# Kohonen algorithm

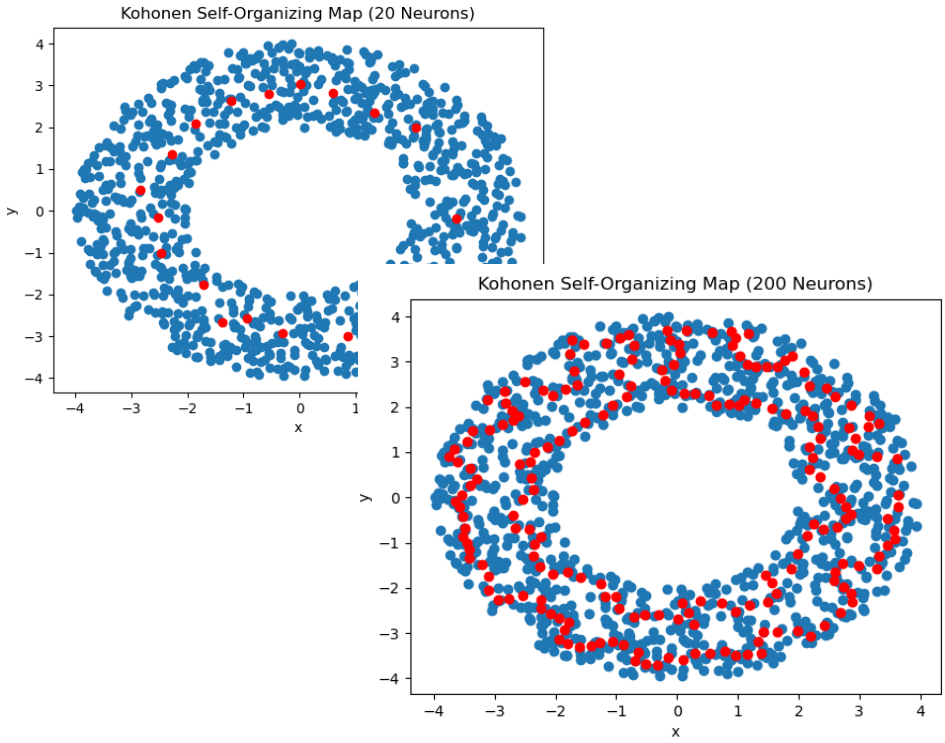
1. Submitters' names:


Abigail Isaacs - 212457535

Yeela Citron - 209914050

Talor Langnes - 204240477

Ohad Wolfman – 316552496


Link to the repository on Github:

https://github.com/ohadwolfman/neural_computation_Ex2_Kohonen_algorithm.git

## 2. introduction:

In this project, we implemented the Kohonen (SOM) algorithm and explore its behavior under different scenarios. The Kohonen algorithm, also known as the Self-Organizing Map algorithm, is a type of artificial neural network used for unsupervised learning and data visualization.

### Part A:

The first part of the project involves fitting a line of random neurons in a data set consisting of points within the square range of $0 <= x <= 1$ and $0 <= y <= 1$. The distribution of the data points is uniform, while the Kohonen level is linearly ordered.The implementation of the algorithm will be tested with two different numbers of neurons: a small number (20) and a large number (200).
The aim is to observe the effect of varying the number of neurons on the algorithm's performance. Additionally, the evolution of the Kohonen map will be examined as the number of iterations increases.
The second part of the Part focuses on fitting a circle of neurons on a "donut" shape. The data set consists of points within the range $4 <= x^2+y^2 <= 16$.
A circle of 300 neurons will be used for this task.

### Part B:

Part B of the project involves replicating an experiment known as the "monkey hand" problem. The data set is defined within a subset of the range $0 <= x <= 1$ and $0 <= y <= 1$, representing the shape of a hand. The Kohonen space consists of 400 neurons arranged in a 20 x 20 mesh. Initially, the entire hand is considered, and the evolution of the mesh is observed over iterations. Then, one of the fingers is "cut off," and the mesh is rearranged accordingly. The aim is to demonstrate how the Kohonen algorithm adapts to changes in the input data distribution.

3. Part A - Code review:
    a. Square shape:
        1) The Kohonen class contains the fields:
            - neurons_amount,
            - learning_rate,
            - Radius,
            - weights.

            Except the init function the class contains the following functions:

```
def initialize_weights(self, input_dim):
    self.weights = np.random.rand(self.neurons_amount, input_dim)

def find_winner(self, input_data):
    distances = np.linalg.norm(input_data - self.weights, axis=1)
    return np.argmin(distances)

def update_weights(self, input_data, winner_idx):
    distance_to_winner = np.abs(np.arange(self.neurons_amount) - winner_idx)
    influence = np.exp(-distance_to_winner / self.radius)
    self.weights += self.learning_rate * influence[:, np.newaxis] * (input_data - self.weights)
```

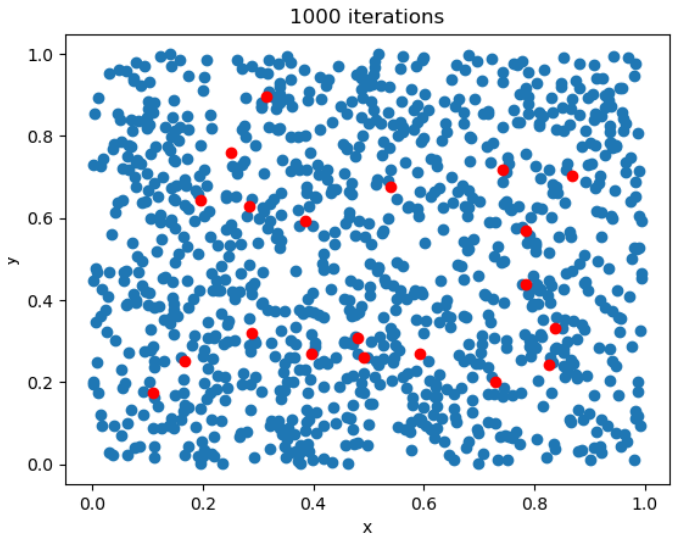
            - initialize_weights – that create samples of weights to the neurons
            - find_winner – for every point from the data, the winner is the neuron with the shortest distance from the that point.
            - update_weights – after we found the winner neuron we updating the weights of the neurons.
            - fit – training the model by updating the weights K times with the previous functions following the number of epochs required
              Into the fit function we added a boolean attribute 'toPlt' – if we want that during the fitting we will present the status of the model in a few possessions (1,30,70,100,300 etc.):

