Kohonen Algorithm (SOM)

Taliya shitreet 314855099

Renana Rimon 207616830

9/06/22

Introduction :

In this project, we are required to implement the Kohonen (SOM) algorithm by using our own code.

A self-organizing map (SOM) algorithm is based on unsupervised, competitive learning. It provides a topology preserving mapping from the high dimensional space to map units. Map units, or neurons, usually form a two-dimensional lattice and thus the mapping is a mapping from high dimensional space onto a plane.

You can read more about SOM here: https://en.wikipedia.org/wiki/Self-organizing_map

Implantation:

The implantation is divided into two parts:

1. **1D - line of neurons (vector).**
2. **2D - grid of neurons (matrix).**

At each function explanation, we will describe both parts.

(according to the color: red=1D, Blue=2D)

initialize the SOM model

1. size:
   - 1D vector - (1, net_size)
   - 2D matrix - $(\sqrt{net\ size}, \sqrt{net\ size})$
2. value: random weights, represent the location of each neuron

```
def __init__(self, data: np.ndarray, net_size: int):
    """
    SOM: self organization map - weight of each neuron
    :param data: train_data
    :param net_size: num of neurons
    """
    rand = np.random.RandomState(0)
    self.SOM = rand.randint(0, 1000, (net_size, 2)).astype(float) / 1000
    self.data = data
    self.net_size = net_size
```

```
def __init__(self, data: np.ndarray, net_size: int):
    """
    SOM: self organization map - weight of each neuron
    :param data: train_data
    :param net_size: num of neurons
    """
    rand = np.random.RandomState(0)
    h = np.sqrt(net_size).astype(int)
    self.SOM = rand.randint(0, 1000, (h, h, 2)).astype(float) / 1000
    self.data = data
    self.net_size = net_size
```

train:

train SOM model - for each sample:
1. find BMU - Best Matching Unit (function below)
2. update weights (function below)
3. update learning rate
4. update radius

-same process for both 1D & 2D-

```
def train_SOM(self, learn_rate=.9, radius_sq=1,                    ⚠3 ✓3
              lr_decay=.1, radius_decay=.1, epochs=10):
    """
    train SOM model - for each sample:
        1. find BMU
        2. update weights
        3. update learning rate
        4. update radius
    :param lr_decay: Rate of decay of the learn_rate
    :param radius_decay: Rate of decay of the radius
    :param epochs: num of iteration
    :return:
    """
    rand = np.random.RandomState(0)
    learn_rate_0 = learn_rate
    radius_0 = radius_sq
    for epoch in np.arange(0, epochs):
        rand.shuffle(self.data)
        for sample in self.data:
            x, y = self.find_BMU(sample)
            self.SOM = self.update_weights(sample, learn_rate, radius_sq, (x, y))
        self.plot("curr iter: " + str(epoch) + " , learning rate: " + str(round(learn_rate, 3)))
        # Update learning rate and radius
        learn_rate = learn_rate_0 * np.exp(-epoch * lr_decay)
        radius_sq = radius_0 * np.exp(-epoch * radius_decay)
    return self.SOM
```

find BMU - Best Matching Unit

The BMU is the cell of the SOM grid that is closest to the sample.
The BMU is found by calculating the **Euclidean distance** from the weight of each cell (neuron) of the grid to this sample.
The neuron that minimizes the Euclidean distance is the chosen one**.**

-same process for both 1D & 2D-

```
def find_BMU(self, sample):
    """
    find the most close neuron for this sample
    clac the oclid distance from this sample to all neurons,
    pick the neuron that minimize the dist
    :param sample: single training example
    :return:
    """
    distSq = (np.square(self.SOM - sample)).sum(axis=2)
    return np.unravel_index(np.argmin(distSq, axis=None), distSq.shape)
```

update weights:

update BMU and its close neighbors weights.

The size of the neighborhood is determined by the current radius:

- 1D neighborhood.
- 2D neighborhood.

Calculating the new weight:

$$w_{ij}^{(t+1)} = w_{ij}^{t} + learning\ rate^{t} \cdot f_{ij}(x, y, radius^{t}) \cdot (sample - w_{ij}^{t})$$

