Kohonen Self Organizing Maps (K-SOM) Algorithm - Neural Networks

Submitting:

- ★ **Yahalom Chasid - 208515577**
- ★ **Or Yitshak - 208936039**
- ★ **Matan Yarin Shimon - 314669342**
- ★ **Netanel Levine - 312512619**

About

This project will train a neural network to classify points to the nearest neuron(cluster), according to the Kohonen algorithm.
A **self-organizing map** (**SOM**) or **self-organizing feature map** (**SOFM**) is an unsupervised machine learning technique used to produce a low-dimensional (typically two-dimensional) representation of a higher dimensional data set while preserving the topological structure of the data. For example, a data set with *p* variables measured in *n* observations could be represented as clusters of observations with similar values for the variables. These clusters then could be visualized as a two-dimensional "map" such that observations in proximal clusters have more similar values than observations in distal clusters. This can make high-dimensional data easier to visualize and analyze.
A **SOM** is a type of artificial neural network. Still, it is trained using competitive learning rather than the error-correction learning (e.g., backpropagation with gradient descent) used by other artificial neural networks.

- ❖ See more on Wikipedia - Self-organizing map
- ❖ See more on Wikipedia - Hybrid Kohonen

Introduction

This project contains two main parts.

A. **Part A**

Part A's goal is to take 4 different datasets containing 2D points (x,y) and train a map of neurons (clusters) that will adapt to the current dataset. For each of the datasets, we will train the neurons twice:

- ➢ 1D map - an array of the neurons.
- ➢ 2D map - a matrix of the neurons.

a. **A.1-** In this part, the dataset is - $\{(x,y) \mid \ x,y \in [0,1]\} \ - \ all\ the\ real\ points\ where\ x,y\ between\ 0\ to\ 1$
The map contains 100 neurons.
The points are uniformly distributed - $x,y \sim U\{[0,1]\}$

b. **A.2-** In this part the dataset is the same as **A.1**.
The amount of neurons is the same as **A.1**.
The x value of each point is normally distributed (also known as gaussian) - $x \sim N(\mu, \sigma^2)$
The y value of each point is uniformly distributed - $y \sim U\{[0,1]\}$

c. **A.3-** In this part the dataset is the same as **A.1 and A.2**.
The amount of neurons is the same as **A.1 and A.2**.
The x value of each point is normally distributed (also known as gaussian) - $x \sim N(\mu, \sigma^2)$
The y value of each point is normally distributed (also known as gaussian) - $y \sim N(\mu, \sigma^2)$

d. **A.4-** In this part the we will try to train the neurons to fit to a **"donut"** shape.
The dataset is -
$\{(x,y) \mid \ x,y \in [0,1]\} \ \cup \{(x,y) \mid \ 0.15^2 \leq (x-0.5)^2 + \ (y-0.5)^2 \leq \ 0.3^2\}$
The amount of neurons is 30.
The points are uniformly distributed - $x,y \sim U\{[0,1]\}$.

B. **Part B**

Part B's goal is to take a dataset containing 2D points (x,y) that looks like a **shape of a hand** and train a map of neurons (clusters) that will adapt to the current dataset shape. After the map is fitted to this dataset we then change the dataset and remove one of the fingers (by creating a new dataset without one finger) and performing a **re_fit** on the new dataset. A **re_fit** is almost the same as the regular **fit** with the difference in the starting neurons weights. While in **fit** the neurons start with random weights, in **re_fit** the starting weights are the weights of the neurons before we changed the dataset.
The purpose of this part is to see how the trained neurons (after they adapt to the full hand shape) adapt and "relize" that a finger is missing and now there is no need for any neuron to be in the area of the missing finger.

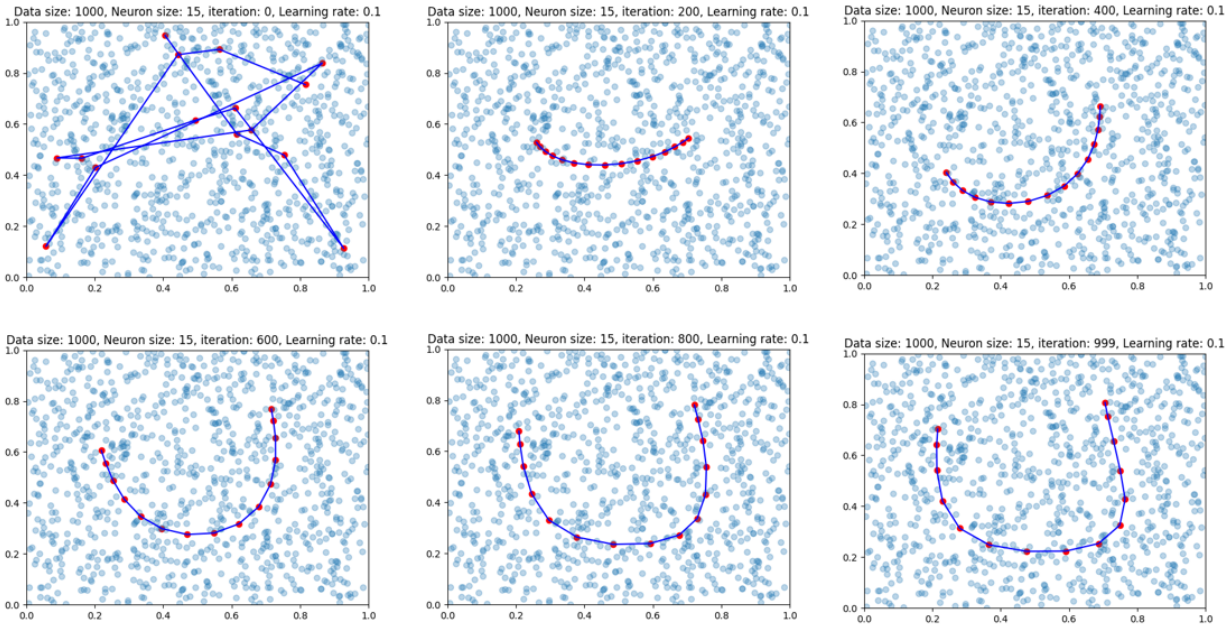
- The dataset is - $\{(x,y) \mid \ x,y \in [0,1]\} \ - \ all\ the\ real\ points\ where\ x,y\ between\ 0\ to\ 1$
- The map contains 225 neurons arranged in a $15x15\ matrix$.
- The points are uniformly distributed - $x,y \sim U\{[0,1]\}$

