Computational Neuro-Project 2- Parts A-B

Ayala Bouhnik-Gelbord
Jonatan Boritsky
Ginton Durlacher
Gil Tzioni

Introduction-

As part of the neuro-computational course, in our second project we were asked to implement the Kohonen (SOM) algorithm.

SOM Algorithm-

The **self-organising map** (**SOM**) learning algorithm is relatively straightforward.
It consists of initializing the weight, iterating over the input data, finding the "winning" neuron for each input, and adjusting weights based on the location of that "winning" neuron.
A pseudocode implementation is provided below:

Initialize weights

For 0 to X number of training epochs:

- Select a sample from the input data set
- Find the "winning" neuron for the sample input
- Adjust the weights of nearby neurons

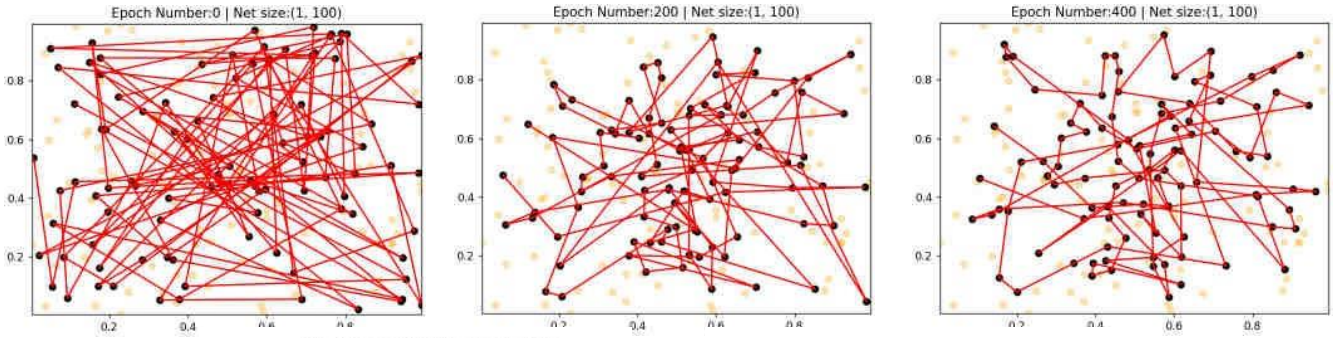
End loop

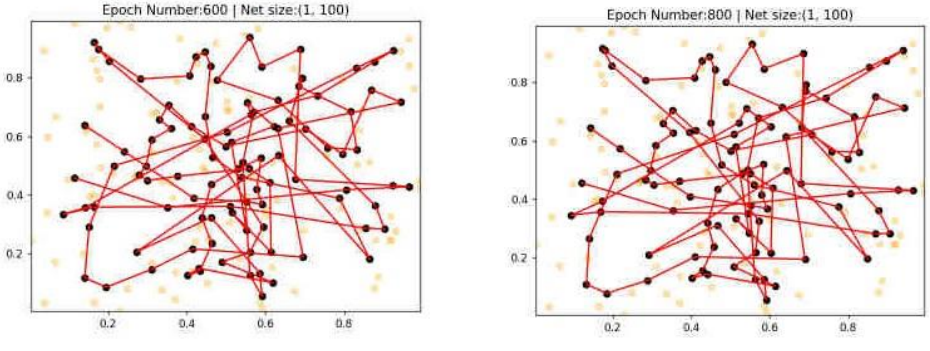
To read more about the algorithm- https://www.cs.hmc.edu/~kpang/nn/som.html

Part A-

First section-

In this part we implemented the Kohonen algorithm and we used it to fit a set of 100 neurons in a topology of a line to a disk, where the **data set** is:
$\{(x, y)\,|0 \leq x \leq 1\,,\ 0 \leq y \leq 1\}$ for which the distribution is uniform while the Kohonen level is linearly ordered.
We obtain data by sampling the data set.
We changed the number of iterations each time, to see its influence on the results:



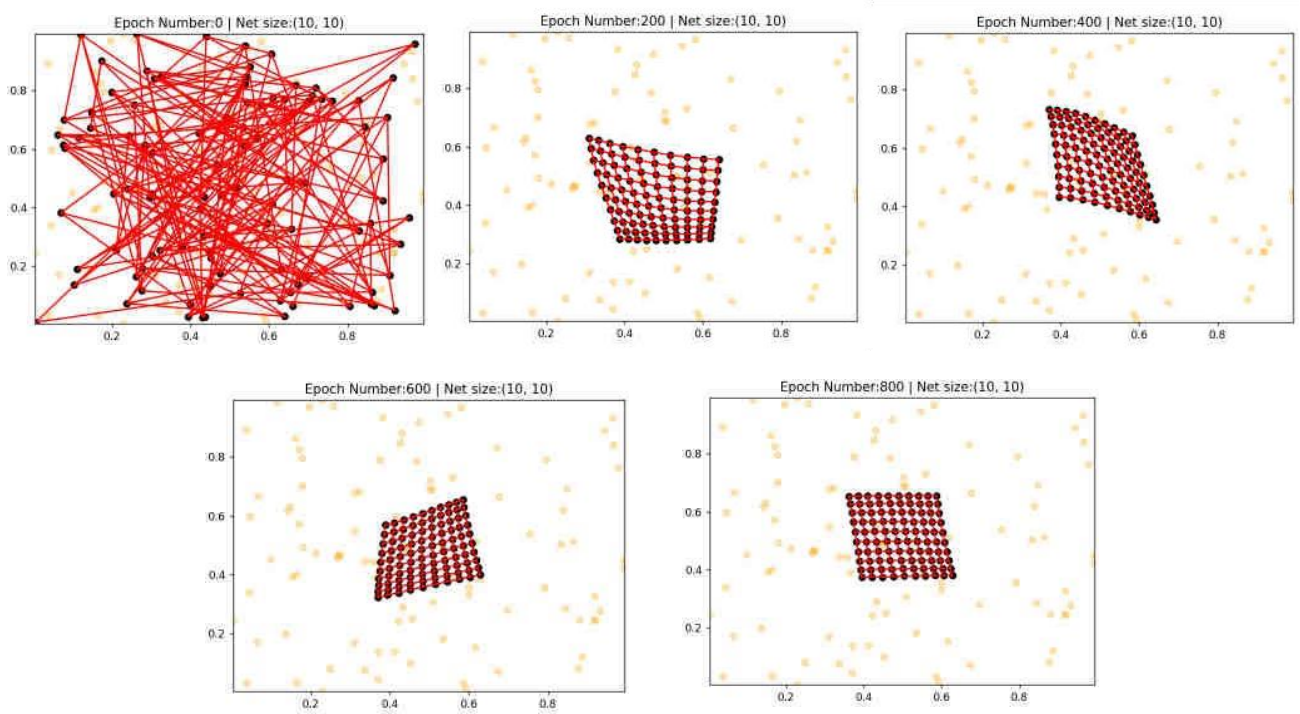


We can see the different parameters at the top of each graph:

Net size: The size of training data.

Epoch Number: The number of iterations that the algorithm will do.

We did the same when the topology of the 100 neurons is arranged in a two dimensional array of 10 x 10. And we got the following results:

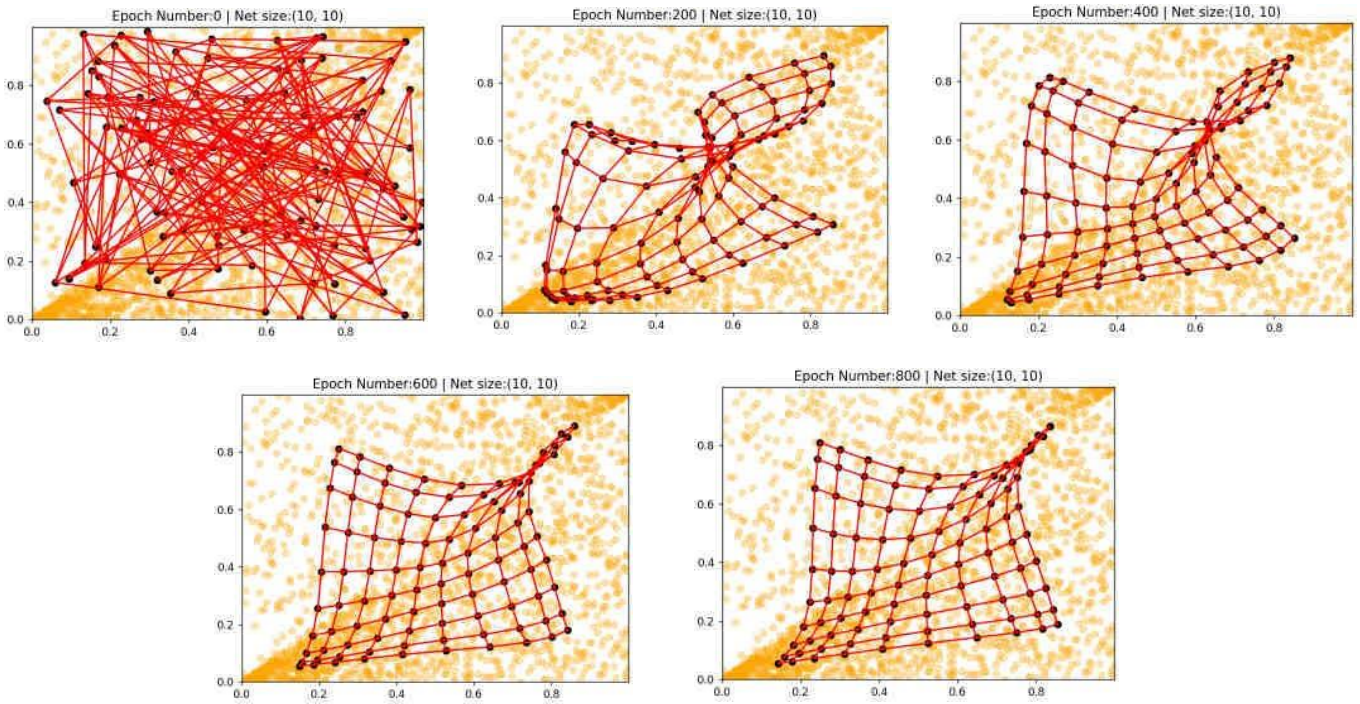


As we can see, after 200 iterations(and possibly even less) the change is minor.
We can see that when the number of iterations of the algorithm increases, the neurons are interpreted more uniformly and more accurately on the data.
Due to the fact that our information is uniform, our neurons tend to disperse uniformly.

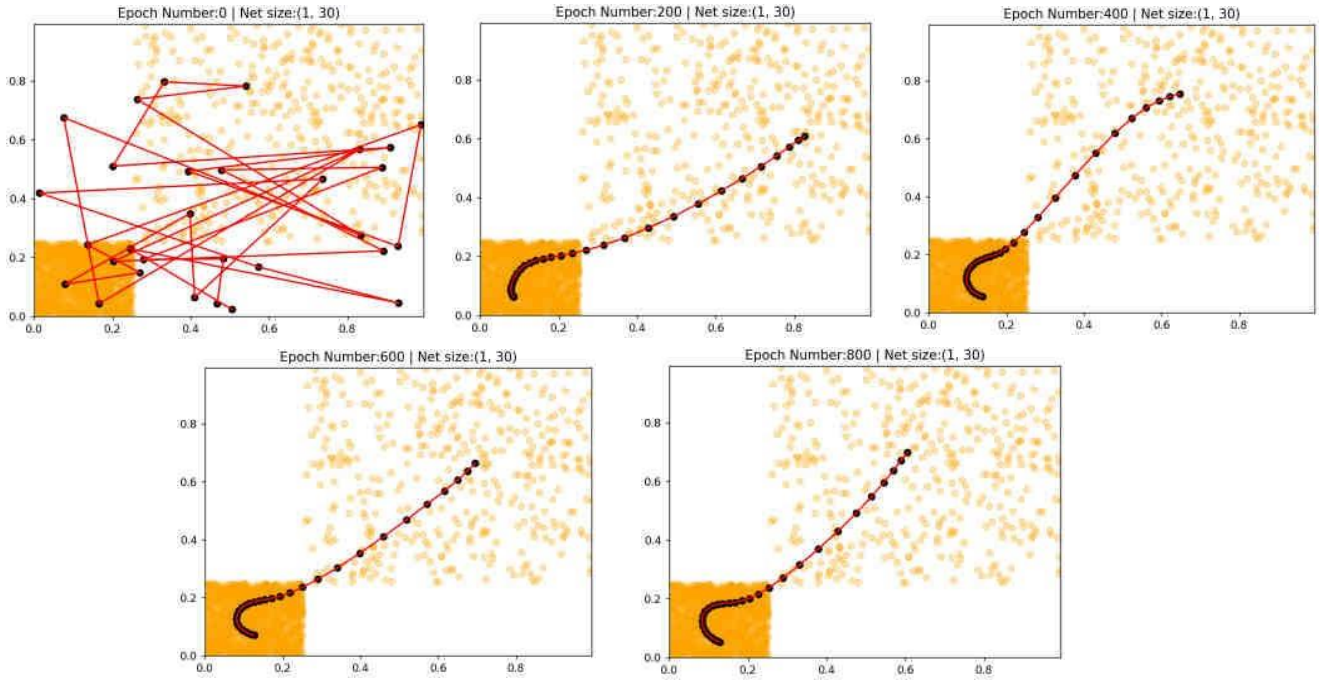
Second section-

In this section, we were asked to do the same with at least two non-uniform distributions on the disk.

First distribution- we put 70% of the data in the bottom right corner. These are the results that we received:



The second distribution- we put 80% of the data in the bottom left square.