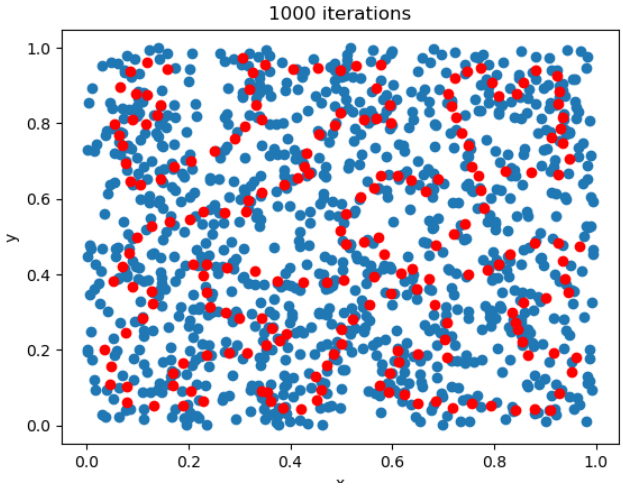
```python
for epoch in range(num_epochs):
    np.random.shuffle(data)

    if toPlt:
        if epoch == 0:
            plot_res_return(axes, 0, 0, data, self.weights)
            axes[0, 0].set_title('0 iterations')
        if epoch == 1:
            plot_res_return(axes, 0, 1, data, self.weights)
            axes[0, 1].set_title('1 iterations')
        if epoch == 30:
            plot_res_return(axes, 0, 2, data, self.weights)
            axes[0, 2].set_title('30 iterations')
        if epoch == 70:
```

2) First we generated the dataset – 1000 points (0<x,y<1) in 2D space X,Y. the we trained the model:

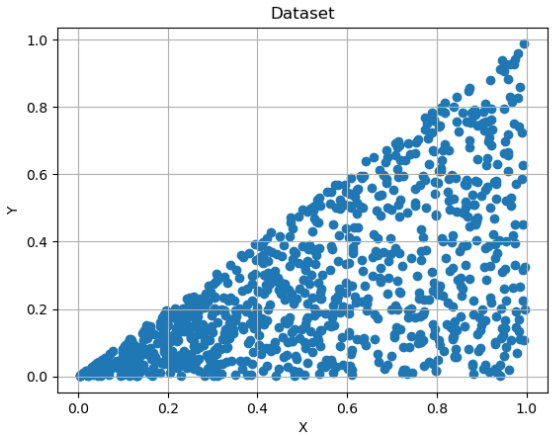
```python
# creating a square uniform dataset
data_square = create_random_dataset(1000)
create_data_plot(data_square)

# activate the kohonen algorithm with 20 neurons and plotting the results
Create_Kohonen_network(data_square, output_dim=20, learning_rate=0.5, num_epochs=1000, toPlt=False)

# activate the kohonen algorithm with 20 neurons and plotting the results
Create_Kohonen_network(data_square, output_dim=200, learning_rate=0.5, num_epochs=1000, toPlt=False)

# creating the first square un-uniform dataset
data_unUni_1 = create_unUni_1_dataset()
create_data_plot(data_unUni_1)
# same as above
Create_Kohonen_network(data_unUni_1, output_dim=20, learning_rate=0.5, num_epochs=1000, toPlt=False)
Create_Kohonen_network(data_unUni_1, output_dim=200, learning_rate=0.5, num_epochs=1000, toPlt=False)
```

3) Create a Kohonen network with 20 and 200 neurons and trained the model on these points:
(We used 1000 epochs and learning rate of 0.5 in all of the trainings below)



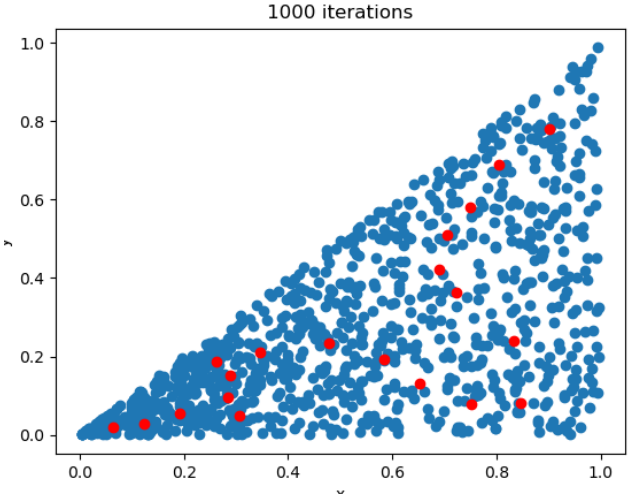



4) creating the first square un-uniform dataset:

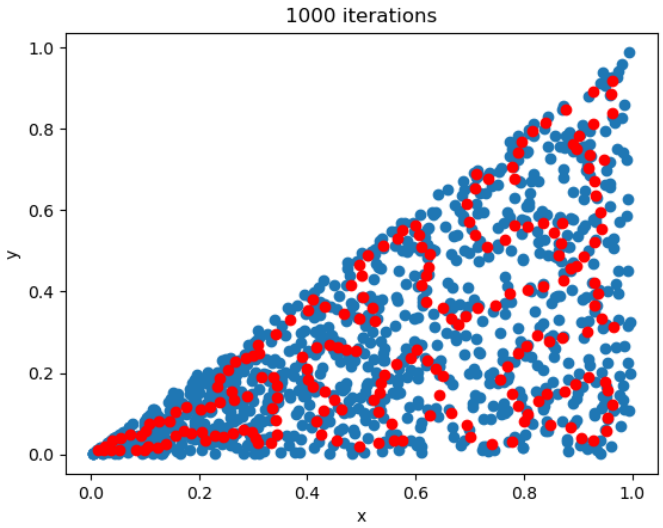



In this code, the data array is generated with non-uniform distributions based on the likelihood proportional to the size of x and uniform to the size of y. The x values are generated uniformly between 0 and 1, and the y values are generated uniformly between 0 and the corresponding x value.