The project

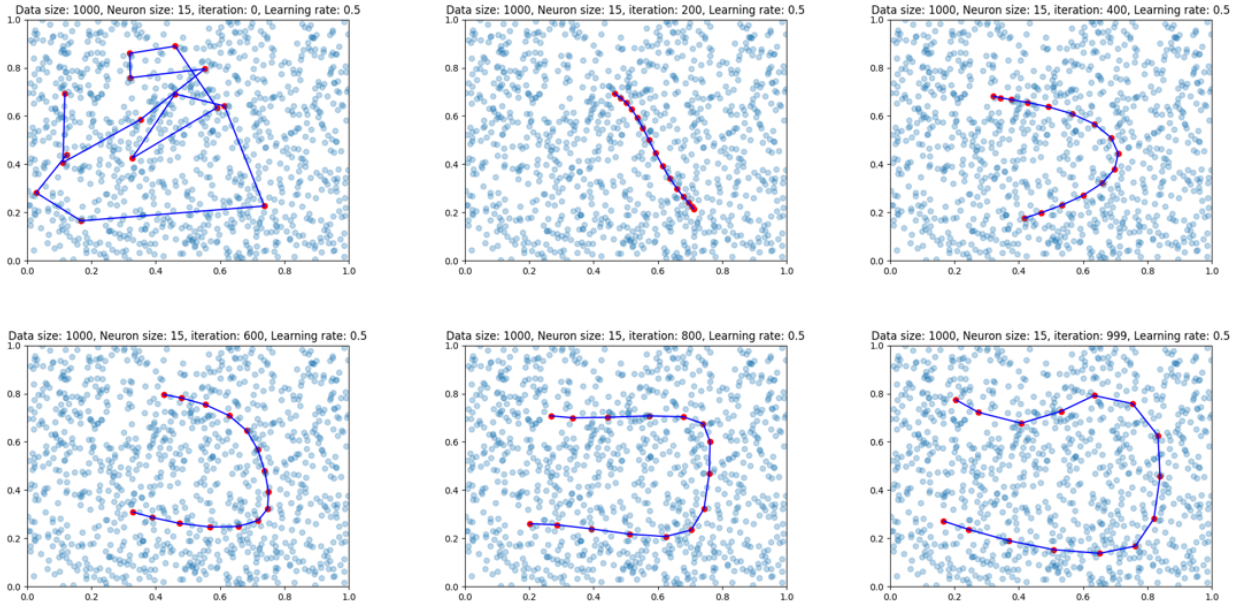
Part A.1

To illustrate the power of the **SOP** and the adaptivity of the neurons.

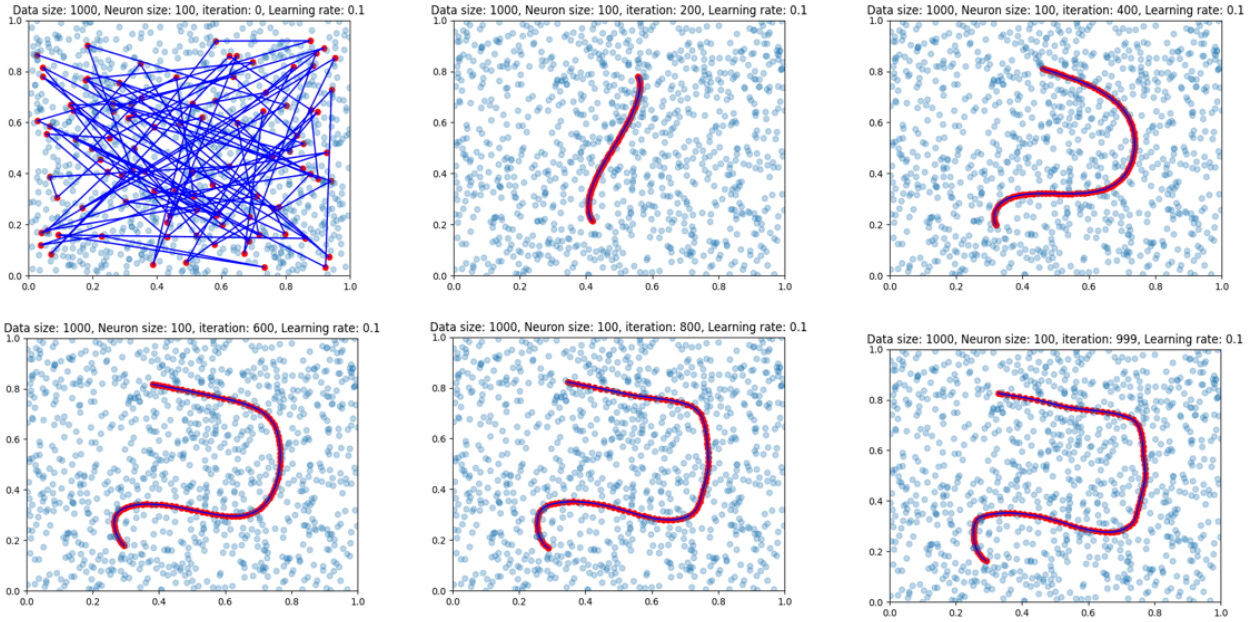
We will first start by training **15** neurons in a linear order (not 2D) with a starting learning rate of **0.1**. In each figure, we can see how the neurons keep changing and trying to adapt to the whole dataset. Each figure demonstrates the location of the neurons in the n iteration such that $n \in \{0, 200, 400, 600, 800, 999\}$



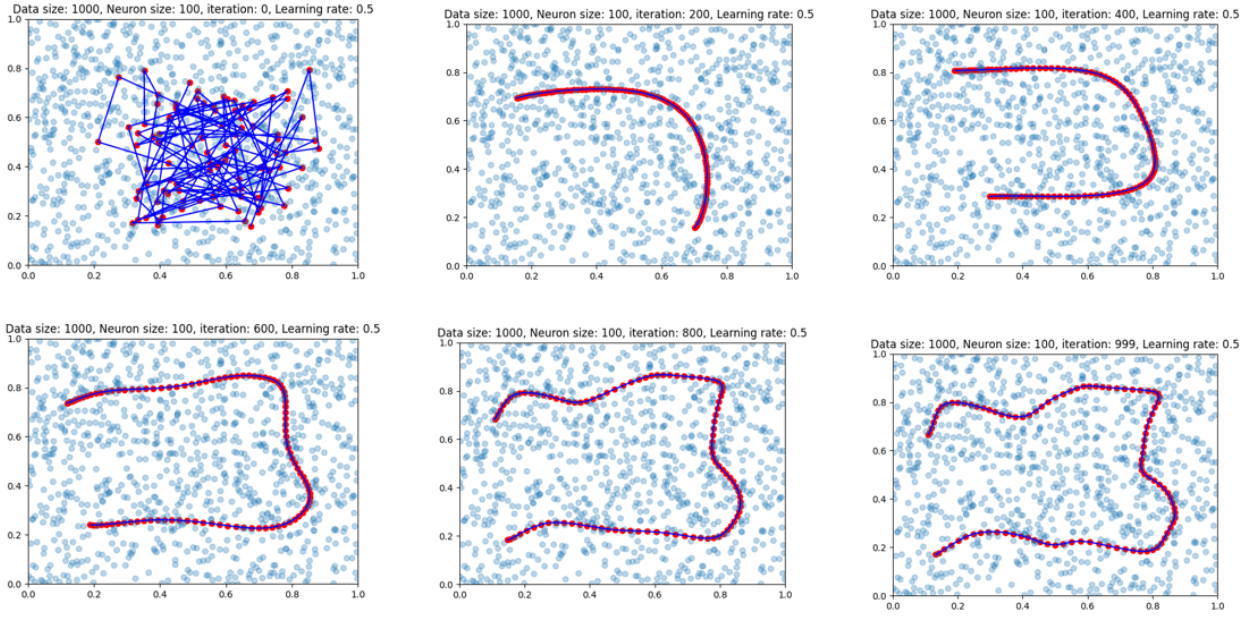
In the next figures, we changed the starting learning rate from **0.1** to **0.5**.