$t := epoch\ number$

$w := neuron\ weight$

$f_{ij}(x, y, radius^{t}) := neighborhood\ distance\ function$

```python
def update_weights(self, sample, learn_rate, radius_sq, bmu_idx, step=3):
    """
    update weight of BMU and its neighboors
    :param sample: single training example
    :param learn_rate:
    :param radius_sq:
    :param bmu_idx: the best neuron
    :param step:
    :return:
    """
    x = bmu_idx[0]
    # if radius is close to zero then only BMU is changed
    if radius_sq < 1e-3:
        self.SOM[x, :] += learn_rate * (sample - self.SOM[x, :])
        return self.SOM
    # Change all cells in a small neighborhood of BMU
    for i in range(max(0, x - step), min(self.SOM.shape[0], x + step)):
        dist_sq = np.square(i - x)
        dist_func = np.exp(-dist_sq / 2 / radius_sq)
        self.SOM[i, :] += learn_rate * dist_func * (sample - self.SOM[i, :])
    return self.SOM
```

```python
def update_weights(self, sample, learn_rate, radius_sq, bmu_idx, step=3):
    x, y = bmu_idx
    # if radius is close to zero then only BMU is changed
    if radius_sq < 1e-3:
        self.SOM[x, y, :] += learn_rate * (sample - self.SOM[x, y, :])
        return self.SOM
    # Change all cells in a small neighborhood of BMU
    for i in range(max(0, x - step), min(self.SOM.shape[0], x + step)):
        for j in range(max(0, y - step), min(self.SOM.shape[1], y + step)):
            dist_sq = np.square(i - x) + np.square(j - y)
            dist_func = np.exp(-dist_sq / 2 / radius_sq)
            self.SOM[i, j, :] += learn_rate * dist_func * (sample - self.SOM[i, j, :])
    return self.SOM
```

plot

plot the Data

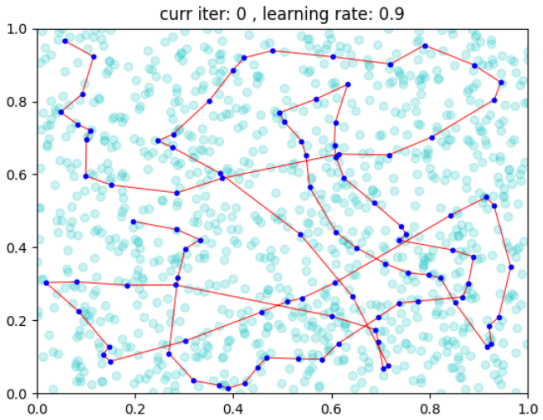
- the Neurons and the **line** in topological order.
- the Neurons and the **grid** in topological order.

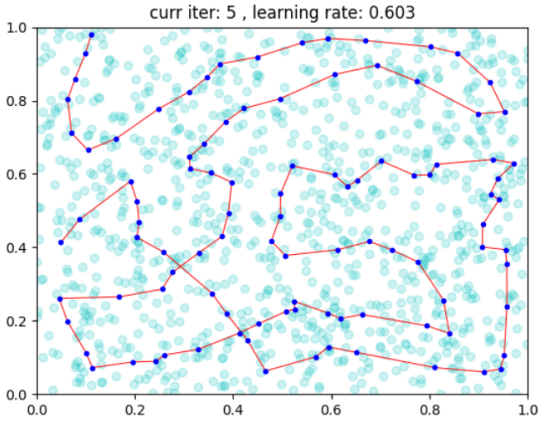
```python
def plot(self, title):
    X = self.SOM[:, 0]  # The X of each point
    Y = self.SOM[:, 1]  # The Y of each point

    fig, ax = plt.subplots()
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    xs = []  # x of each point in axis 1 (cols)
    ys = []  # y of each point in axis 1 (rows)
    for i in range(self.SOM.shape[0]):
        xs.append(self.SOM[i, 0])
        ys.append(self.SOM[i, 1])

    ax.plot(xs, ys, 'r-', markersize=0, linewidth=0.7)
    ax.plot(X, Y, color='b', marker='o', linewidth=0, markersize=3)
    ax.scatter(self.data[:, 0], self.data[:, 1], c="c", alpha=0.2)
    plt.title(title)
    plt.show()
```

```python
def plot(self, title):
    X = self.SOM[:, :, 0]  # The X of each point
    Y = self.SOM[:, :, 1]  # The Y of each point

    fig, ax = plt.subplots()
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    for i in range(self.SOM.shape[0]):
        xs = []  # x of each point in axis 1 (cols)
        ys = []  # y of each point in axis 1 (rows)
        xh = []  # x of each point in axis 0 (cols)
        yh = []  # x of each point in axis 1 (rows)
        for j in range(self.SOM.shape[1]):
            xs.append(self.SOM[i, j, 0])
            ys.append(self.SOM[i, j, 1])
            xh.append(self.SOM[j, i, 0])
            yh.append(self.SOM[j, i, 1])

        ax.plot(xs, ys, 'r-', markersize=0, linewidth=0.7)
        ax.plot(xh, yh, 'r-', markersize=0, linewidth=0.7)

    ax.plot(X, Y, color='b', marker='o', linewidth=0, markersize=3)
    ax.scatter(self.data[:, 0], self.data[:, 1], c="c", alpha=0.2)
    plt.title(title)
    plt.show()
```