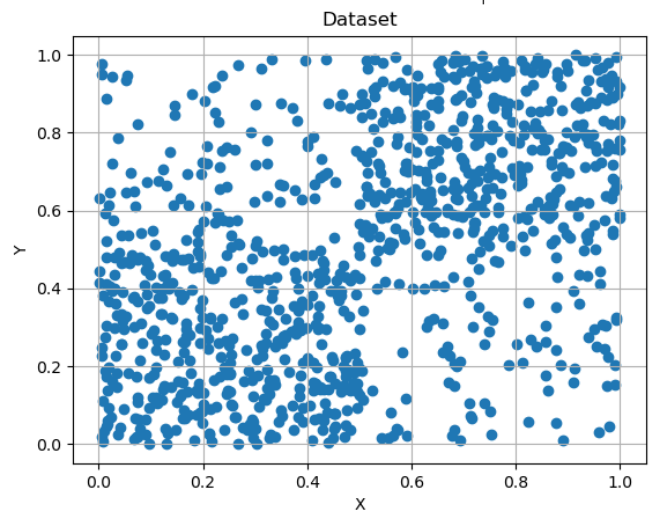
Then we train the model again with 20 and 200 points:



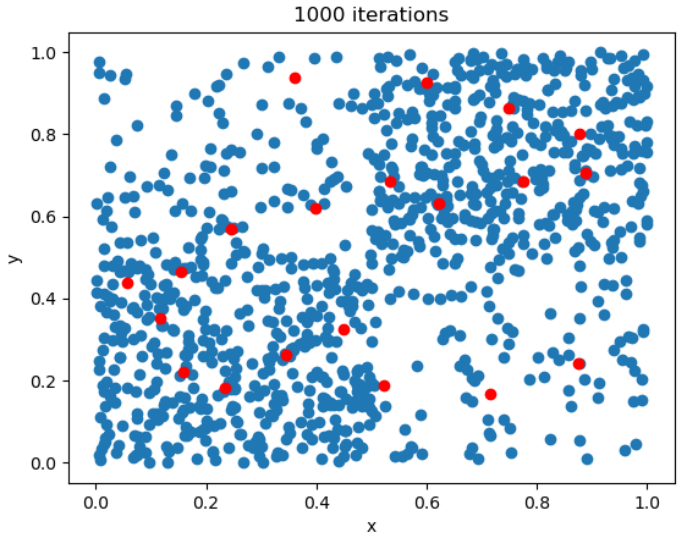
And 200 points:

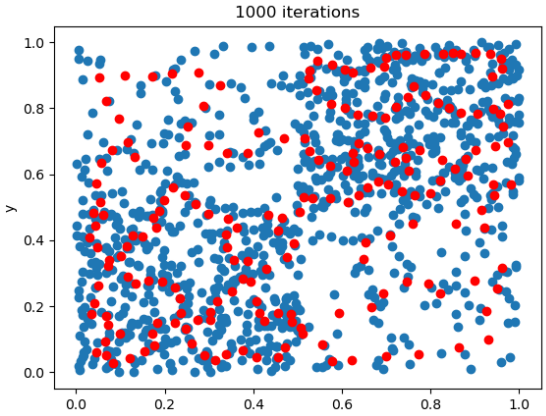



5) Then we created the second square un-uniform dataset:

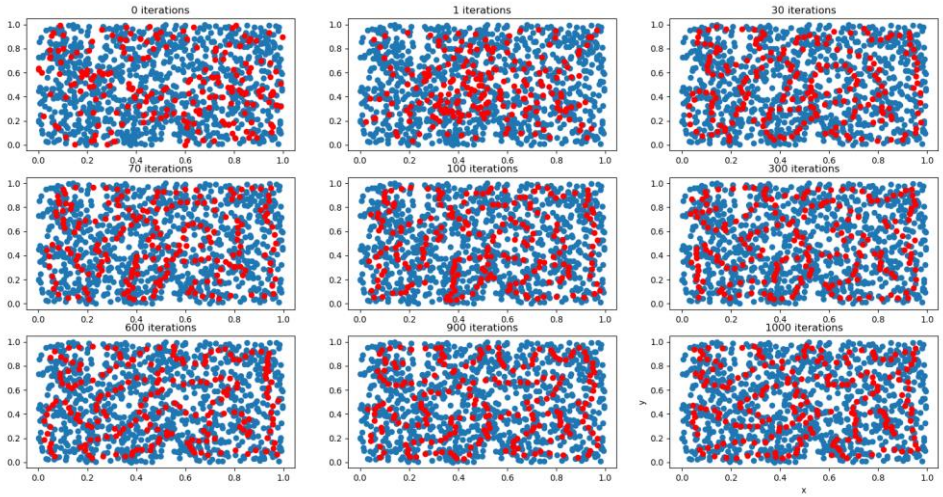



And we trained again with 20 and 200 points:

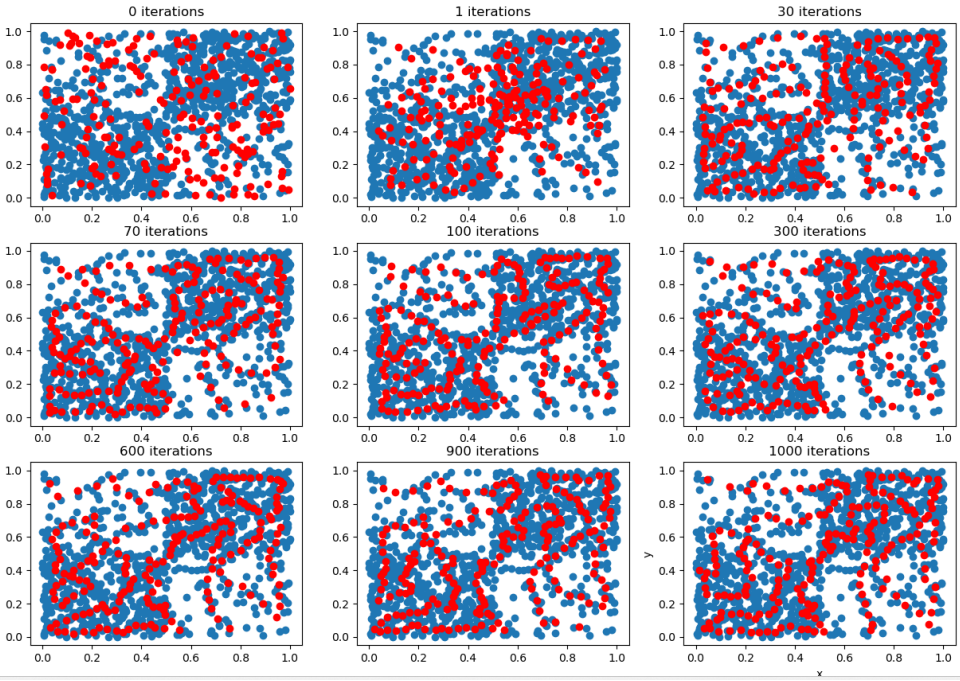





6) In the end we showed a fig containing 9 plots from 9 steps in the Kohonen algorithm the regular square uniform data (by passing to fit 'toPlt=True'):




And the second un-uniform square data:



b. Circle shape

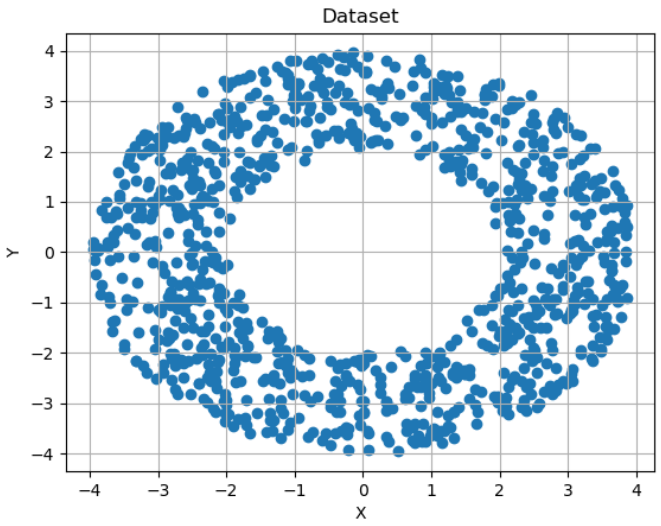
1) In the very same way we solved the circle problem:

```
# creating a donut uniform dataset
data_donut = create_circle_dataset()
create_data_plot(data_donut)
# activate the kohonen algorithm with 20 neurons and plotting the results
Create_Kohonen_network(data_donut, output_dim=20, learning_rate=0.5, num_epochs=1000, toPlt=False)
# activate the kohonen algorithm with 20 neurons and plotting the results
Create_Kohonen_network(data_donut, output_dim=200, learning_rate=0.5, num_epochs=1000, toPlt=False)

# creating the first donut un-uniform dataset
data_donut_unUni_1 = create_unUni_1_circle_dataset()
create_data_plot(data_donut_unUni_1)
# same as above
Create_Kohonen_network(data_donut_unUni_1, output_dim=20, learning_rate=0.5, num_epochs=1000, toPlt=False)
Create_Kohonen_network(data_donut_unUni_1, output_dim=200, learning_rate=0.5, num_epochs=1000, toPlt=False)
```

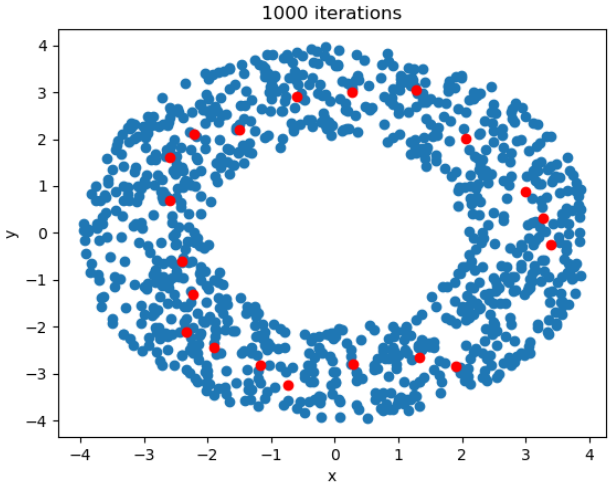
2) The create_circle_dataset function generates a synthetic circular dataset containing 100 points within a specified range. This dataset will be used to train the SOM.

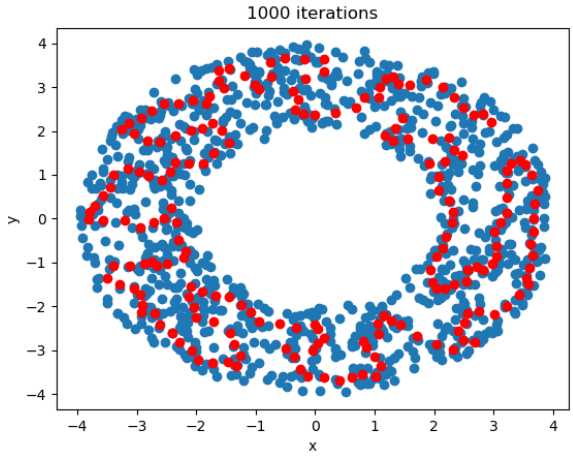
```
40  def create_circle_dataset():
41      dataset = []
42      while len(dataset) < 1000:  # Generate 100 points
43          x = np.random.uniform(-4, 4)
44          y = np.random.uniform(-4, 4)
45          distance_squared = x ** 2 + y ** 2
46          if 4 <= distance_squared <= 16:
47              dataset.append([x, y])
48      return np.array(dataset)
```

In this part of the code, a circular dataset is created using the create_circle_dataset function, and then a plot of the dataset is generated using the create_plot function.

