[0] - nueron_lst[n][0]) ** 2 + (row[1] - nueron_lst[n][1]) ** 2)
                if dist < small_dist:
                    small_dist=dist
                    spot = n

            # update weights of nuerons
            new = nueron_lst[spot] + self.learning_rate * ker[mid] * (sampled_points[-1] - nueron_lst[spot])
            new = self.check_new(new)
            nueron_lst[spot] = new

            # update weights
            for j in range(1, self.neighborhood):
                if spot-j >= 0:
                    new = nueron_lst[spot-j] + self.learning_rate * ker[mid-j] * (sampled_points[-1] - nueron_lst[spot-j])
                    new = self.check_new(new)
                    nueron_lst[spot-j] = new
                if spot+j < len(nueron_lst):
                    new = nueron_lst[spot+j]+ self.learning_rate * ker[mid+j] * (sampled_points[-1] - nueron_lst[spot+j])
                    new = self.check_new(new)
                    nueron_lst[spot+j] = new
        if i%10==0:
            print("after iteration ",i )
            self.print_graph(nueron_lst,sampled_points)
    print("after last iteration " )
    self.print_graph(nueron_lst, sampled_points)
```

In [8]:
```
data_line = pd.DataFrame()
random.seed(1)
for i in range(2000):
    randX = random.uniform(0, 1)
    randY = random.uniform(0, 1)
    data_line[i] = [randX, randY]
data_line = data_line.T

som_line = SelfOrginizingMap_line(learning_rate=0.9, data=data_line, neighborhood=25, n=100, epoch=100)
som_line.SOM()
```

In [9]:
```
data_corner = pd.DataFrame()
random.seed(1)
for i in range(1000):
    randX = random.uniform(0, 1)
    randY = random.uniform(0, 1)
    if randX <= 0.5:
        randX = randX ** 2
    else:
        randX = randX ** 0.5
    if randY <= 0.5:
        randY = randY ** 2
    else:
        randY = randY ** 0.5
    data_corner[i] = [randX, randY]
data_corner = data_corner.T

som_line = SelfOrginizingMap_line(learning_rate=0.9, data=data_corner, neighborhood=25, n=100, epoch=100)
som_line.SOM()
```

In [10]:
```
data_cubed = pd.DataFrame()
random.seed(1)
for i in range(1000):
    randX = random.uniform(0, 1)
    randY = random.uniform(0, 1)
    data_cubed[i] = [randX ** 3, randY ** 3]
data_cubed = data_cubed.T

som_line = SelfOrginizingMap_line(learning_rate=0.9, data=data_cubed, neighborhood=25, n=100, epoch=100)
som_line.SOM()
```

In [11]:
```
# make doughnut
doughnut = pd.DataFrame()
random.seed(1)
i = 0
while i < 2000:
#     print(i)
    randX = random.uniform(-4, 4)
    randY = random.uniform(-4, 4)
    if 2 <= randX ** 2 + randY ** 2 <= 4:
        doughnut[i] = [randX, randY]
        i += 1
doughnut = doughnut.T

som_doughnut = SelfOrginizingMap_line(learning_rate=0.9, data=doughnut, neighborhood=15, n=30, epoch=100)
som_doughnut.SOM_d()
```

SelfOrginizingMap_grid

In [2]:
```
class SelfOrginizingMap_grid:
    def __init__(self, learning_rate, data, neighborhood, n, epoch):
        self.learning_rate = learning_rate
        self.data = data
        self.neighborhood = neighborhood
        self.n = n
        self.epoch = epoch

    def print_graph(self, nlist, slist):
        figure(figsize=(12, 12))
        X = []
        Y = []
        for i in range(len(nlist)):
            for j in range(len(nlist[i])):
                X.append(nlist[i][j][0])
                Y.append(nlist[i][j][1])
                plt.annotate((i, j),(nlist[i][j][0],nlist[i][j][1]))
                plt.plot(nlist[i][j][0],nlist[i][j][1],'*r')

        for i in range(self.n):
            Xrow = []
            Yrow = []
            Xcol = []
            Ycol = []
            for j in range(self.n):
                Xrow.append(nlist[i][j][0])
                Yrow.append(nlist[i][j][1])
                Xcol.append(nlist[j][i][0])
                Ycol.append(nlist[j][i][1])
            plt.plot(Xrow, Yrow, "r")
            plt.plot(Xcol, Ycol, "r")