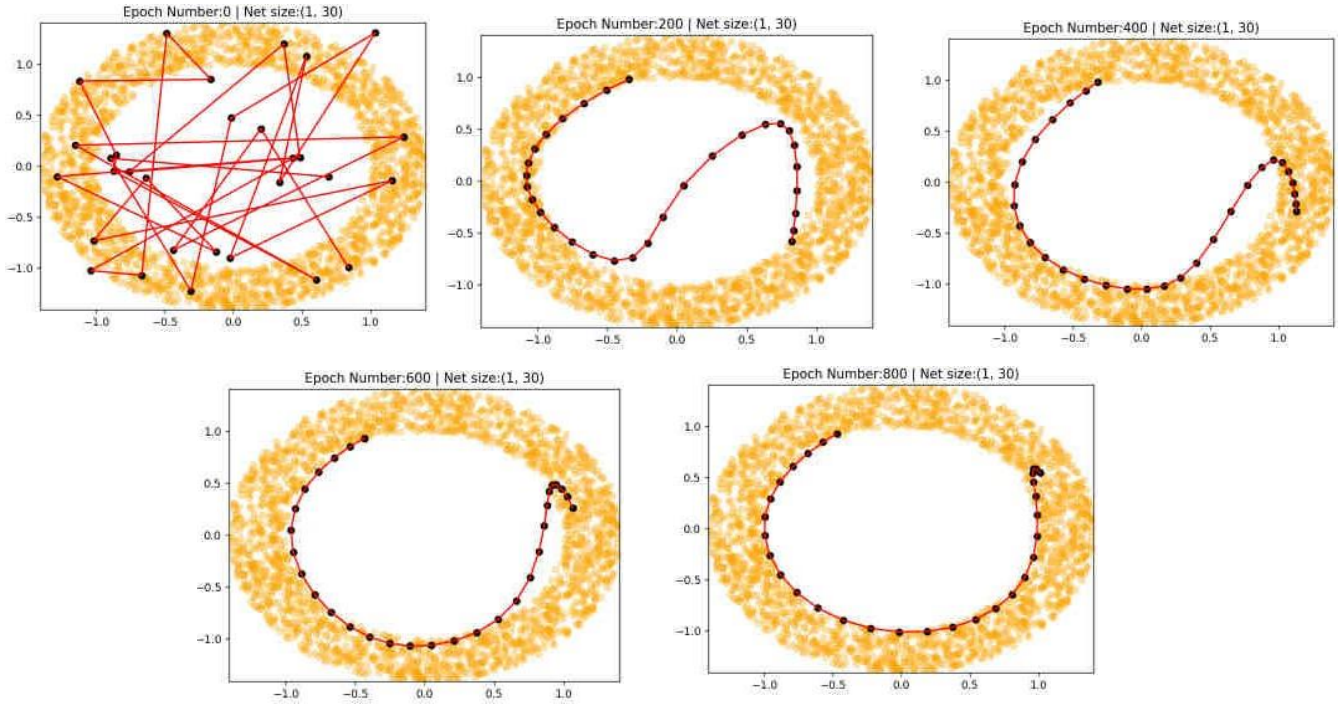


We can see that most of the neurons are between 0 to 0.3, and there are a small number of neurons after 0.4. After epoc 200 the change is minimal.