In the next figures, the number of neurons will be **100** in a linear order (not 2D) and with a starting learning rate of **0.1**.

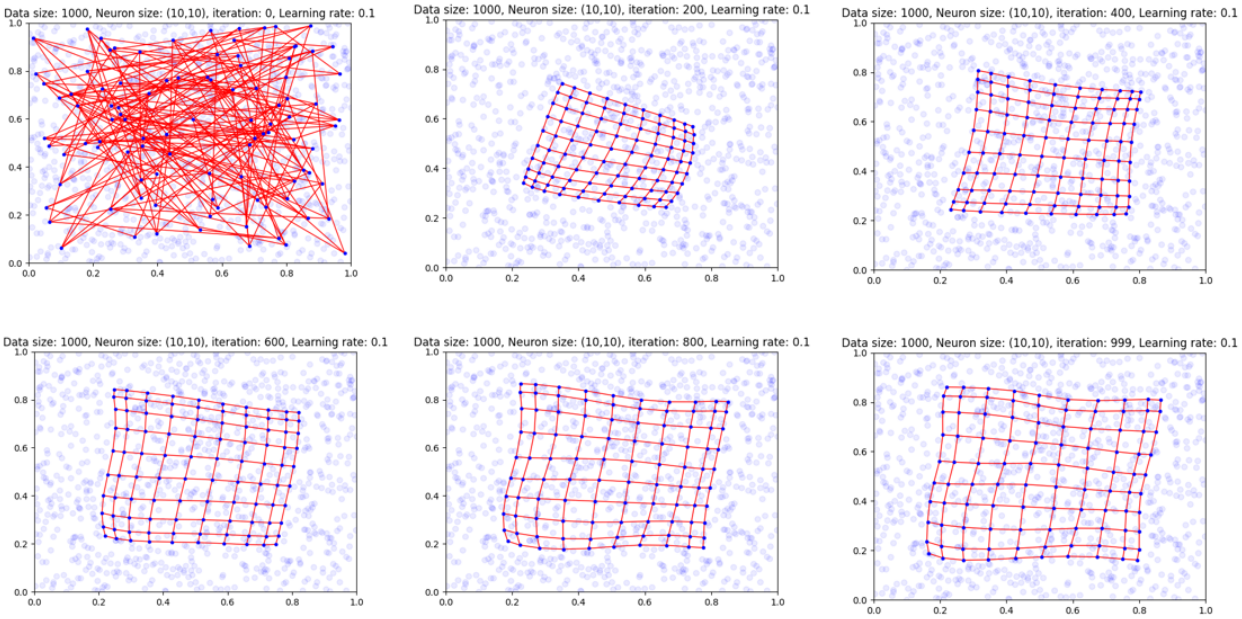


In the next figures, we changed the starting learning rate from **0.1** to **0.5**.

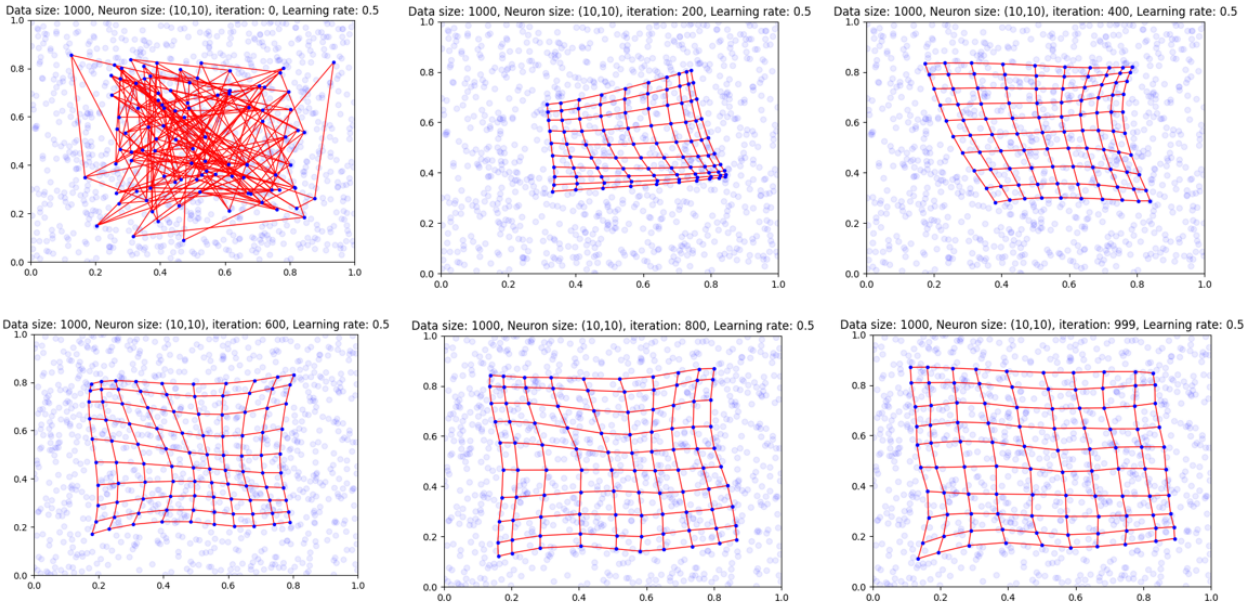


As we can see, even **100** neurons(in linear order) couldn't spread in a way that can reach the entire dataset in a good way.
This is why we use the 2D map of neurons.

In the next figures, the number of neurons will be **100** as a 2D map(matrix) and with a starting learning rate of **0.1**.



In the next figures, we changed the starting learning rate from **0.1** to **0.5**.



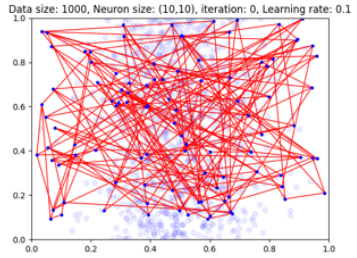
To conclude this part, we saw a significant difference when changing the structure of the neurons from a 1D array into a 2D array.
We can see that the final result (as shown in the bottom right figures) actually spread in a way that it seems that the euclidean distance between each **point** to its nearest **neuron** for all the points in the dataset is approximately equal.

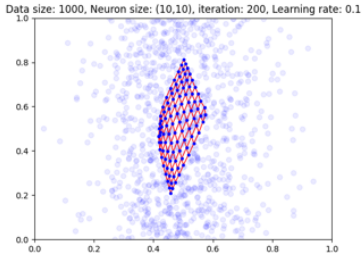
Part A.2

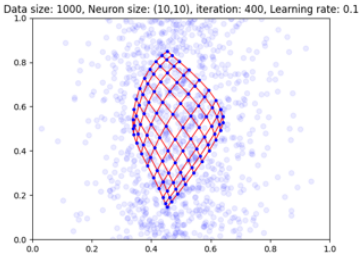
In this part the number of neurons is **100.**

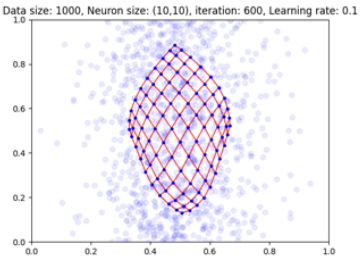
The x value of each point is normally distributed (also known as gaussian) - $x \sim N(\mu, \sigma^2)$
The y value of each point is uniformly distributed - $y \sim U\{[0,1]\}$
In each figure, we can see how the neurons keep changing and trying to adapt to the whole dataset.
Each figure demonstrates the location of the neurons in the n iteration such that
$n \in \{0, 200, 400, 600, 800, 999\}$

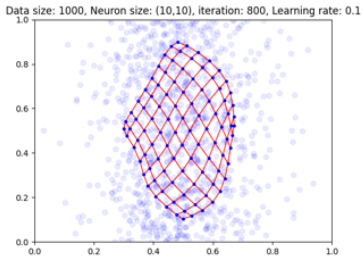
Starting with a learning rate of **0.1**

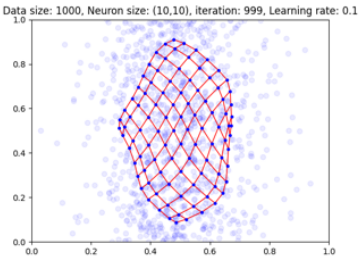




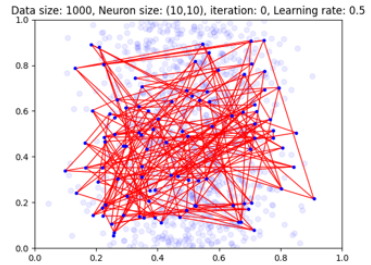


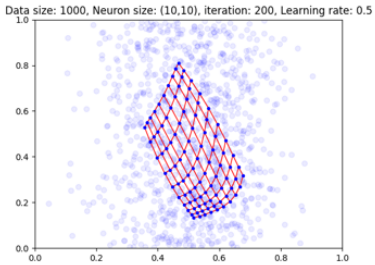


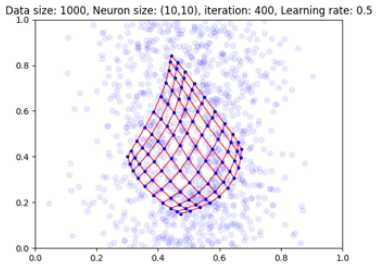


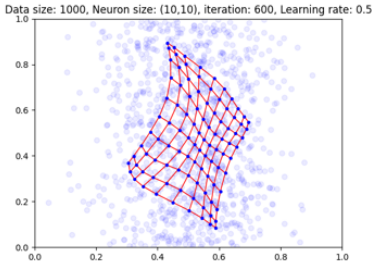


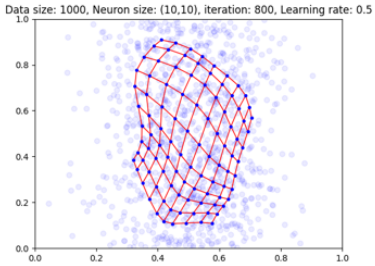
In the next figures, we changed the starting learning rate from **0.1** to **0.5**.

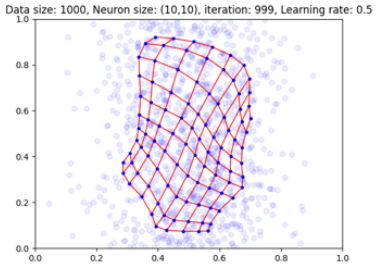












To conclude this part, we can see how well the final result (as shown in the bottom right figures) actually spread in a way that it seems that the euclidean distance between each **point** to its nearest **neuron** for all the points in the dataset is approximately equal.
In addition, we can see a difference between starting with a learning rate of **0.1** and starting with a learning rate of **0.5** and we can how the map increases spreading when doing more iterations.

Part A.3

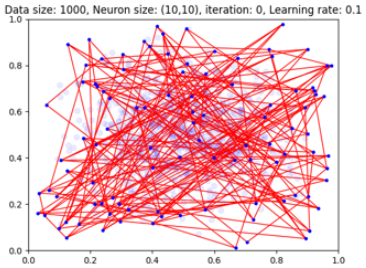
In this part the number of neurons is **100.**

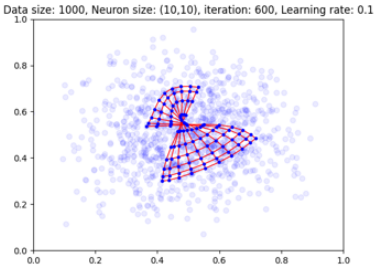
The x value of each point is normally distributed (also known as gaussian) - $x \sim N(\mu, \sigma^2)$

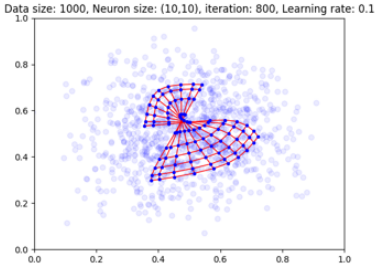
The y value of each point is normally distributed (also known as gaussian) - $y \sim N(\mu, \sigma^2)$

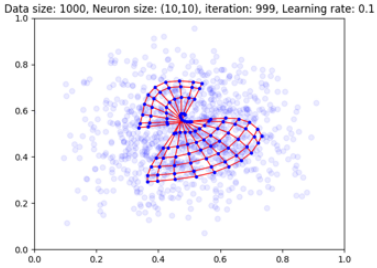
In each figure, we can see how the neurons keep changing and trying to adapt to the whole dataset.
Each figure demonstrates the location of the neurons in the n iteration such that
$n \in \{0, 200, 400, 600, 800, 999\}$

Starting with a learning rate of **0.1**

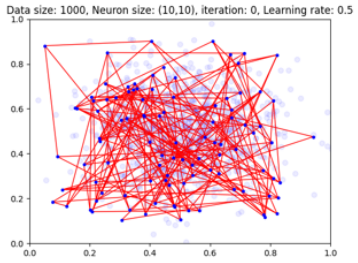


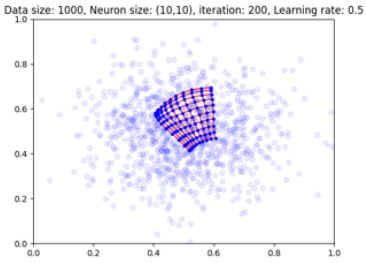






In the next figures, we changed the starting learning rate from **0.1** to **0.5**.





To conclude this part, we can see how well the final result (as shown in the bottom right figures) actually spread in a way that it seems that the euclidean distance between each **point** to its nearest **neuron** for all the points in the dataset is approximately equal.
In addition, we can see a difference between starting with a learning rate of **0.1** and starting with a learning rate of **0.5** and we can how the map increases spreading when doing more iterations.

Part A.4

In this part the we will try to train the neurons to fit to a **"donut"** shape.

The dataset is - $\{(x,y) \mid \ x,y \in [0,1]\} \ \cup \{(x,y) \mid 0.15^2 \leq (x-0.5)^2 + (y-0.5)^2 \leq 0.3^2\}$

The amount of neurons is **30**.

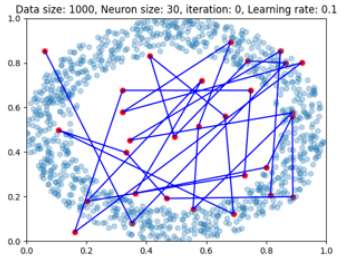
The points are uniformly distributed - $x,y \sim U\{[0,1]\}$.

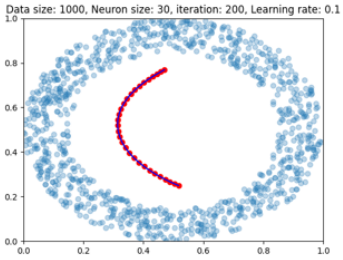
In each figure, we can see how the neurons keep changing and trying to adapt to the whole dataset.

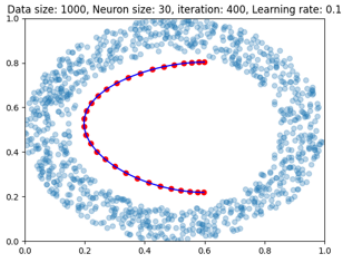
Each figure demonstrates the location of the neurons in the n iteration such that

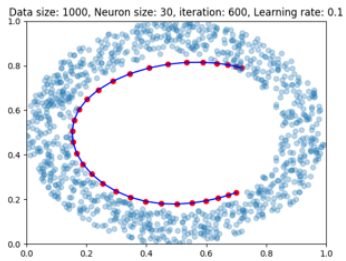
$n \in \{0, 200, 400, 600, 800, 999\}$

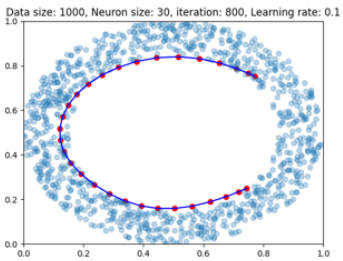
Starting with a learning rate of **0.1** and the neurons structured in a 1D array.

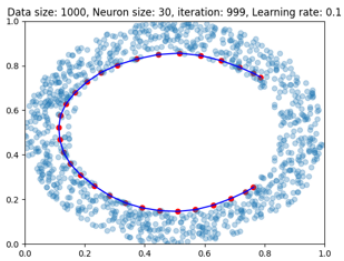




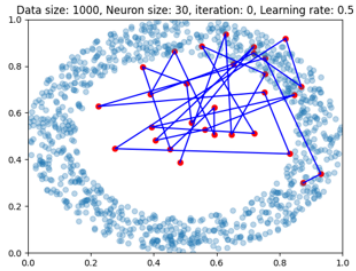


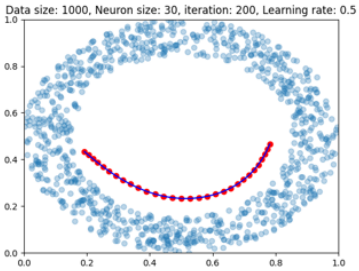


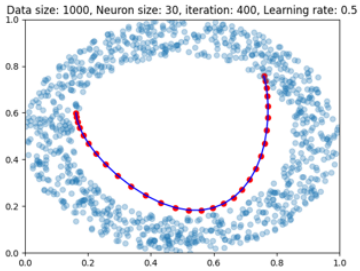


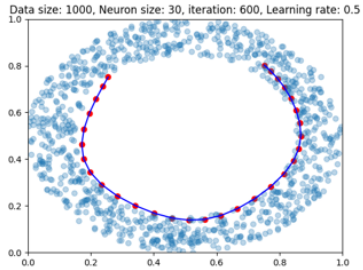


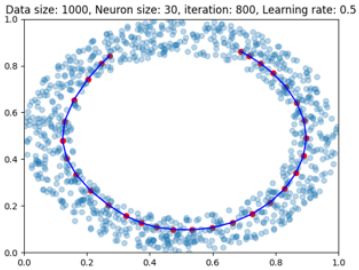
In the next figures, we changed the starting learning rate from **0.1** to **0.5**.

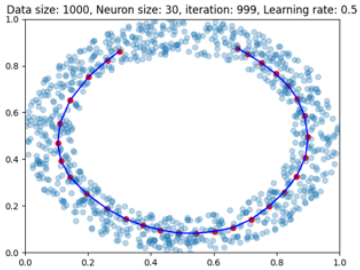








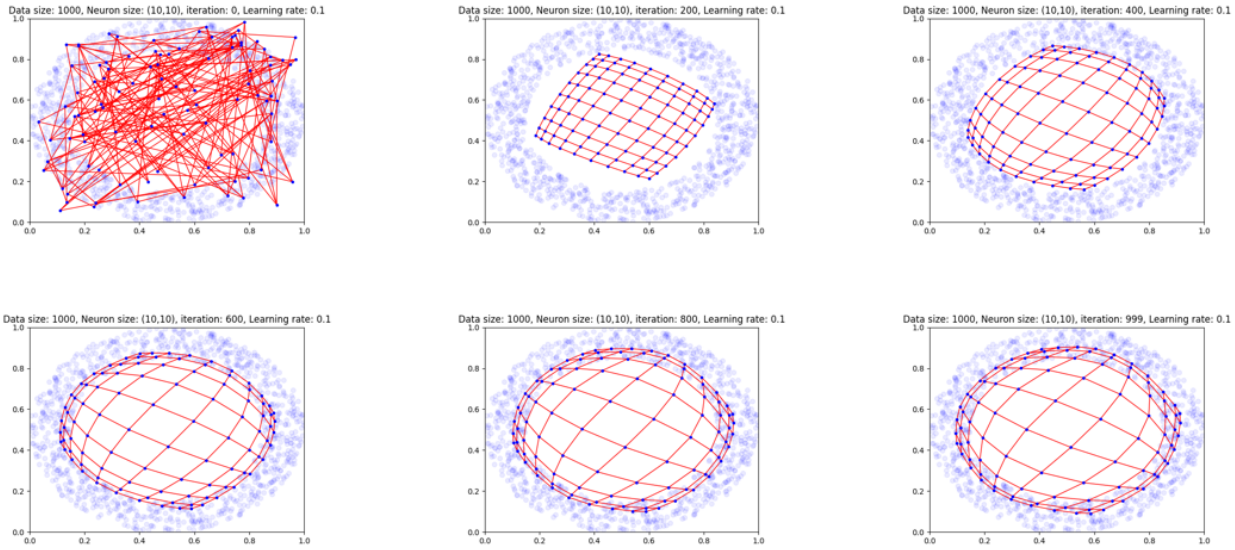




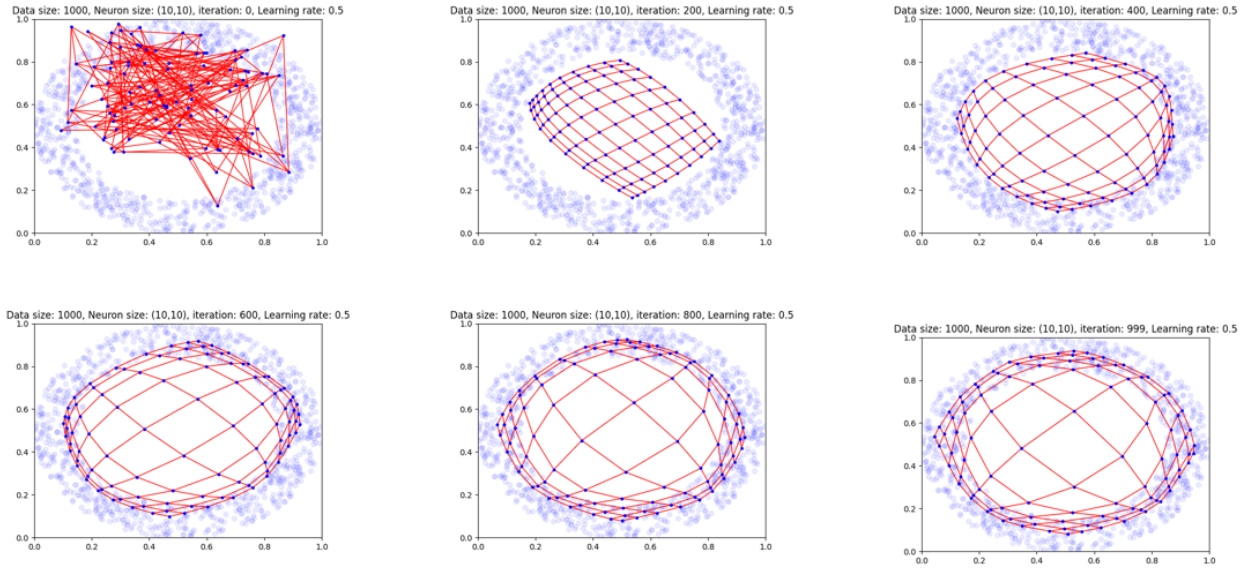
We can see that the neurons tried to create a circle in order to spread wisely between the entire dataset.

Next we will see the difference when those **30** neurons are ordered in a 2D array.

Starting with a learning rate of **0.1** and the neurons structured in a 2D array(matrix/map).



In the next figures, we changed the starting learning rate from **0.1** to **0.5**.

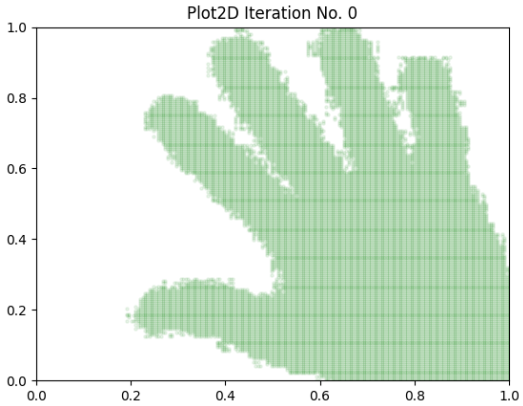


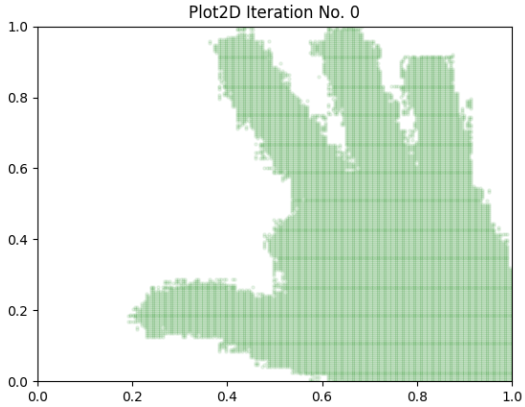
To conclude this part and all of part A, we can see how well the final result (as shown in the bottom right figures) actually spread in a way that it seems that the euclidean distance between each **point** to its nearest **neuron** for all the points in the dataset is approximately equal - with the map the neurons construct a sort of a ball-shaped structure that allows them to keep close as possible to all the points. In addition, we can see a difference between starting with a learning rate of **0.1** and starting with a learning rate of **0.5** and we can how the map increases spreading when doing more iterations.

Part B

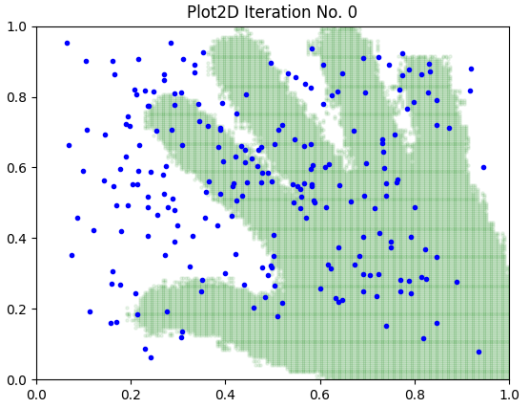
In this part, we took two photos of a human hand. One with 5 fingers and one with 4 fingers.

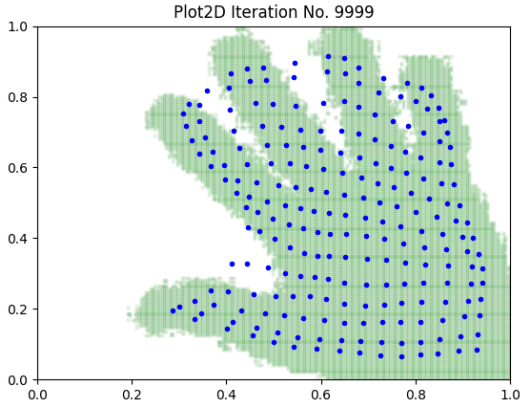
Then we create a dataset of points that looks like the shape of the hands above.