        Xp=[]
        Yp=[]
        for n in slist:
            Xp.append(n[0])
            Yp.append(n[1])
        plt.plot(Xp,Yp,'.')
        plt.show()

    def make_nlist(self):
        nueron_lst = []
        row = []

        xarr = np.linspace(.4, .6, 10)
        yarr = np.linspace(.4, .6, 10)

        for j in range(self.n):
            row = []
            for i in range(self.n):
                randX = random.uniform(0, 1)
                randY = random.uniform(0, 1)
                row.append([xarr[i],yarr[j]])
            nueron_lst.append(row)
        return nueron_lst
```

```
    def SOM(self):
        nueron_lst = self.make_nlist()
        mid = 10

        ker = cv2.getGaussianKernel(mid * 2 + 1,-1)
        ker = ker.dot(ker.T)

        sampled_points = []
        for i in range(self.epoch):
            print("iteration ",i)
            self.print_graph(nueron_lst, sampled_points)
            if i > 0 and i % 7 == 0:
                self.learning_rate /= .5
                self.neighborhood -= 1
            sampled_points = []
            data = self.data.sample(frac=1)
            for index, row in data.iterrows():
                sampled_points.append(np.array([row[0], row[1]]))
                small_dist = np.inf

                # find closest nueron
                for k in range(len(nueron_lst)):
                    for r in range(len(nueron_lst[k])):
                        dist = np.sqrt((row[0] - nueron_lst[k][r][0]) ** 2 + (row[1] - nueron_lst[k][r][1]) ** 2)
                        if dist < small_dist:
                            small_dist = dist
                            spotrow = k
                            spotcol = r

                # update weights
                for j in range(spotrow - self.neighborhood, spotrow + self.neighborhood + 1):
                    for k in range(spotcol - self.neighborhood, spotcol + self.neighborhood + 1):
                        if 0 <= j < len(nueron_lst) and 0 <= k < len(nueron_lst[j]):
                            new = nueron_lst[j][k] + self.learning_rate * ker[mid-spotrow+j][mid-spotcol+k] *
                            (sampled_points[-1] - nueron_lst[j][k])
                            nueron_lst[j][k] = new
        print("iteration ",i )
        self.print_graph(nueron_lst, sampled_points)
```

In [3]:
```
data = pd.DataFrame()
random.seed(1)
for i in range(1000):
    randX = random.uniform(0, 1)
    randY = random.uniform(0, 1)
    data[i] = [randX, randY]
data = data.T

som1 = SelfOrginizingMap_grid(learning_rate=0.7, data=data, neighborhood=10, n=10, epoch=50)
som1.SOM()
```

In [4]:
```
data_sq = pd.DataFrame()
random.seed(1)
for i in range(1000):
    randX = random.uniform(0, 1)
    randY = random.uniform(0, 1)
    if randX <= 0.5:
        randX = randX ** 2
    else:
        randX = randX ** 0.5
    if randY <= 0.5:
        randY = randY ** 2
    else:
        randY = randY ** 0.5
    data_sq[i] = [randX, randY]
data_sq = data_sq.T

som_sq = SelfOrginizingMap_grid(learning_rate=0.6, data=data_sq, neighborhood=5, n=10, epoch=10)
som_sq.SOM()
```

In [5]:
```
# make doughnut
doughnut = pd.DataFrame()
random.seed(1)
i = 0
while i < 1000:
    randX = random.uniform(-4, 4)
    randY = random.uniform(-4, 4)
    if 2 <= randX ** 2 + randY ** 2 <= 4:
        doughnut[i] = [randX, randY]
        i += 1
doughnut = doughnut.T