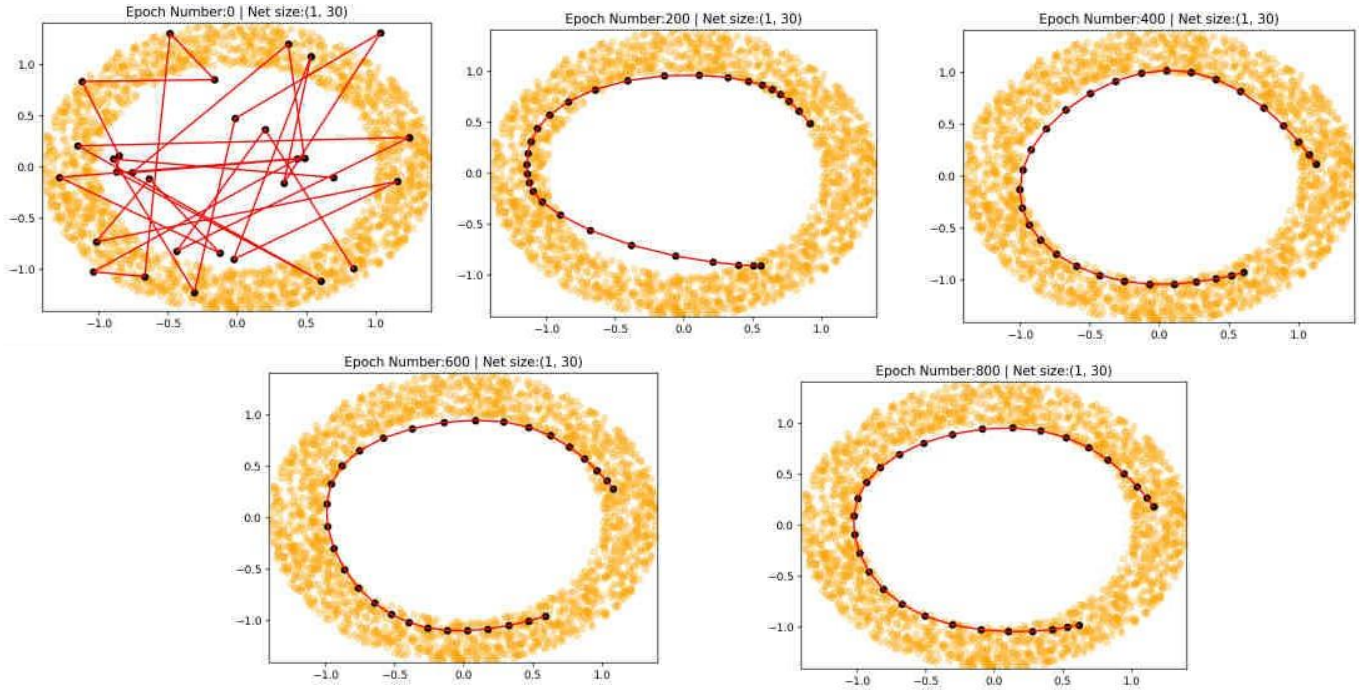
Third section-

In this part we conducted the experiments as above for fitting a circle of neurons on a "donut" shape i.e. {<$x. y$> | $2 \leq x^2+y^2 \leq 4$}. The line of neurons has 30 neurons organized as a circle topology.

We run the algorithm when the learning rate is 0.4, and we received the following results:



We repeated the same experiment, but when the learning rate value is 0.7, we received the following results:



As illustrated in the graphs above, when the learning rate value is 0.7 the results are slightly better in the lowest epochs.

As we can see, after 400 iterations (and possibly even less) the change is minor.

Part B-

First section-

In this section we chose to build the 'monkey hand' shape by using a combination of the following 5 polynomials:

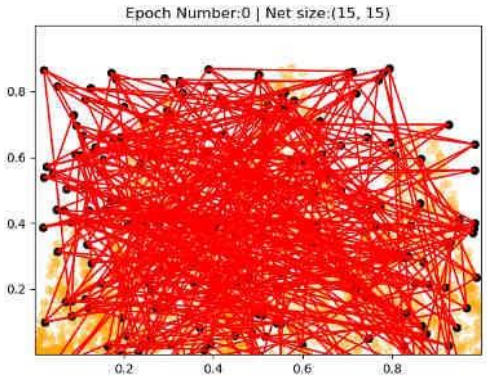
T-function: $-(0.7x - \sqrt{0.2})^2 + 0.2$ S-function: $-(8x - \sqrt{0.4})^2 + 0.4$

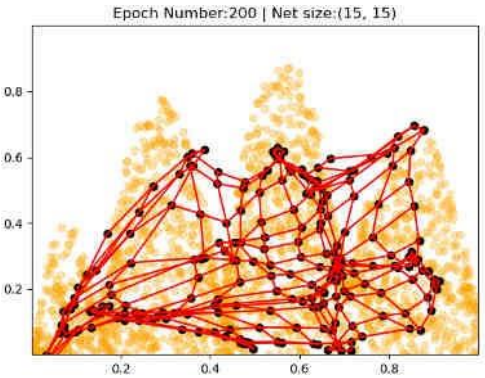
L-function: $-(6x - \sqrt{3})^2 + 0.8$ M-function: $-(6x - \sqrt{11.5})^2 + 0.9$

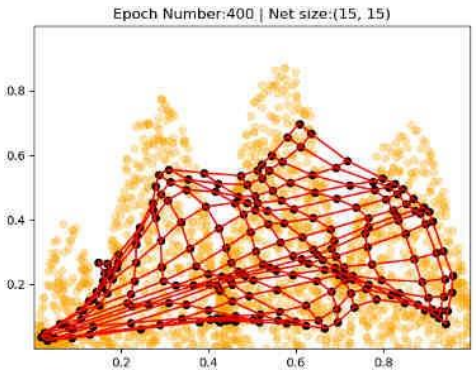
R-function: $-(5x - \sqrt{18})^2 + 0.7$

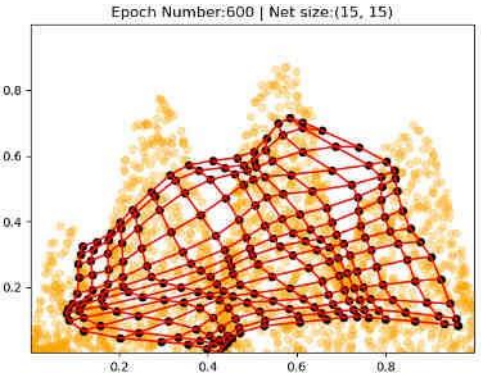
2000 uniformly distributed points were scattered into the shape.