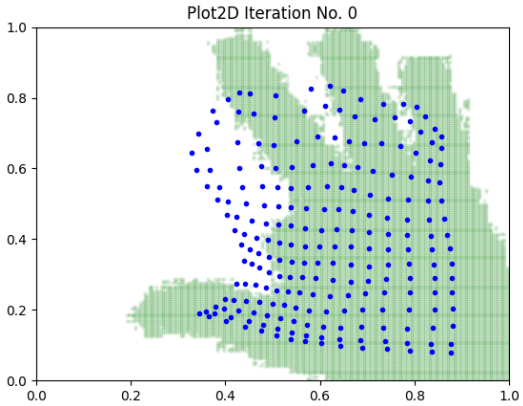


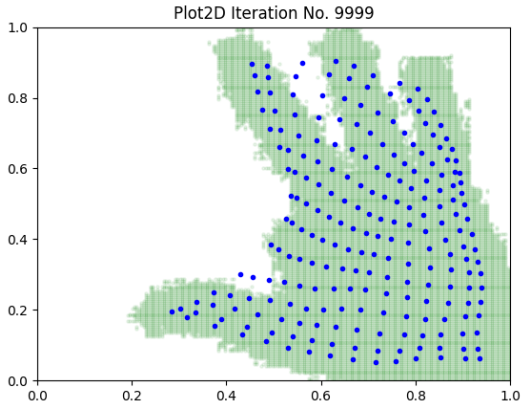
Next, let's look at the beginning point of the map and the ending point of the map.





The same for the 4 fingers dataset - The difference here is that the starting weights are from the last map.

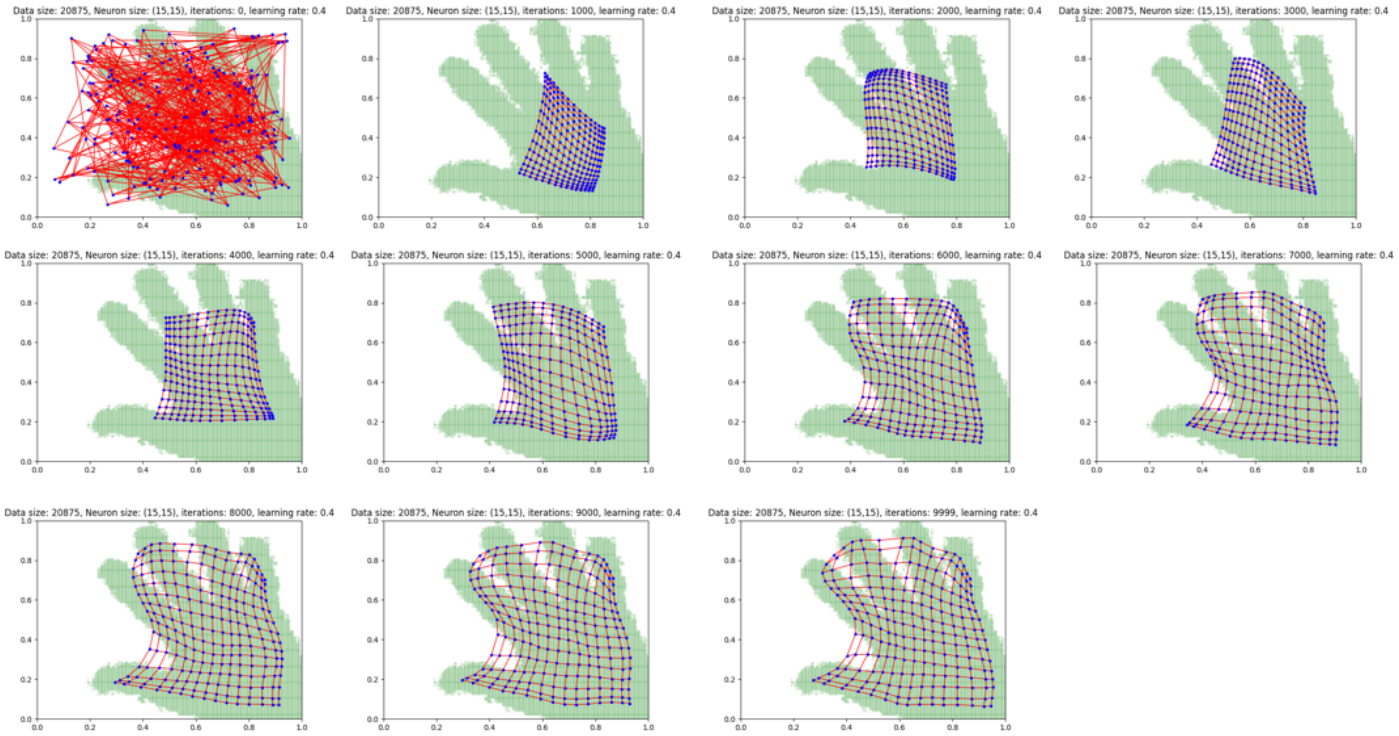




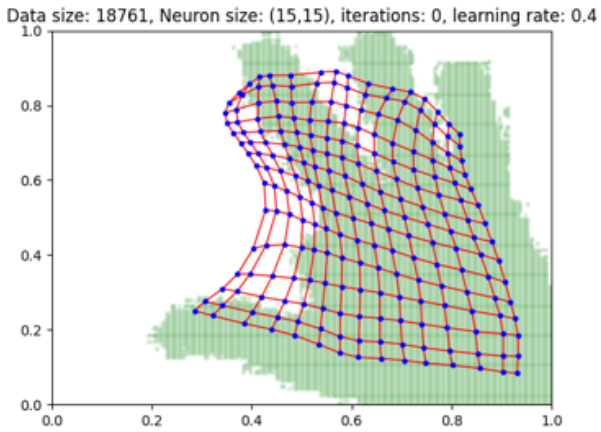
As we see, from the beginning of the 4 fingers dataset it's clear that there are some neurons where the 5th finger was and with the iterations, they adapt the fact that the finger is missing as we can see in the 10000 iteration plot.

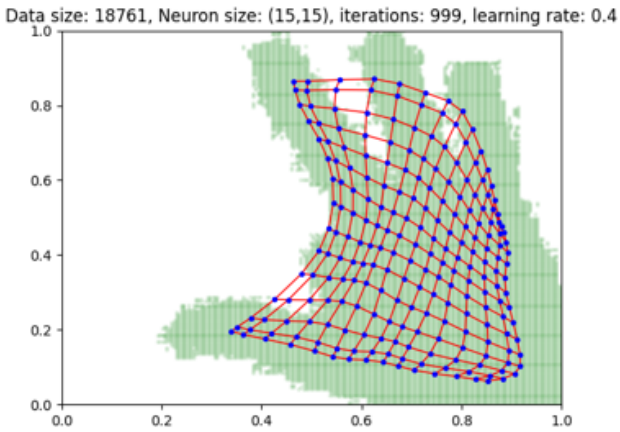
Now, we will show the progress of the network through the iterations.

Starting with the 5 fingers dataset.
We chose a starting learning rate of **0.4** and the amount of iterations is **10000,** we plotted the current map every **1000** iterations.



Now, we take the current weights of the neurons from the 5 fingers dataset and they will be the starting weights for the 4 fingers dataset. Due to the fact, that the neurons were already trained it took them **1000** Iterations to adapt to this dataset while the first dataset needed **10000** to achieve the same goal.





To conclude this part, we saw that the starting points (starting weights) of the neurons are important and can decrease the number of iterations if the starting points are initiated in a smart way (not always achievable). And that the neurons are mutable to changes in the data - meaning they adapt well.

Code

The files are:

- **Kohonen.py - file of the Kohonen class where we create the map fit and draw.**
- **Main.py - file for creating the data and running the script of all the parts.**

```python
import random
import numpy as np
import matplotlib.pyplot as plt


class Kohonen:
    """
    This class represent Self Organizing Map that implements the Kohonen algorithm.
    """

    def __init__(self, learning_rate=0.1, neurons_amount=[100]):
        """
        :param 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).
        :param neurons_amount: An array that represent 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.
        :param data: (2D array) The data that the network will fit to.
        :param neurons: (3D array) The  first dimension contains n arrays the represent layers, each layer contains
                        m arrays that represent neurons and each neurone contains the weights of this neurone.
                        (the neurons length is equal to the length of the data instances)
        :param 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).
        :param lamda: A time constant that used to the determine how to decrease the radius and the learning rate
                      in each iteration.
        """
        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

    def fit(self, data_set, iteration=10000):
        """
        This function fits the network to a given data. at first the functions initialize the neurons in random weights.
        then in each iteration the function choose a random vector from the data and check which neuron is the closest
        to the vector using Euclidean distance. according to the closest neuron, the learning rate and the radius the
        function compute how much each neuron should move towards the vector.
        :param data_set: (2D array) Array that contain all the data vectors.
        :param iteration: The number of the iteration to repeat the steps that mentioned above.
        :return:
        """
        # initializing the data and time constant.
```

```
self.data = np.array(data_set)
self.lamda = iteration / np.log(self.radius)

# initializing the neurons with random weights.
for layer in range(len(self.neurons_amount)):
    self.neurons.append([])
    for n in range(self.neurons_amount[layer]):
        weights = []
        for t in range(len(data_set[0])):
            weights.append(random.uniform(0, 1))
        self.neurons[layer].append(weights)
self.neurons = np.array(self.neurons)
# starting the fitting process.
for i in range(iteration):
    # selecting random vector from the given data.
    vec = self.data[int(random.uniform(0, len(self.data)))]
    # find which neuron is the closest to the vector.
    nn = self.nearest_neuron(vec)
    # updating the learning rate and the radius.
    curr_learning_rate = self.learning_rate * np.exp(-i / self.lamda)
    curr_radius = self.radius * np.exp(-i / self.lamda)
    # going over the neurons in each layer and compute how to change each neuron.
```

```
        for j in range(len(self.neurons)):
            for n in range(len(self.neurons[j])):
                curr_neuron = self.neurons[j][n]
                d = np.linalg.norm(np.array(nn) - np.array([j, n]))
                neighbourhood = np.exp(- (d ** 2) / (2 * (curr_radius ** 2)))
                self.neurons[j][n] += curr_learning_rate * neighbourhood * (vec - curr_neuron)
        # plotting the network to track progress.
        if (i % 1000 == 0) or i == iteration - 1:
            if len(self.neurons_amount) == 1:
                self.plot1D(i)
            else:
                self.plot2D(i)

def refit(self, data, iteration=1000):
    """
    This function do the same as the fit function without initializing the neurons with random weights.
    :param data:
    :param iteration:
    :return:
    """
    self.data = np.array(data)
```

```
    self.lamda = iteration / np.log(self.radius)
    for i in range(iteration):
        vec = self.data[int(random.uniform(0, len(self.data)))]
        nn = self.nearest_neuron(vec)
        # nearest_n = self.neurons[nn[0]][nn[1]]
        curr_learning_rate = self.learning_rate * np.exp(-i / self.lamda)
        curr_radius = self.radius * np.exp(-i / self.lamda)

        for j in range(len(self.neurons)):
            for n in range(len(self.neurons[j])):
                curr_neuron = self.neurons[j][n]
                d = np.linalg.norm(np.array(nn) - np.array([j, n]))
                neighbourhood = np.exp(- (d ** 2) / (2 * (curr_radius ** 2)))
                self.neurons[j][n] += curr_learning_rate * neighbourhood * (
                            vec - curr_neuron)  # dist(curr_neuron, vec)
        if (i % 1000 == 0) or i == iteration - 1:
            if len(self.neurons_amount) == 1:
                self.plot1D(i)
            else:
                self.plot2D(i)
```

```
def nearest_neuron(self, vec):
    """
    This function checks find which neuron is the closest to a given vector using Euclidean distance.
    :param vec: Input vector
    :return: A tuple that contains the index of the closest neuron to the vector in self.neurons array.
    """
    min_dist = np.inf
    loc = None
    for i in range(len(self.neurons)):
        for n in range(len(self.neurons[i])):
            curr_neuron = self.neurons[i][n]
            curr_dist = dist(curr_neuron, vec)
            if min_dist > curr_dist:
                loc = (i, n)
                min_dist = curr_dist
    return loc
```

```
def plot1D(self, t):                                                    ⚠18 ⚠5 ✔8
    """
    this function plot a network with 1 layer in the t'th iteration.
    :param t: the current iteration
    :return:
    """
    xs = []
    ys = []
    for i in range(self.neurons.shape[0]):
        for j in range(self.neurons.shape[1]):
            xs.append(self.neurons[i, j, 0])
            ys.append(self.neurons[i, j, 1])

    fig, ax = plt.subplots()
    ax.scatter([xs], [ys], c='r')
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    ax.plot(xs, ys, 'b-')
    ax.scatter(self.data[:, 0], self.data[:, 1], alpha=0.3)
    title = "Data size: "+str(self.data.shape[0])+", Neuron size: "+str(self.neurons.shape[0])+", iterations: "+str(t)+"
    plt.title(title)
    plt.savefig("plot1D - "+title+".png")
    plt.show()
```

```python
def plot2D(self, t):
    """
    this function plot a network with 1 layer in the t'th iteration.
    :param t: the current iteration
    :return:
    """
    neurons_x = self.neurons[:, :, 0]
    neurons_y = self.neurons[:, :, 1]
    fig, ax = plt.subplots()
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    for i in range(neurons_x.shape[0]):
        xh = []
        yh = []
        xs = []
        ys = []
        for j in range(neurons_x.shape[1]):
            xs.append(neurons_x[i, j])
            ys.append(neurons_y[i, j])
            xh.append(neurons_x[j, i])
            yh.append(neurons_y[j, i])
        ax.plot(xs, ys, 'r-', markersize=0, linewidth=1)
```

```python
        ax.plot(xh, yh, 'r-', markersize=0, linewidth=1)
    ax.plot(neurons_x, neurons_y, color='b', marker='o', linewidth=0, markersize=3)
    ax.scatter(self.data[:, 0], self.data[:, 1], c="g", alpha=0.05, s=5)
    title = "Data size: "+str(self.data.shape[0])+", Neuron size: ("+str(self.neurons.shape[0])+","+str(self.neurons.shape
    plt.title(title)
    plt.savefig("plot2D - "+title+".png")
    plt.show()


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