som_doughnut = SelfOrginizingMap_grid(learning_rate=0.9, data=doughnut, neighborhood=5, n=10, epoch=40)
som_doughnut.SOM()
```

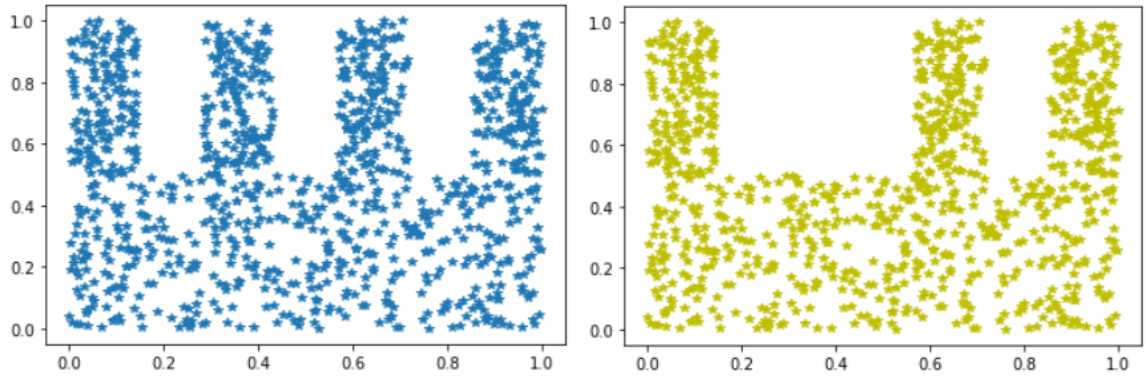
In [6]:
```
data_cubed = pd.DataFrame()
random.seed(1)
for i in range(1000):
    randX = random.uniform(0, 1)
    randY = random.uniform(0, 1)
    data_cubed[i] = [randX ** 3, randY ** 3]
data_cubed = data_cubed.T