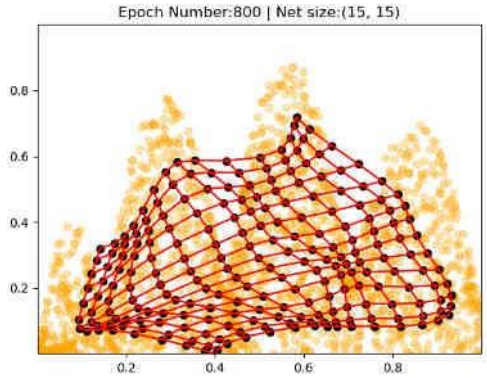
The model was fitted with learning rate 3 and radius 5 (for increasing the closeness of the data points to their cluster's centre and for scattering the points more sparsely into the shape)







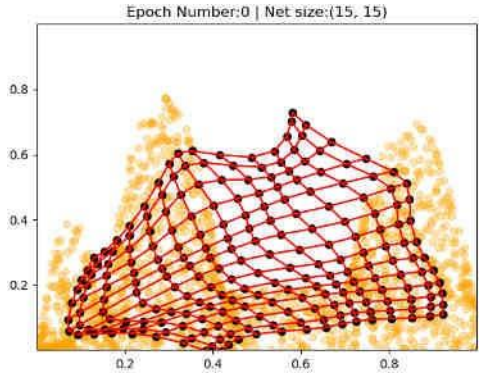


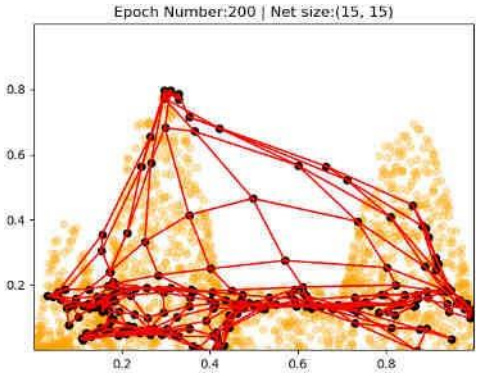


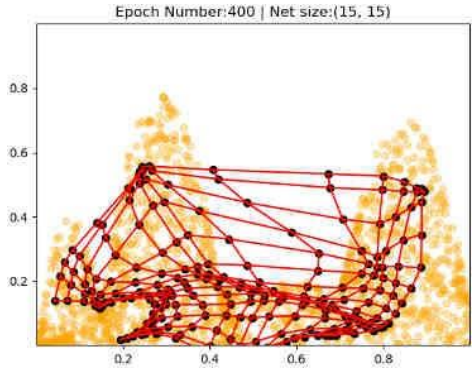
As we can see, the points arranged successfully in the shape.

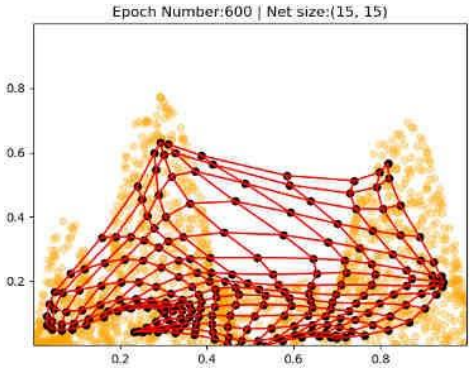
Second section-

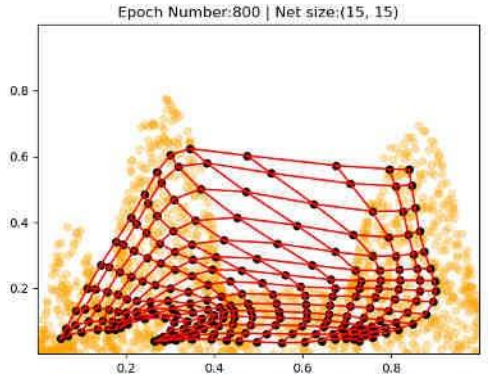
In this section we eliminate the data points which represent the one before them most left finger and continue fitting the model with the same map from the previous stopping point.