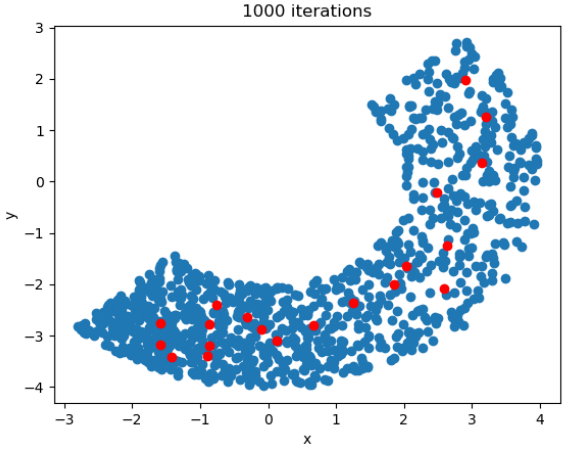


3) Next, we activated the kohonen algorithm with 20 and 200 neurons and plotted the result:

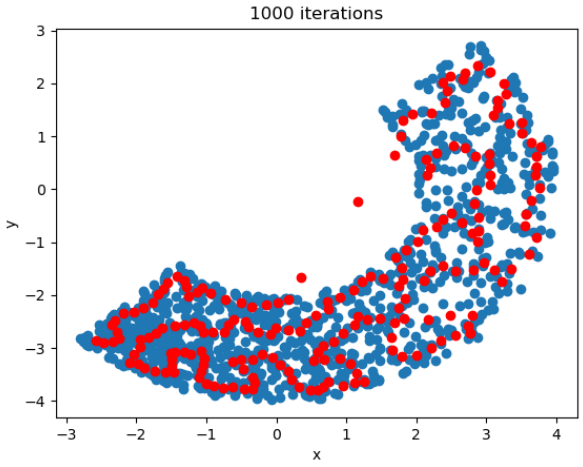



4) After creating the first donut un-uniform dataset the code creates two Kohonen networks with different numbers (20 and 200) of neurons, trains them using the dataset, and stores the resulting weights of the networks. The purpose is to compare the performance of the networks with different numbers of neurons in mapping and clustering the circular dataset.
20 neurons:

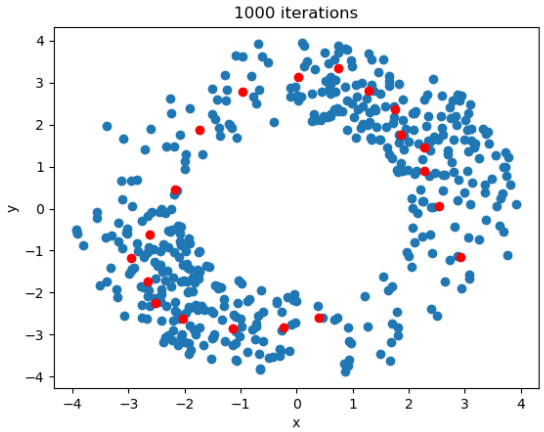


And 200 neurons:

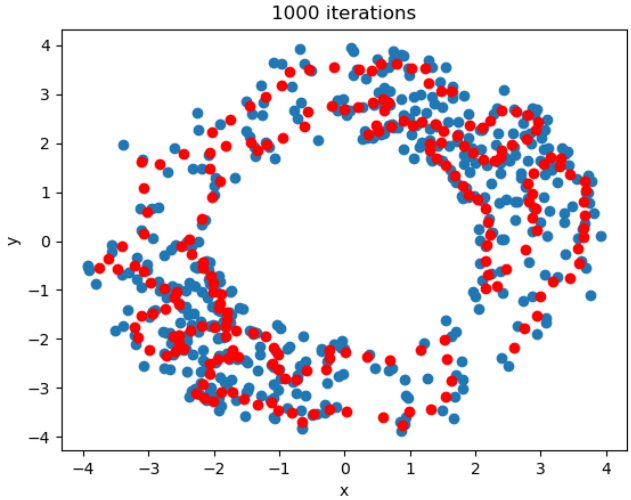



The resulting dataset will consist of 1000 data points that are distributed in a circular pattern within the specified radius range. The points will be closer to the origin and sparser as the distance from the origin increases.

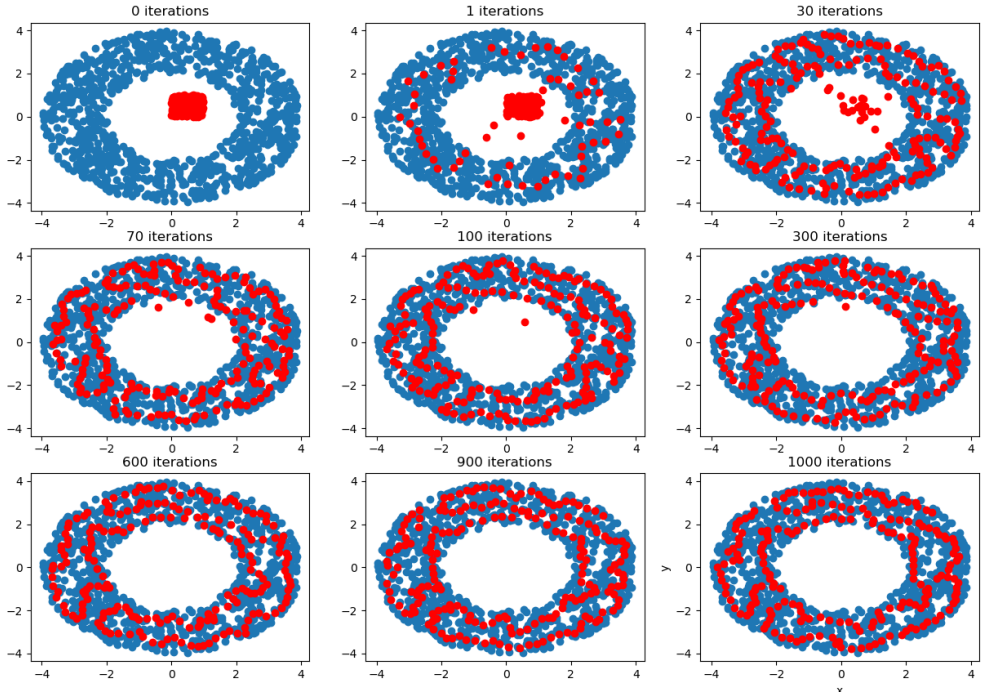
5) Creating the second non-uniform dataset on a circle (donut) and trining 20 neurons:



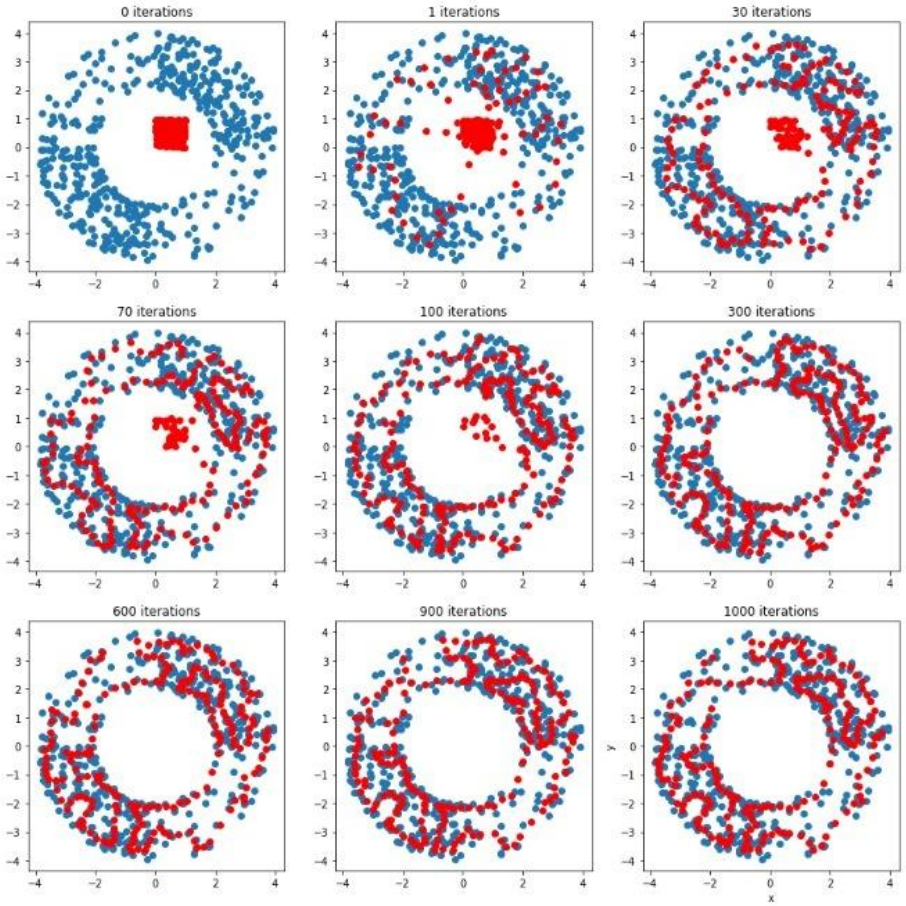

And 200 neurons:



6) In the end we showed a fig containing 9 plots from 9 steps in the Kohonen algorithm the regular square uniform data (by passing to fit 'toPlt=True'):




And the second un-uniform square data



# 4. Part B - Code review:

a) Monkey Hand

1. In this part we defined the class of Kohonen a little bit differently, which represents the Self-Organizing Map (SOM) using the Kohonen algorithm.

```
def __init__(self, learning_rate=0.1, neurons_amount=[100]):
    self.learning_rate = learning_rate
    self.neurons_amount = neurons_amount
    self.data = None
    self.neurons = []
    self.radius = max(self.neurons_amount[0], len(self.neurons_amount)) / 2
    self.lamda = None
```

2. Class constructor (__init__)
   The constructor initializes the Kohonen class with the provided parameters, including the learning rate (learning_rate) and the number of neurons in each layer (neurons_amount). It also initializes other variables such as the input data, the neurons, the radius, and the time constant (lamda):

   - learning_rate: According to this parameter the network calculates how much the neurons should change (the radius is getting learning_rate in each iteration of the fit function).
   - neurons_amount: An array that represents the layers of the network such that the length of it is the number of layers and the i'th value is the number of neurons in the i'th level.
   - data: (2D array) The data that the network will fit to.
   - neurons: (3D array) The first dimension contains n arrays that represent layers, each layer contains m arrays that represent neurons and each neuron contains the weights of this neuron. (the neurons length is equal to the length of the data instances)
   - radius: A tuning parameter, according to it the network calculates which neurons should change and how much.(the radius is getting smaller in each iteration of the fit function).
   - lamda: A time constant used to determine how to decrease the radius and the learning rate in each iteration.

3. fit method:
   This method trains the SOM using the provided dataset (data_set). It first initializes the neurons with random weights. Then, in each iteration, it selects a random vector from the dataset, finds the closest neuron to that vector using the Euclidean distance, and updates the weights of the neurons based on the learning rate and radius. The learning rate and radius decrease over time using the time constant (lamda). The method also includes code for plotting the network's progress at certain intervals.

```
48      def fit(self, data_set, iteration=10000):
49
50          # initializing the data and time constant.
51          self.data = np.array(data_set)
52          self.lamda = iteration / np.log(self.radius)
53
54          # initializing the neurons with random weights.
55          for layer in range(len(self.neurons_amount)):
56              self.neurons.append([])
57              for n in range(self.neurons_amount[layer]):
58                  weights = []
59                  for t in range(len(data_set[0])):
60                      weights.append(random.uniform(0, 1))
61                  self.neurons[layer].append(weights)
62          self.neurons = np.array(self.neurons)
63          # starting the fitting process.
64          for i in range(iteration):
65              # selecting random vector from the given data.
66              vec = self.data[int(random.uniform(0, len(self.data)))]
67              # find which neuron is the closest to the vector.
68              nn = self.nearest_neuron(vec)
69              # updating the learning rate and the radius.
70              curr_learning_rate = self.learning_rate * np.exp(-i / self.lamda)
71              curr_radius = self.radius * np.exp(-i / self.lamda)
72              # going over the neurons in each layer and compute how to change each neuron.
73              for j in range(len(self.neurons)):
74                  for n in range(len(self.neurons[j])):
75                      curr_neuron = self.neurons[j][n]
76                      d = np.linalg.norm(np.array(nn) - np.array([j, n]))
77                      neighbourhood = np.exp(- (d ** 2) / (2 * (curr_radius ** 2)))
78                      self.neurons[j][n] += curr_learning_rate * neighbourhood * (vec - curr_neuron)
79              # plotting the network to track progress.
80              if (i % 1000 == 0) or i == iteration - 1:
81                  if len(self.neurons_amount) == 1:
82                      self.plot1D(i)
83                  else:
84                      self.plot2D(i)
```