som_cubed = SelfOrginizingMap_grid(learning_rate=0.9, data=data_cubed, neighborhood=25, n=10, epoch=100)
som_cubed.SOM()
```

Part B:

Discussions and Illustrations

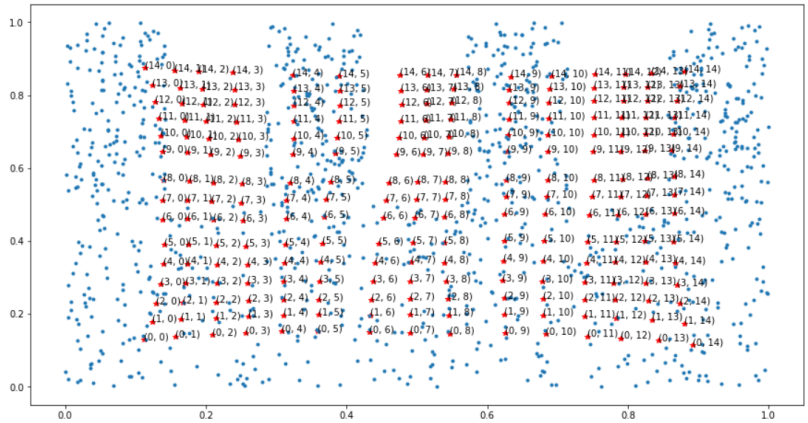
For this part of the project we reproduced the experiment on the "monkey hand" as described in class, **we used our implementation.**
First we made the data set of the hands:



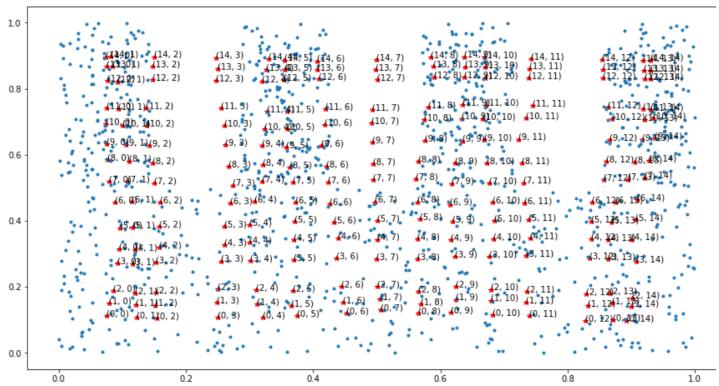
Then we trained the self-organizing map on the “hand with 4 fingers” data, then we took the result map and retrained it on the “hand with 3 fingers” data.

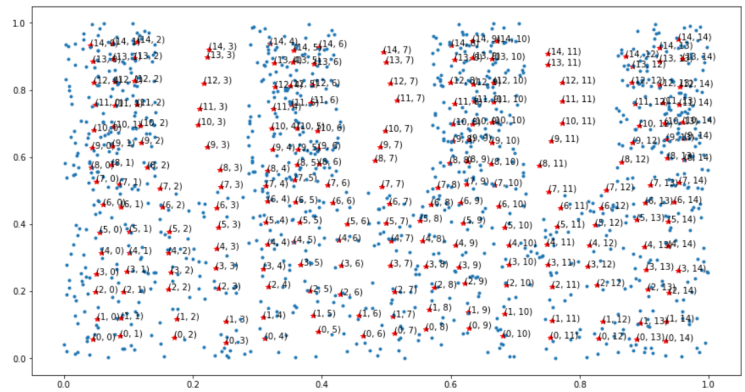
You can see that over time the neurons that were over the missing finger start spreading out to the neighboring fingers and we are left with a single line of neurons in the middle that is stuck there because it is getting pulled from both sides.

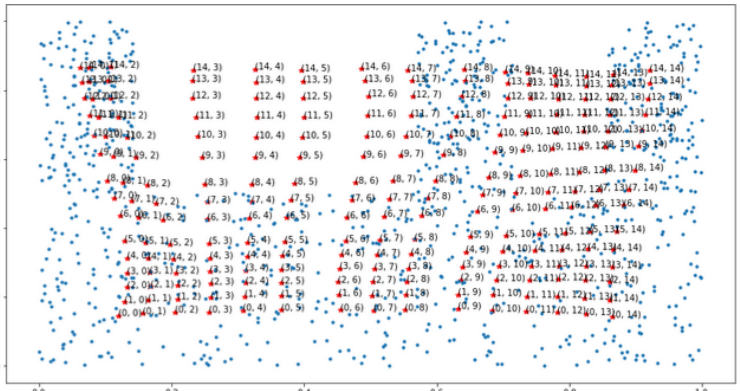
end of iteration 0

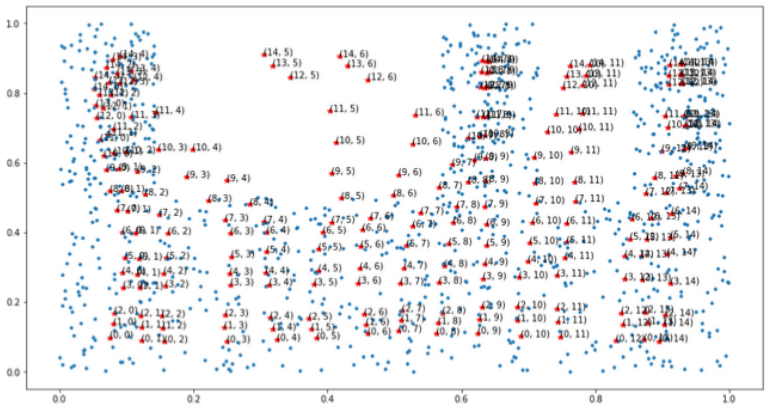


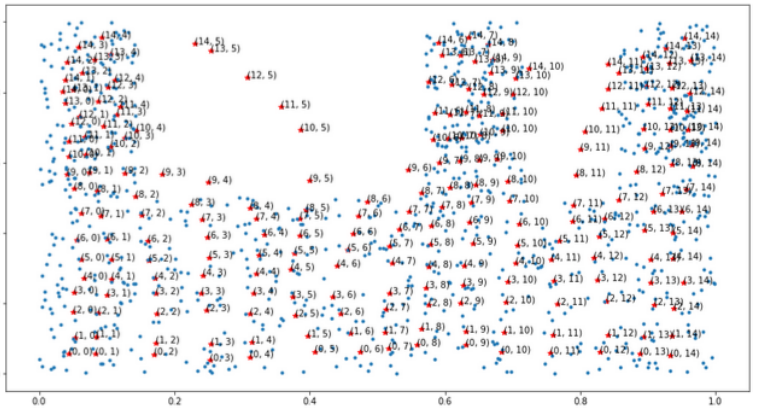
end of iteration 40











Code

Monkey Hand

In [3]:

```
fin=np.arange(7)
fin=np.true_divide(fin, 7)
fin=np.append(fin,1)
print(fin)



df_4_finger=pd.DataFrame()
df_3_finger=pd.DataFrame()
random.seed(1)
#hand
i=0
while i<500:
    randX = random.uniform(0, 1)
    randY = random.uniform(0, 1)
    if randY<=0.5:
        df_4_finger[i] = [randX, randY]
        df_3_finger[i] = [randX, randY]
        i+=1