As we can see, the points re-arranged successfully in the shape

Code-

SOM implementation:

```
import random

import numpy as np
import matplotlib.pyplot as plt


class SOM(object):
    def __init__(self, hight, width, alpha=0.6, seed=6, radius=2):
        np.random.seed(seed)
        self.shape = (hight, width)
        self.alpha = alpha
        self.radius_strength = radius
        self.data = None

    def fit_data(self, data, iterations=1000, map=None):
        self.data = data
        self.iteration_limit = iterations
        xMax = np.max(data[:, 0])
        xMin = np.min(data[:, 0])
        yMax = np.max(data[:, 1])
        yMin = np.min(data[:, 1])
        if map is None:
            self.map = np.array([[(random.uniform(xMin + 0.001, xMax),
                                   random.uniform(yMin + 0.001, yMax)) for i
in range(self.shape[1])]
                                 for j in range(self.shape[0])])
        else:
            self.map = map

        for i in range(self.iteration_limit):
            n = random.randint(0, len(data) - 1) # randomly pick an input vector
            bmu = self.BMU(data[n])
            if i % 200 == 0:
                draw(self, xMax, xMin, i)
            self.update_map(bmu, data[n], i)
```

```
        #Select a sample from the input data set
        #Find the "winning" neuron for the sample input
        #Adjust the weights of nearby neurons



    def BMU(self, vector): #BMU - Best Matching Unit
        closest_neuron = np.inf
        for x in range(self.shape[0]):
            for y in range(self.shape[1]):
                dist = np.linalg.norm(vector - self.map[x, y])
    #Compute the winner neuron closest to the vector (Euclidean distance)
                if dist < closest_neuron:
                    closest_neuron = dist
                    closest_vector = (x, y)
        return closest_vector

    def update_map(self, bmu, X_i, t):
    #Update map by found BMU at iteration t.
        for x in range(self.shape[0]):
            for y in range(self.shape[1]):
                dist_from_bmu = np.linalg.norm(np.array(bmu) - np.array([x,
y]))
                alpha = self.alpha * np.exp(-t / 300) # update alpha & radius
                radius = np.exp(-np.power(dist_from_bmu, 2) /
self.radius_strength)
                self.map[(x, y)] += alpha * radius * (X_i - self.map[(x, y)])


def draw(self, max, min, t):
    xs = []
    ys = []
    fig, ax = plt.subplots()
    ax.scatter(self.data[:, 0], self.data[:, 1], alpha=0.3, c='orange')

    for i in range(self.map.shape[0]):
        for j in range(self.map.shape[1]):
            xs.append(self.map[i, j, 0])
            ys.append(self.map[i, j, 1])
    ax.scatter([xs], [ys], c='black')
    ax.set_xlim(min, max)
    ax.set_ylim(min, max)

    xs = []
    ys = []
    for i in range(self.map.shape[0]):
        for j in range(self.map.shape[1]):
            xs.append(self.map[i, j, 0])
            ys.append(self.map[i, j, 1])
        ax.plot(xs, ys, 'r-')
        xs = []
        ys = []
```

```
    xs = []
    ys = []
    for i in range(self.map.shape[1]):
        for j in range(self.map.shape[0]):
            xs.append(self.map[j, i, 0])
            ys.append(self.map[j, i, 1])
        ax.plot(xs, ys, 'r-')
        xs = []
        ys = []

    xs = []
    ys = []

    ax.set_title("Epoch Number:" + str(t) + " | " + "Net size:" +
str(self.shape))
    plt.show()
```

Main:

```
import random
import numpy as np
from SOM import SOM


def DataSet(num_of_neurons=100, section="Section 1"):
    data = np.empty((num_of_neurons, 2), dtype=object)
    random.seed(10)
    if section == "Section_1":
        for i in range(num_of_neurons):
            data[i, 0] = random.randint(0, 2000) / 2000
            data[i, 1] = random.randint(0, 2000) / 2000
    elif section == "Section_2_the_first_distribution":  # 70% to be in the
bottom right corner - non uniform
        for i in range(num_of_neurons):
            flag = random.randint(0, 100)
            if flag < 70:
                data[i, 0] = i / 2000
                data[i, 1] = random.randint(0, i) / 2000
            else:
                data[i, 0] = i / 2000
                data[i, 1] = random.randint(i, 2000) / 2000
    elif section == "Section_2_the_second_distribution":  # 80% to be in the
bottom left square - non uniform
        tmp = 0
        for i in range(int(num_of_neurons * 0.2)):
            data[i, 0] = random.randint(500, 2000) / 2000
            data[i, 1] = random.randint(500, 2000) / 2000
            tmp = i
        for j in range(tmp, tmp + int(num_of_neurons * 0.8) + 1):
            data[j, 0] = random.randint(0, 500) / 2000
            data[j, 1] = random.randint(0, 500) / 2000
    elif section == "Section_3":
        num = 0
        while num < num_of_neurons:
```

```
            x = random.uniform(-2, 2)
            y = random.uniform(-2, 2)
            if 1 <= x ** 2 + y ** 2 <= 2:
                data[num, 0] = x
                data[num, 1] = y
                num += 1
    elif section == "Section_4":
        num = 0
        T_points_num = np.int(0.29 * num_of_neurons)
        S_points_num = np.int(0.08 * num_of_neurons)
        L_points_num = np.int(0.22 * num_of_neurons)
        M_points_num = np.int(0.24 * num_of_neurons)
        R_points_num = num_of_neurons - T_points_num - S_points_num -
L_points_num - M_points_num
        for i in range(T_points_num):
            x = random.uniform(0, 1)
            data[num, 0] = x
            y = random.uniform(0, -(0.7*x - np.sqrt(0.2)) ** 2 + 0.2)
            data[num, 1] = y
            num +=1
        for i in range(S_points_num):
            x = random.uniform(0, 0.14947550746085458)
            data[num, 0] = x
            y = random.uniform(-(0.7*x - np.sqrt(0.2)) ** 2 + 0.2, -(8*x -
np.sqrt(0.4)) ** 2 + 0.4)
            data[num, 1] = y
            num +=1
        for i in range(L_points_num):
            x = random.uniform(0.14947550746085458, 0.4566916482105978)
            data[num, 0] = x
            y = random.uniform(-(0.7*x - np.sqrt(0.2)) ** 2 + 0.2, -(6*x -
np.sqrt(3)) ** 2 + 0.8)
            data[num, 1] = y
            num +=1
        for i in range(M_points_num):
            x = random.uniform(0.42350508983327384, 0.7048497641529742)
            data[num, 0] = x
            y = random.uniform(-(0.7*x - np.sqrt(0.2)) ** 2 + 0.2, -(6*x -
np.sqrt(11.5)) ** 2 + 0.9)
            data[num, 1] = y
            num +=1
        for i in range(R_points_num):
            x = random.uniform(0.7067875529577493, 0.9986513633006309)
            data[num, 0] = x
            y = random.uniform(-(0.7 * x - np.sqrt(0.2)) ** 2 + 0.2, -(5 * x -
np.sqrt(18)) ** 2 + 0.7)
            data[num, 1] = y
            num += 1
    return data.astype(np.float64)

def main():
    data1 = DataSet(section="Section_1")
```

```
    # Question 1, part 1-
    SOM(hight=1, width=100, alpha=0.4, radius=2).fit_data(data=data1,
iterations=1000)

    # Question 1, part 1-
    SOM(hight=10, width=10, alpha=0.4, radius=60).fit_data(data=data1,
iterations=1000)

    # # Question 1, part 2-
    data2 = DataSet(num_of_neurons=2000,
section="Section_2_the_first_distribution")
    SOM(hight=10, width=10, radius=6, alpha=0.4).fit_data(data=data2,
iterations=1000)
    #
    # # Question 1, part 2-
    data3 = DataSet(num_of_neurons=2000,
section="Section_2_the_second_distribution")
    SOM(hight=1, width=30, radius=30, alpha=0.4).fit_data(data=data3,
iterations=1000)

    # Question 1, part 3-
    data4 = DataSet(num_of_neurons=2000, section="Section_3")
    SOM(hight=1, width=30, radius=20, alpha=0.4).fit_data(data=data4,
iterations=1000)

    # Question 2, part 1-
    num_of_neurons = 2000  # number of data points
    data5 = DataSet(num_of_neurons=num_of_neurons, section="Section_4")
    som = SOM(hight=15, width=15, radius=5, alpha=3)
    som.fit_data(data=data5, iterations=1000)

    # Question 2, part 2-
    # eliminate the data points which represent the one before the most left
finger
    start = np.int(0.29 * num_of_neurons) + np.int(0.08 * num_of_neurons) +
np.int(0.22 * num_of_neurons)
    end = np.int(0.29 * num_of_neurons) + np.int(0.08 * num_of_neurons) +
np.int(0.22 * num_of_neurons)\
          + np.int(0.24 * num_of_neurons)
    data6 = np.concatenate((data5[0:start, :], data5[end:data5.shape[0], :]))
    # continue fitting the model with the same map from the previous
stopping point
    SOM(hight=15, width=15, radius=5, alpha=3).fit_data(data=data6,
iterations=1000, map=None)
```