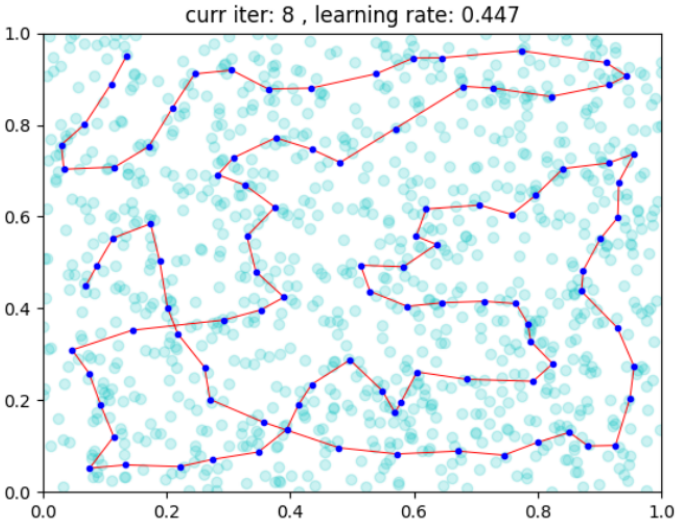
PART A:

1. In this section we required to the create a data set of: {(x,y) | 0 <= x <= 1, 0<=y<=1}- this data is distributed uniformly and uses 100 neurons as a line (1D):

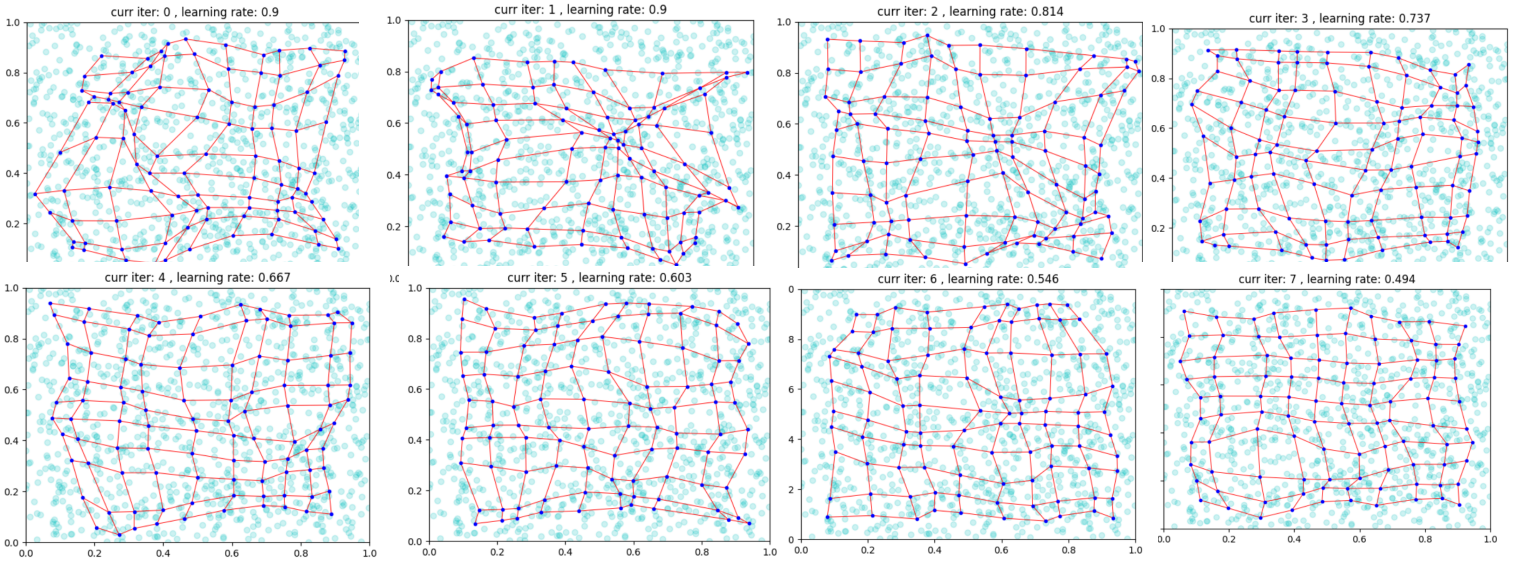




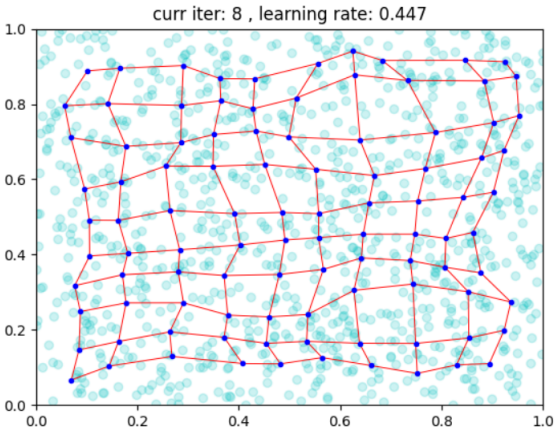
Result :



And then we did the same when the topology of the 100 neurons is arranged in a **two dimensional array** of 10 x 10.

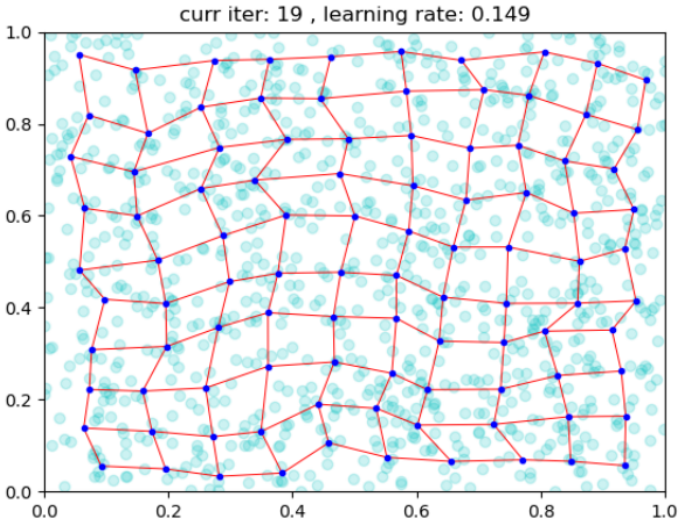


Last iteration:



We can see between each iteration how the algorithm changes the Learning rate and the weights of the SOM which affect the position of the neurons.
Because of the uniform Data - we can see how the neurons strive to spread out uniformly on the space.

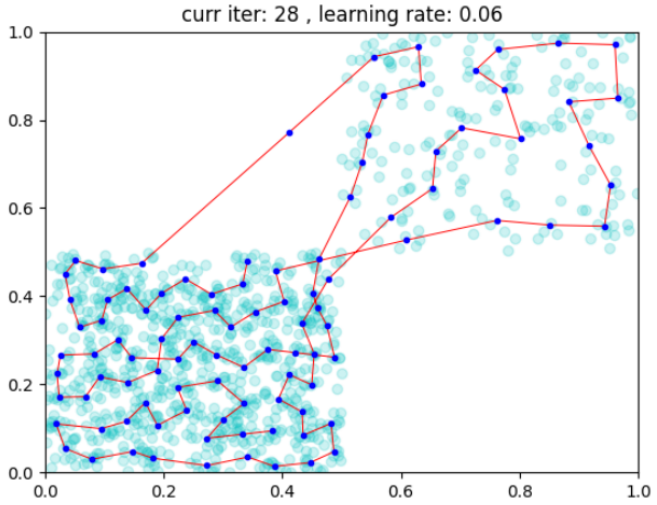
After 20 iterations, the network is more uniformly distributed.



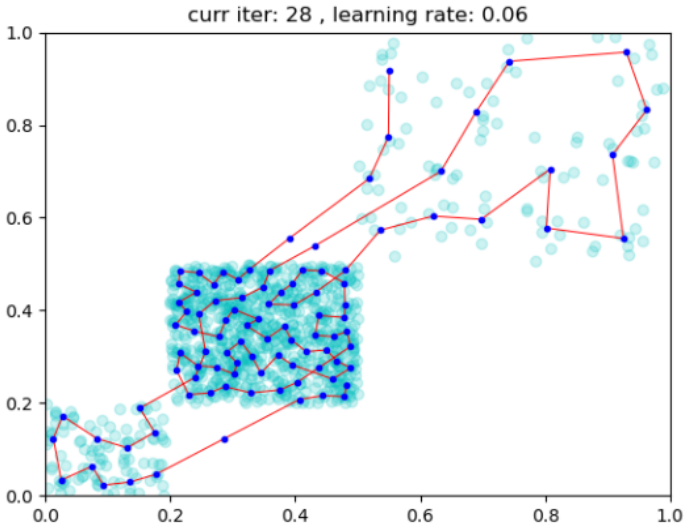
2. Non-uniformly data set:

For this section we required to create two data sets of Non -uniform (x,y) such that:

a. 80% of the data : {(x,y) |0=< x,y<=0.5} and 20% of the data: {(x,y) | 0.5=< x,y<=1}:



b. 80% of the data : {(x,y) |0.2=< x,y<=0.5} , 10% of the data: {(x,y) | 0.5=< x,y<=1}
10% of the data: {(x,y) | 0=< x,y<=0.2}:

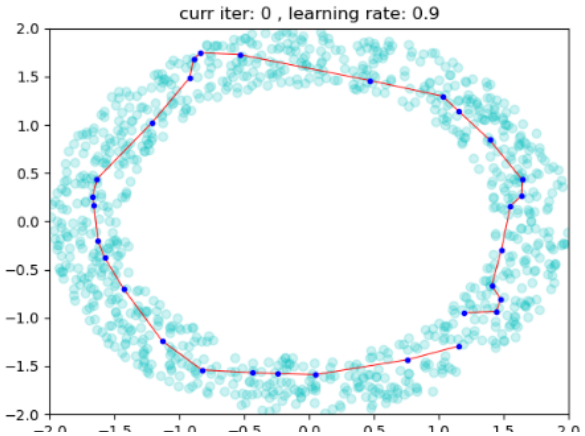


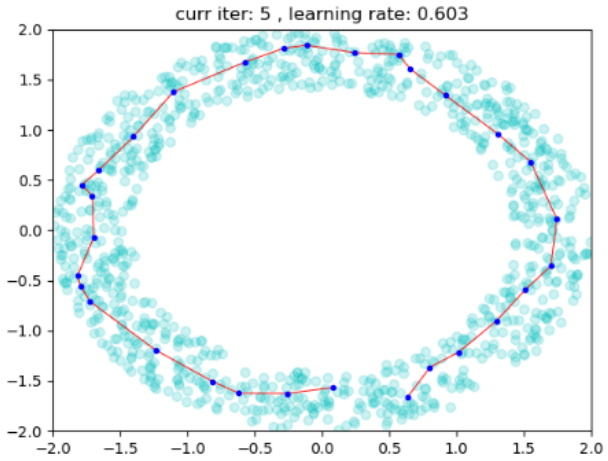
3. circle data set

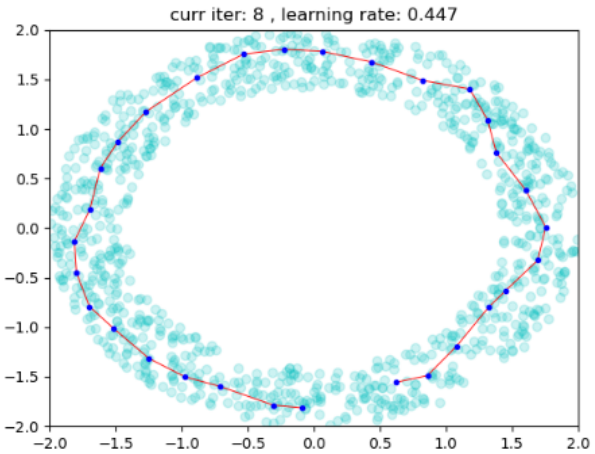
same experiments as above for fitting a circle of neurons on a "donut" shape:

$\{< x, y > \mid 2 \leq x^2 + y^2 \leq 4\}$

The line of neurons has 30 neurons organized as a circle topology.







PART B:

In this part we were required to fit our SOM model with a data set of "Monkey Hand" and 225 neurons arranged in a 15 x 15 mesh.

We created a mask by binary-matrix (put 1 in places "belong" to the hand)

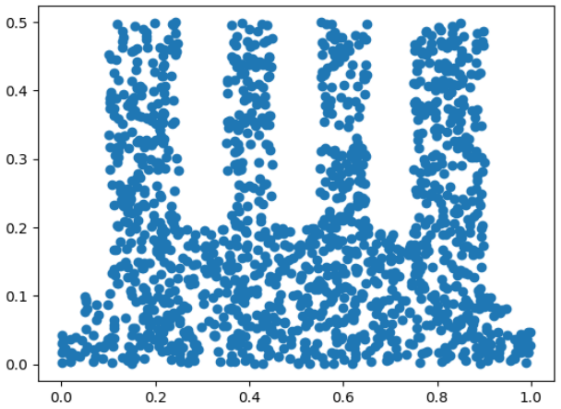
1 → point in this index coordinate {(x,y) | 0 <= x <= 1, 0<=y<=1 }

0 → nothing

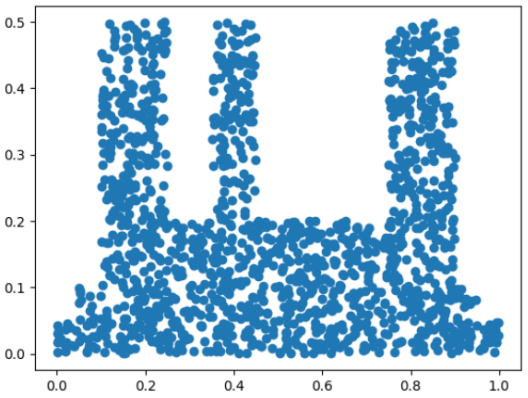
We chose to use our SOM implantation.

The data-set look like:

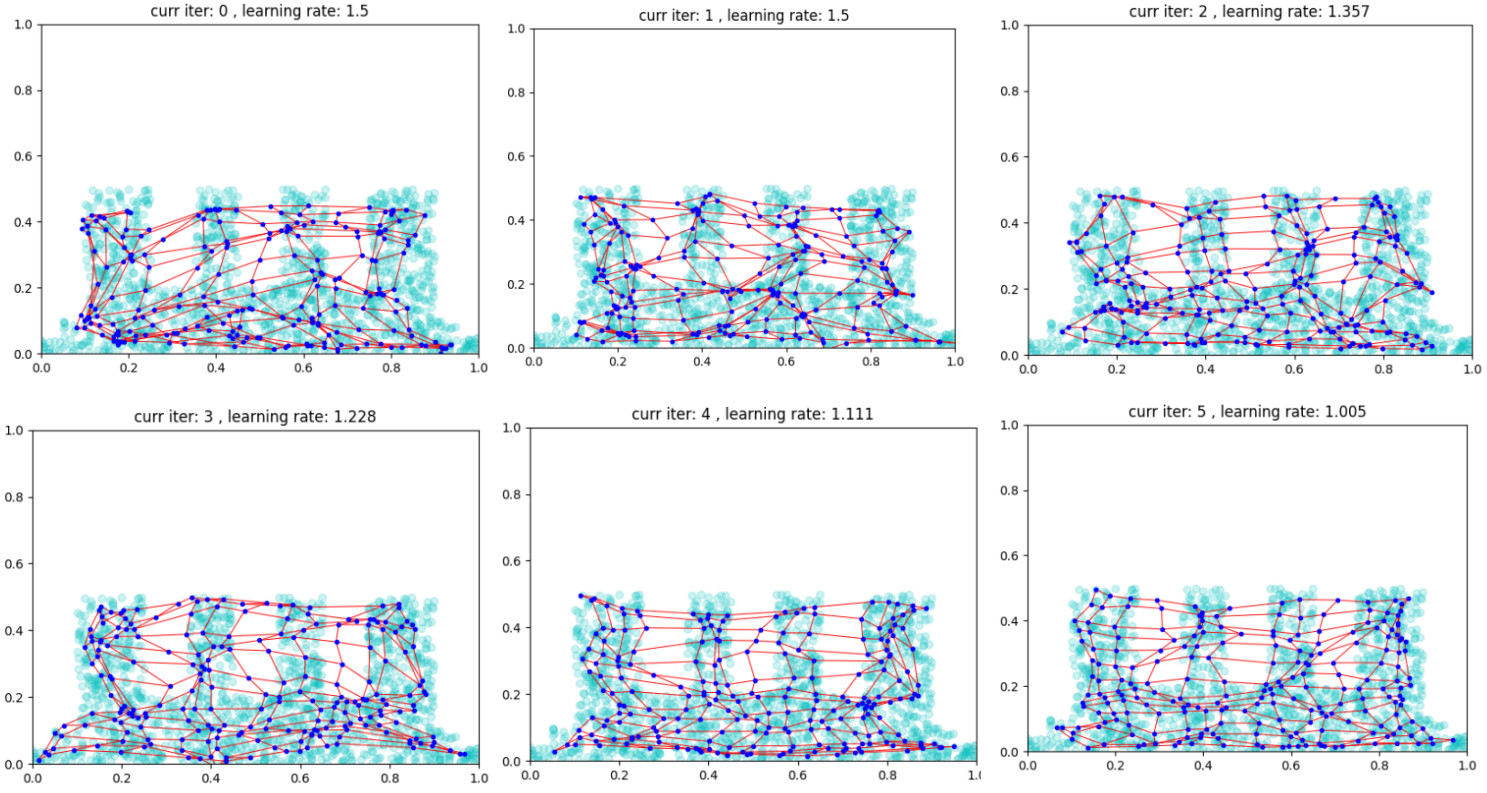
Monkey Hand 4 fingers

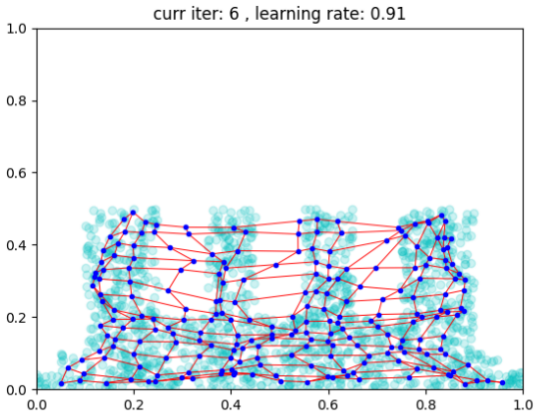


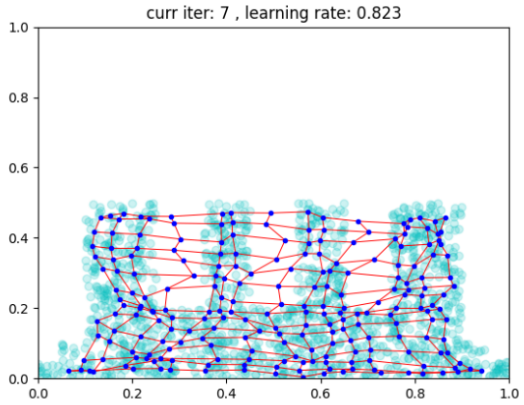
Monkey Hand 3 fingers



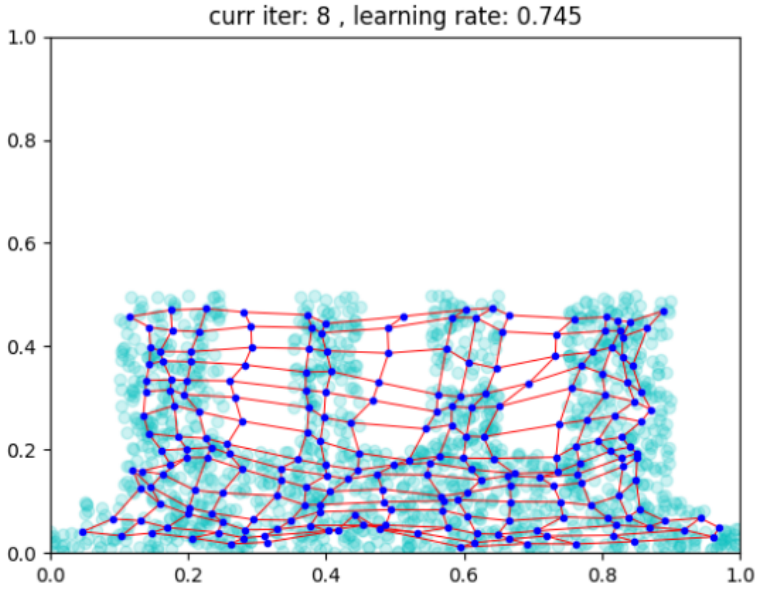
Lets see how the neurons spread over the Monkey Hand in each iteration :



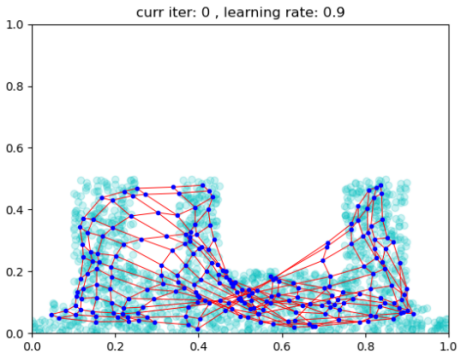


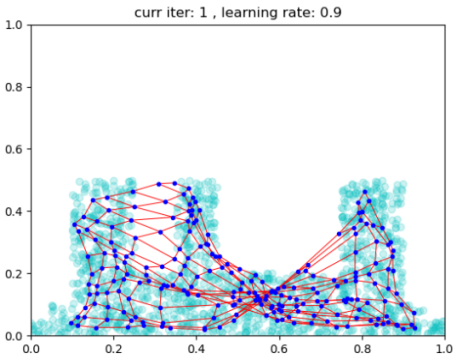


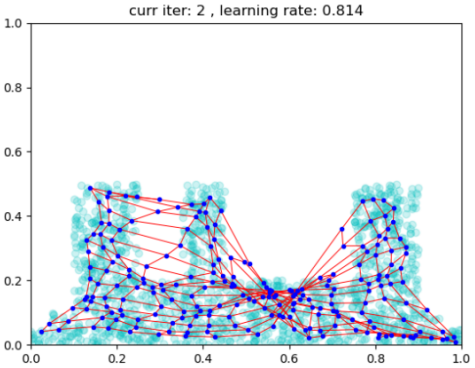
The result :

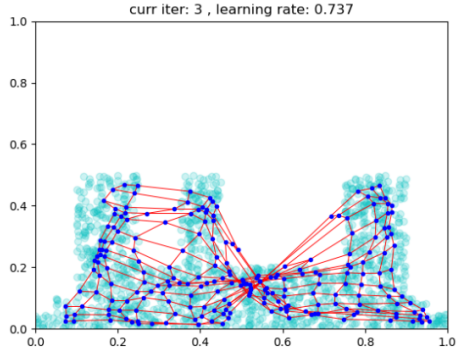


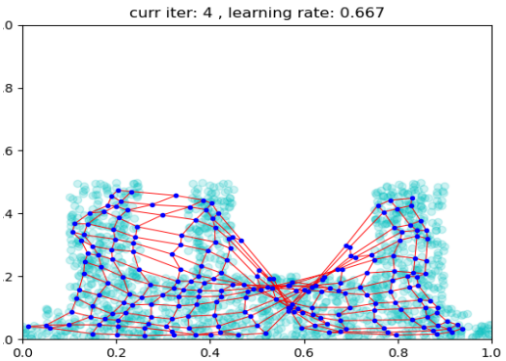
At the first iteration we can see that the neurons spread randomly (according to the algorithm) , and after each iteration the neurons spread more and more according to the shape of the monkey hand.

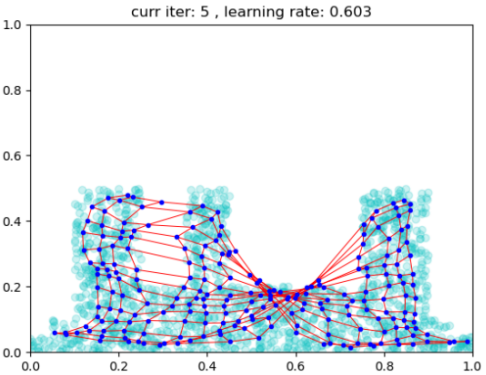

curr iter: 0 , learning rate: 0.9

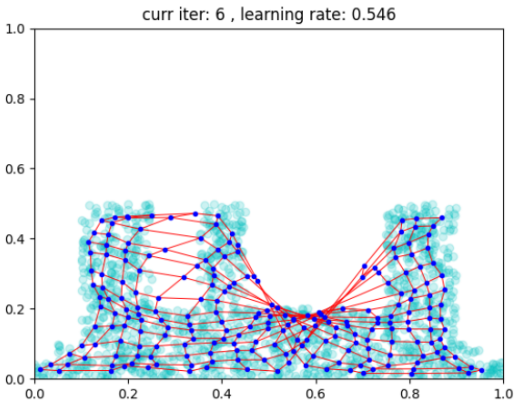

curr iter: 1 , learning rate: 0.9

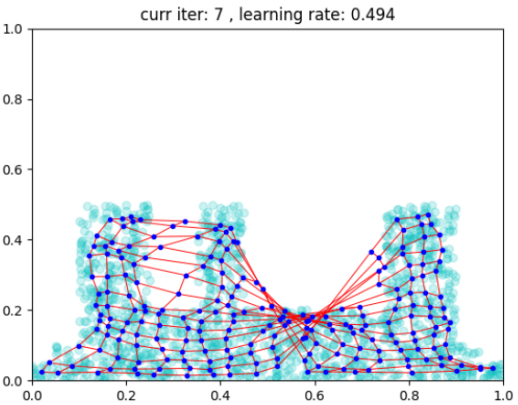

curr iter: 2 , learning rate: 0.814

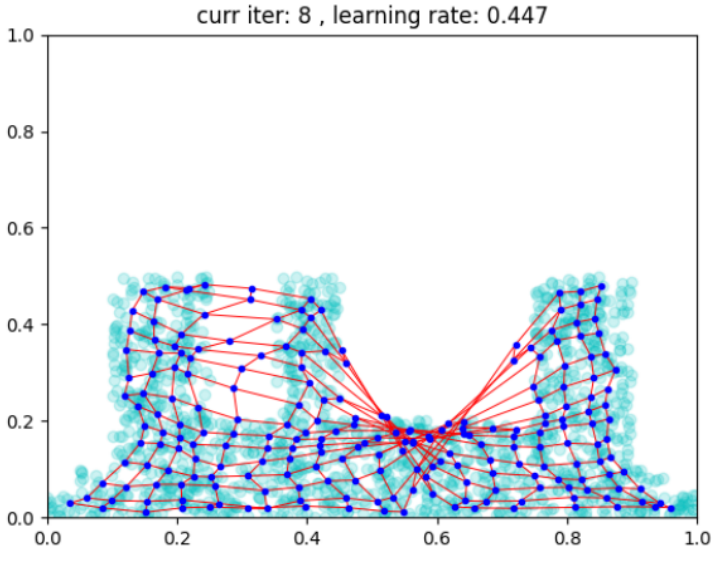

curr iter: 3 , learning rate: 0.737


curr iter: 4 , learning rate: 0.667


curr iter: 5 , learning rate: 0.603


curr iter: 6 , learning rate: 0.546


curr iter: 7 , learning rate: 0.494


curr iter: 8 , learning rate: 0.447