#pointer
while i<650:
    randX = random.uniform(0, 1)
    randY = random.uniform(0, 1)
    if randY>=0.5:
        if fin[0]<randX<fin[1]:
            df_4_finger[i] = [randX, randY]
            df_3_finger[i] = [randX, randY]
            i+=1
```

```
#ring
while i<800:
    randX = random.uniform(0, 1)
    randY = random.uniform(0, 1)
    if randY>=0.5:
        if fin[4]<randX<fin[5]:
            df_4_finger[i] = [randX, randY]
            df_3_finger[i] = [randX, randY]
            i+=1

#pinky
while i<950:
    randX = random.uniform(0, 1)
    randY = random.uniform(0, 1)
    if randY>=0.5:
        if fin[6]<randX<fin[7]:
            df_4_finger[i] = [randX, randY]
            df_3_finger[i] = [randX, randY]
            i+=1

#middle
while i<1100:
    randX = random.uniform(0, 1)
    randY = random.uniform(0, 1)
    if randY>=0.5:
        if fin[2]<randX<fin[3]:
            df_4_finger[i] = [randX, randY]
            i+=1

df_4_finger=df_4_finger.T
df_3_finger=df_3_finger.T
```

```
Xp4=[]
Yp4=[]
for index, row in df_4_finger.iterrows():
    Xp4.append(row[0])
    Yp4.append(row[1])

plt.plot(Xp4,Yp4,'*')
plt.show()

Xp3=[]
Yp3=[]
for index, row in df_3_finger.iterrows():
    Xp3.append(row[0])
    Yp3.append(row[1])

plt.plot(Xp3,Yp3,'*y')
plt.show()
```

In [24]:

```
def SOM_hand(data,n,nueron_lst):
#     nueron_lst=make_nlistsq(n)
    mid = 10
    num = 5
    alpha = 1
    ker = cv2.getGaussianKernel(mid * 2 + 1,-1)
    ker = ker.dot(ker.T)

    sampled_points = []
    for i in range(75):

        if i > 0 and i % 11 == 0:
            alpha-=0.1
            num-=1


        sampled_points = []
        data = data.sample(frac=1)
        for index, row in data.iterrows():
            sampled_points.append(np.array([row[0], row[1]]))
            small_dist = np.inf

            # find closest nueron
            for n in range(len(nueron_lst)):
                for r in range(len(nueron_lst[n])):
                    dist = np.sqrt((row[0] - nueron_lst[n][r][0]) ** 2 + (row[1] - nueron_lst[n][r][1]) ** 2)
                    if dist < small_dist:
                        small_dist = dist
                        spotrow=n
                        spotcol=r

            for j in range(spotrow - num, spotrow + num + 1):
                for k in range(spotcol - num, spotcol + num + 1):
                    if 0 <= j < len(nueron_lst) and 0 <= k < len(nueron_lst[j]):

                        new = nueron_lst[j][k] + alpha * ker[mid-spotrow+j][mid-spotcol+k] *
                        (sampled_points[-1] - nueron_lst[j][k])
                        nueron_lst[j][k] = new
        if i%10==0:
            print("end of iteration ",i )
            print_graphsq(nueron_lst,sampled_points)

    print("iteration ",i )
    print_graphsq(nueron_lst,sampled_points)
    return nueron_lst
```

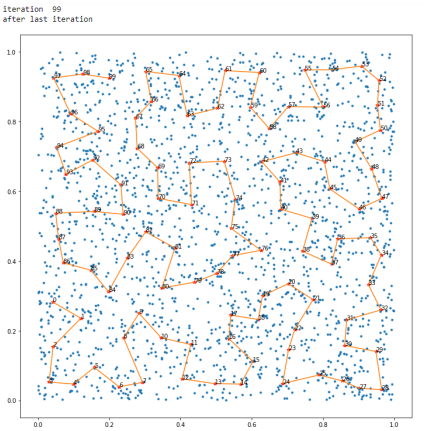
In [25]:

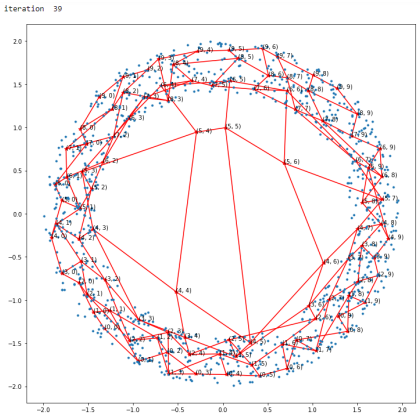
```
nueron_lst=make_nlistsq(15)
nl=SOM_hand(df_4_finger,15,nueron_lst)
SOM_hand(df_3_finger,15,nl)
```

Summary and Conclusion

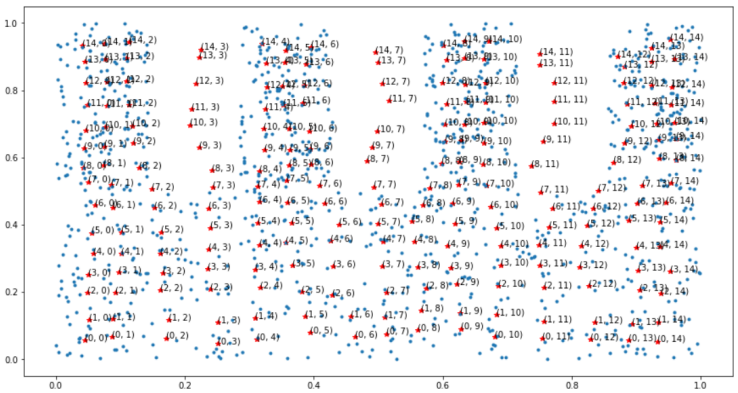
For the first part where the data is uniformly distributed we assumed that the map would be uniformly spread over the space of the sampled points.
From the results we showed we can see that our assumption was correct, and both the grid and the line spreaded and used all of the space.

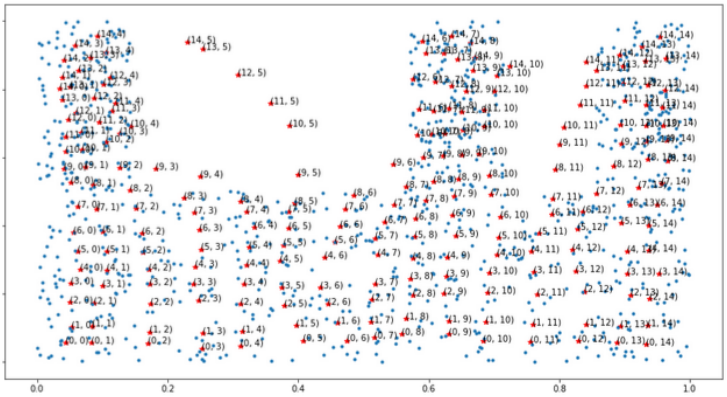
Then for the part where the data is not uniformly distributed we assumed that the map would be constricted towards the denser area of the sampled place and have very few neurons on the sparse area of the sampled place.
We can see that our assumptions were correct, for both the grid and the line.





For the second part of this project, where we worked on the “monkey hand” data, we saw the map arranged to the data of the four fingers, then when we “took off” one finger we assumed that the map would change and there would not be any neuron where the finger used to be, in the result we saw that one column of neurons stayed. We assumed that this happened because they were pulled by both sides and that made them stay in their place.





During this whole project we checked for each data what hyperparameters will give us the best result, so we set them differently for each data set.
The hyperparameters we examined were: learning rate, number of neighbors and the number of epochs.
For each of the data sets we set those hyperparameters so that the map is arranged the best way.

```
som_sq = SelfOrginizingMap_grid(learning_rate=0.6, data=data_sq, neighborhood=5, n=10, epoch=10)
som_sq.SOM()
```

```
som_doughnut = SelfOrginizingMap_grid(learning_rate=0.9, data=doughnut, neighborhood=5, n=10, epoch=40)
som_doughnut.SOM()
```

```
som_cubed = SelfOrginizingMap_grid(learning_rate=0.9, data=data_cubed, neighborhood=25, n=10, epoch=100)
som_cubed.SOM()
```

We enjoyed working on this project and seeing how changing the hyperparameters changed how the network works, and the results.

The code can be found here -
https://github.com/Yehudit-Brickner/nueroComputationEx2