4. refit method:
   This method is similar to the fit method but does not initialize the neurons with random weights. It can be used to continue training the SOM with new data.

```
87      def refit(self, data, iteration=1000):
88          self.data = np.array(data)
89          self.lamda = iteration / np.log(self.radius)
90          for i in range(iteration):
91              vec = self.data[int(random.uniform(0, len(self.data)))]
92              nn = self.nearest_neuron(vec)
93              # nearest_n = self.neurons[nn[0]][nn[1]]
94              curr_learning_rate = self.learning_rate * np.exp(-i / self.lamda)
95              curr_radius = self.radius * np.exp(-i / self.lamda)
96              for j in range(len(self.neurons)):
97                  for n in range(len(self.neurons[j])):
98                      curr_neuron = self.neurons[j][n]
99                      d = np.linalg.norm(np.array(nn) - np.array([j, n]))
100                     neighbourhood = np.exp(- (d ** 2) / (2 * (curr_radius ** 2)))
101                     self.neurons[j][n] += curr_learning_rate * neighbourhood * (
102                                 vec - curr_neuron)  # dist(curr_neuron, vec)
103             if (i % 1000 == 0) or i == iteration - 1:
104                 if len(self.neurons_amount) == 1:
105                     self.plot1D(i)
106                 else:
107                     self.plot2D(i)
```

5. Nearest neuron method:
   This method calculates the closest neuron to a given input vector using the Euclidean distance. It returns the index of the closest neuron in the neurons array.

```
115     def nearest_neuron(self, vec):
116         """
117         This function checks find which neuron is the closest to a given vector using Euclidean distance.
118         :param vec: Input vector
119         :return: A tuple that contains the index of the closest neuron to the vector in self.neurons array.
120         """
121         min_dist = np.inf
122         loc = None
123         for i in range(len(self.neurons)):
124             for n in range(len(self.neurons[i])):
125                 curr_neuron = self.neurons[i][n]
126                 curr_dist = dist(curr_neuron, vec)
127                 if min_dist > curr_dist:
128                     loc = (i, n)
129                     min_dist = curr_dist
130         return loc
```

6. Plotting methods:
   The code includes two plotting methods, plot1D and plot2D, for visualizing the SOM during training. The plot1D method plots the network with 1 layer, and the plot2D method plots the network with multiple layers.

```
132     def plot1D(self, t):
133         """
134         this function plot a network with 1 layer in the t'th iteration.
135         :param t: the current iteration
136         :return:
137         """
138         xs = []
139         ys = []
140         for i in range(self.neurons.shape[0]):
141             for j in range(self.neurons.shape[1]):
142                 xs.append(self.neurons[i, j, 0])
143                 ys.append(self.neurons[i, j, 1])
144         fig, ax = plt.subplots()
145         ax.scatter([xs], [ys], c='r')
146         ax.set_xlim(0, 1)
147         ax.set_ylim(0, 1)
148         ax.plot(xs, ys, 'b-')
149         ax.scatter(self.data[:, 0], self.data[:, 1], alpha=0.3)
150         plt.title("Plot1D Iteration No. " + str(t))
151         # plt.savefig("plot1D iteration " + str(t) + ".png")
152         plt.show()
```

7. dist function:
   This function calculates the Euclidean distance between two vectors using the numpy library.

```
184 def dist(vec, weights):
185     return np.sqrt(((vec - weights) ** 2).sum())
```

8. Process image function:
   This function processes an image by reading it, converting it to grayscale, resizing it, and extracting the non-white points from the image. The points are normalized to a range of [0, 1] based on the maximum coordinates.

```python
def process_image(image_path):
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    image = cv2.resize(image, (0, 0), fx=0.5, fy=0.5)
    points = np.argwhere(image != 255).astype(np.float32)
    max_coords = points.max(axis=0) * 1.0
    points[:, 0] /= max_coords[0]
    points[:, 1] /= max_coords[1]
    return points
```

9. main function:
   The main function demonstrates the usage of the Kohonen class. It processes an image of a hand using the process_image function, creates an instance of the Kohonen class, and trains the SOM with the hand points. It then refits the SOM with another set of hand points from a different image. Finally, it displays the second monkey hand image using plt.imshow.

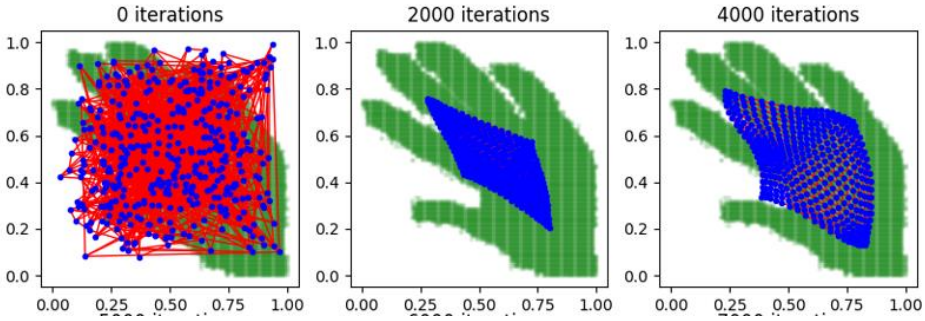
   By running this code, you would train a SOM using the Kohonen algorithm to learn and represent the patterns in the provided hand images. The plotting functions allow you to visualize the progress and final representation of the SOM.

```python
def main():
    hand_points = process_image("monkeyHand.jpg")
    layers = (np.ones(20) * 20).astype(int)
    ko = Kohonen(neurons_amount=layers, learning_rate=0.4)
    ko.fit(hand_points, iteration=10000)

    hand2_points = process_image("FourFingersMonkeyHand.jpg")
    ko.refit(hand2_points)

    hand2 = cv2.imread("FourFingersMonkeyHand.jpg")
    plt.imshow(hand2)
    pass

if __name__ == '__main__':
    main()
```

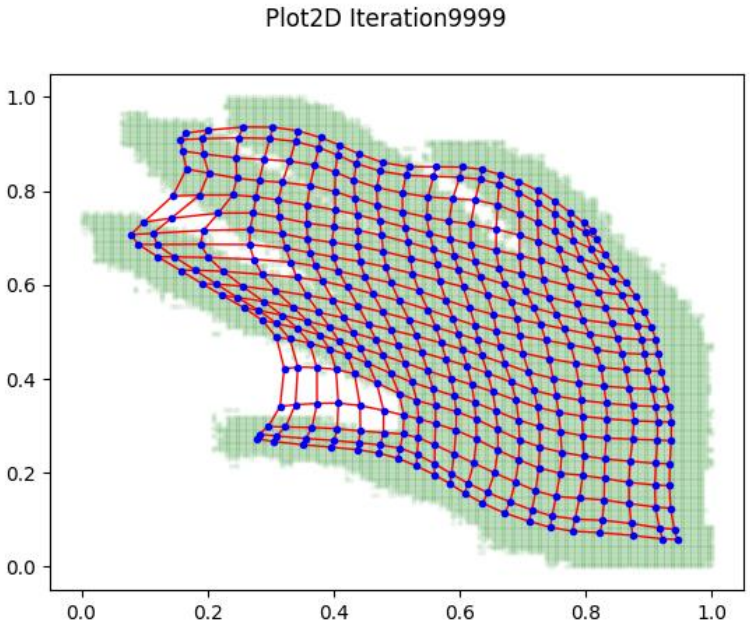
10. Input & Output:

We trained the model on an image of monkey's hand we took from the internet, and we cut it with the help of 'paint':

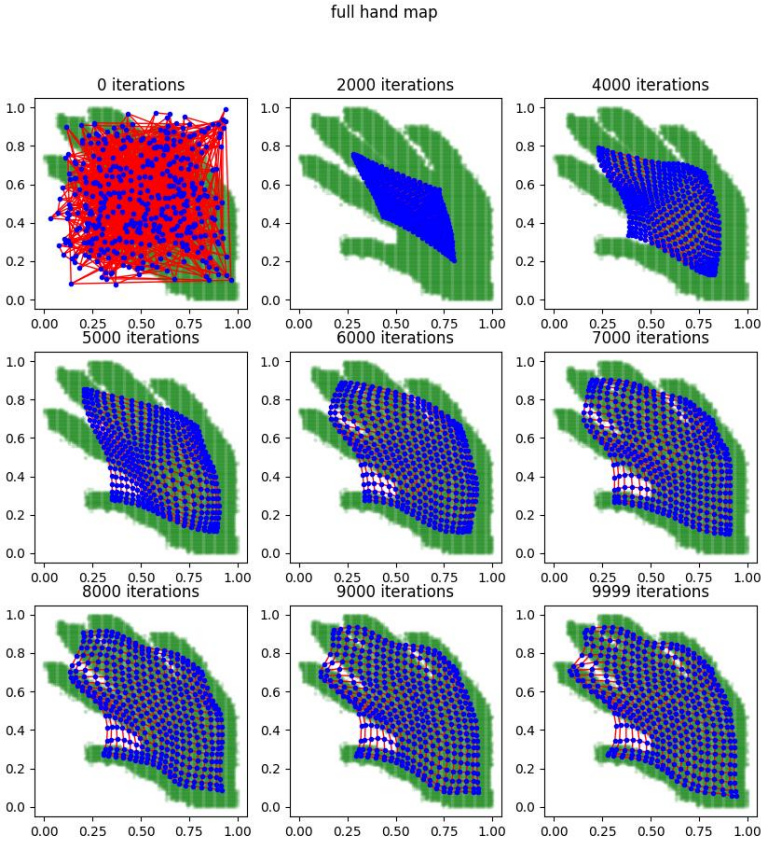
The code takes the points in this shape as the data points. Afterward it takes 400 neurons (20*20 map) and fit the map into the shape.
The code process goes like this:




Such that after 10,000 iterations the output is:

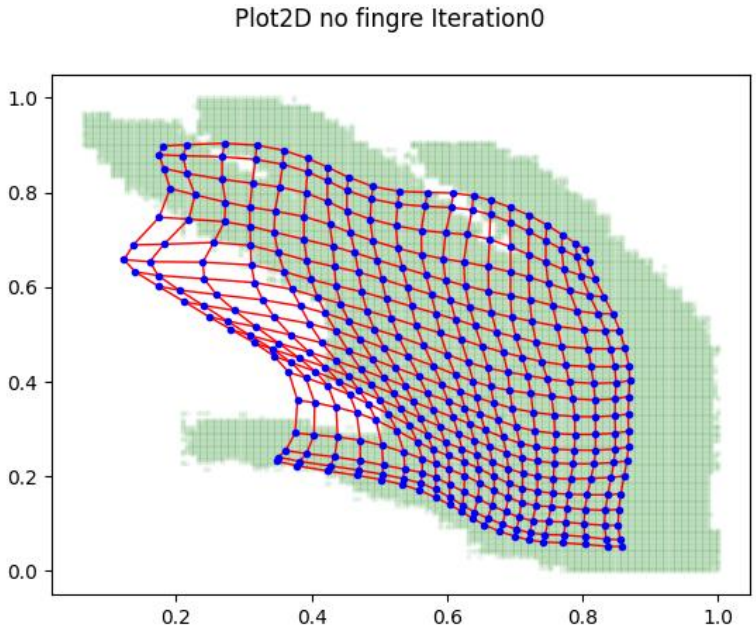


The whole process:




11. In the second section of this part we cut manually one finger of the monkey's hand and train the model from the same point it stopped from the data of the 5 fingers in purpose to see if the code rearrange the map correctly:

And indeed we can see that the map is rearranged properly, such that the map before the re-fit is in the previous mode (of the 5 fingers):




And after 999 iterations it comes to this: