Introduction to Neural Computation

Kohonen (SOM) algorithm

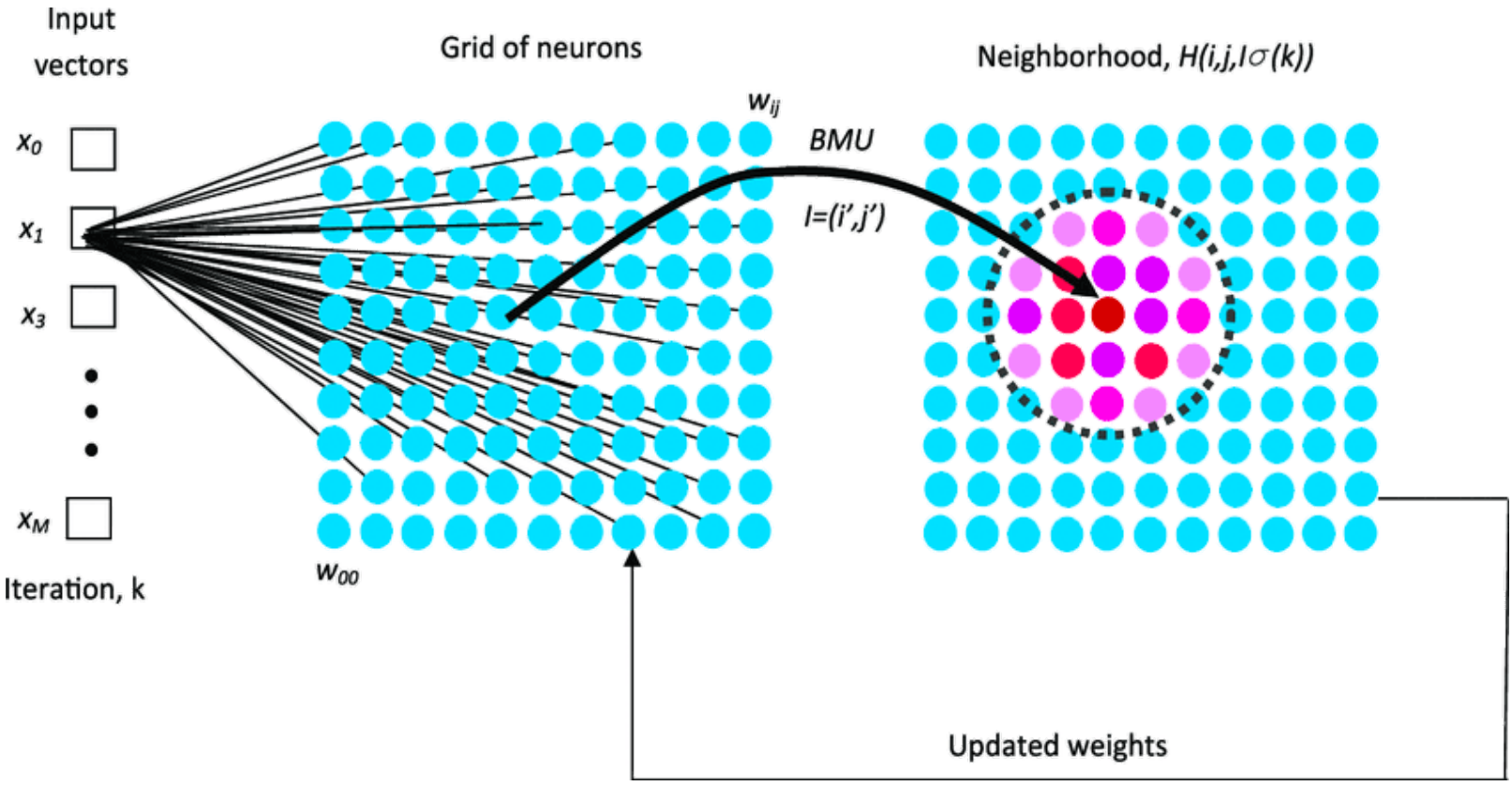
Ex2 - Part A, B

Noamya Shani – 316503986(evening)

Assaf Yekutiel – 206316895 (morning)

Avidan Abitbol– 302298963(evening)

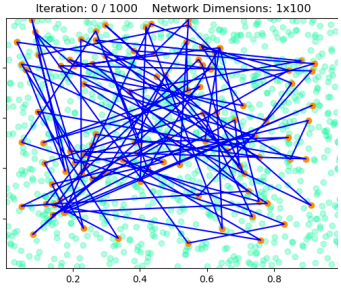
Itamar Kraitman– 208925578(evening)

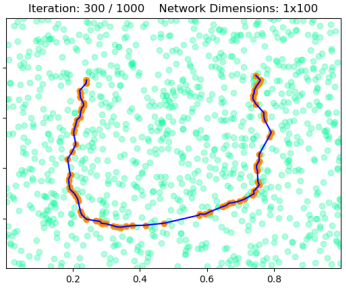


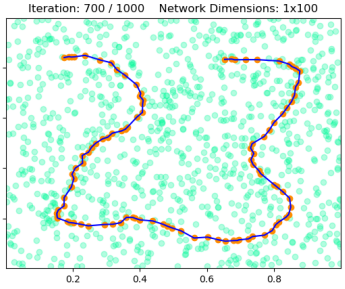
Part A1

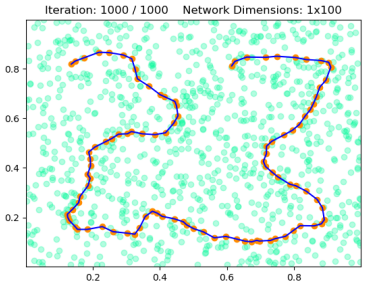
In this part, we applied the Kohonen (SOM) algorithm. part A, class 1 includes all the data points which satisfied the condition- {(x,y) | 0 <= x <= 1, 0<=y<=1} with uniform distribution. We used our Implementation of the algorithm “Kohonen (SOM)” in this part. We ran the algorithm twice. Once with a network of neurons in a 1X100 structure And a second time with a network of neurons in a 10X10 structure.

1X100 neurons

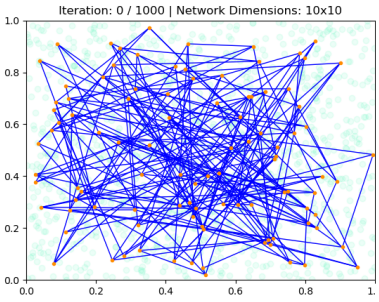


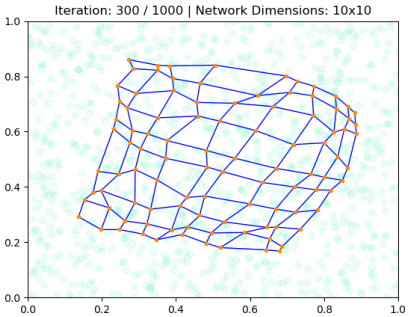


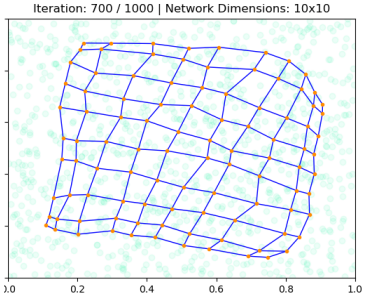


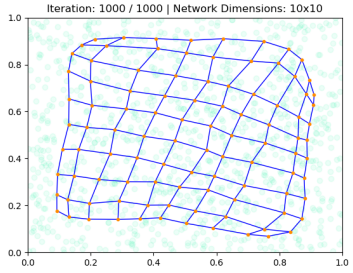


10X10 neurons







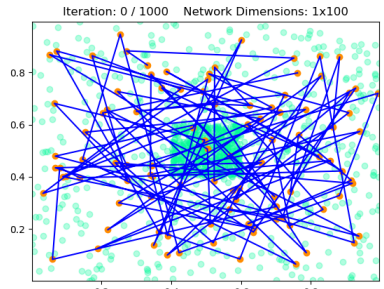


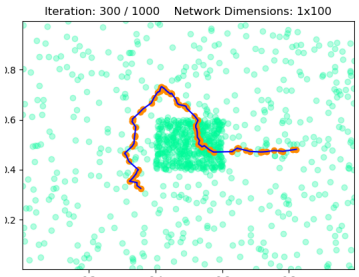
It is noticeable how the neurons spread relatively equally over the data, this is what we expect to see from the fact that the data is evenly distributed in space so we do not have large clusters of data.

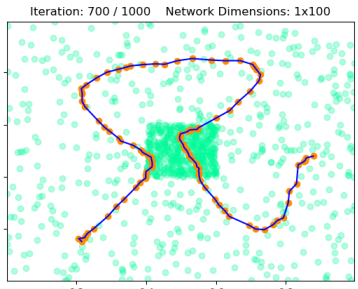
Part A2

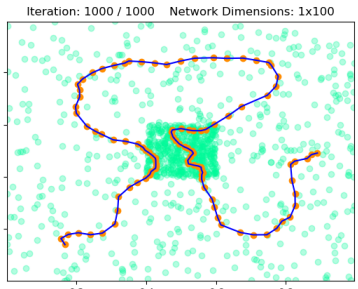
In class 2 we decide to organize the data distribution by making a window on the right upper side and in the center from 50% of the data and the rest uniformly. We ran the algorithms twice. Once with a network of neurons in a 1X100 structure And a second time with a network of neurons in a 10X10 structure.

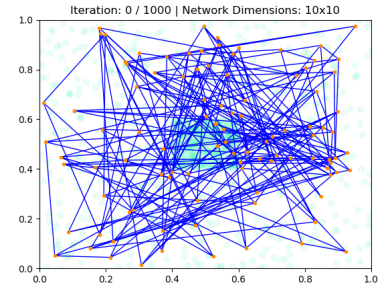
Most of the data is in the middle

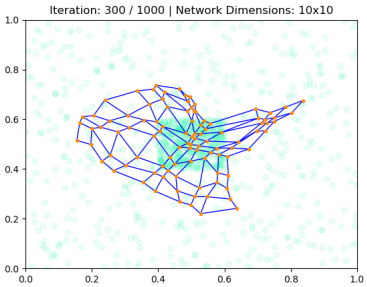


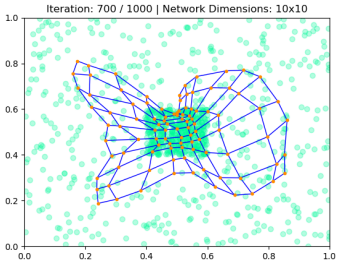


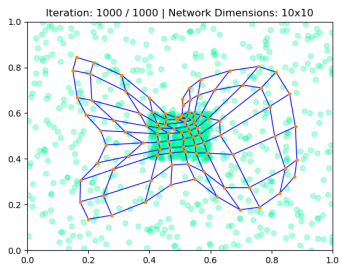




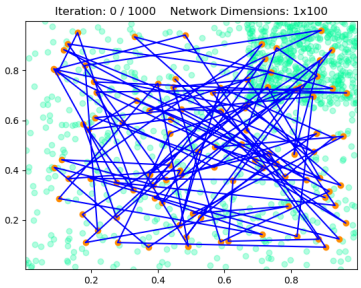


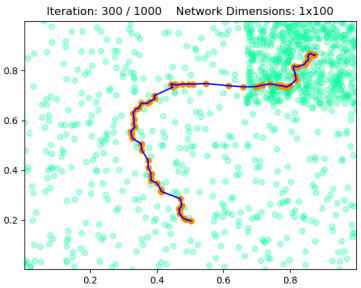


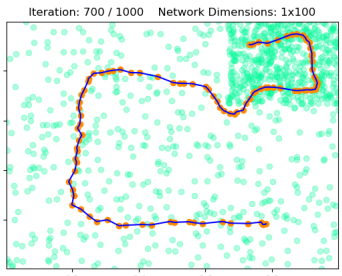


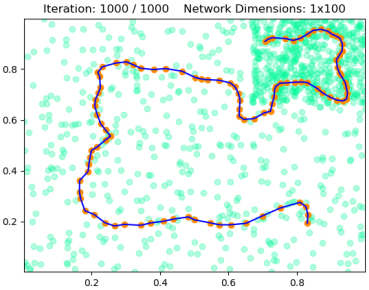


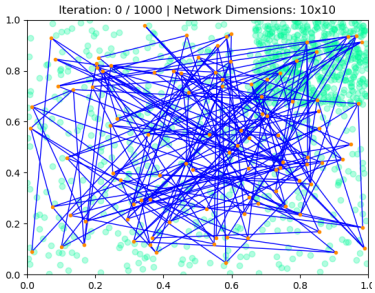
Most of the data is in the top right corner

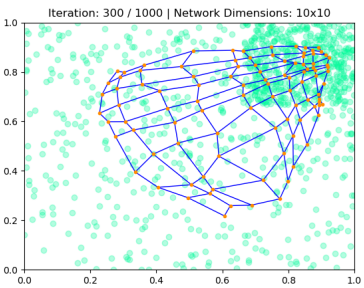


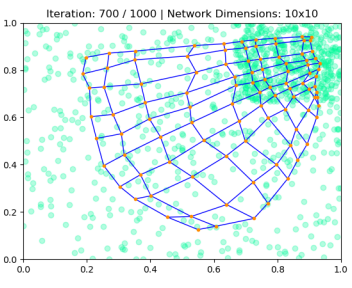


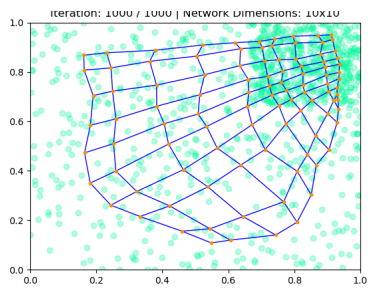












In this section, we can see how a significant portion of the neurons are heavily concentrated where there is a large cluster of data, and in addition, they are very close to each other in those areas.

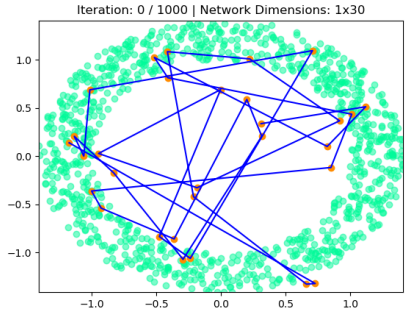
In addition, we can see how the rest of the neurons map out the rest of the data which is what we expect to see in this type of data.

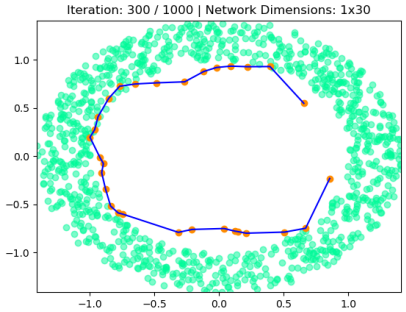
Another thing that is important to note is the size of the change as we progress in iterations the size of the change decreases and is also more local.

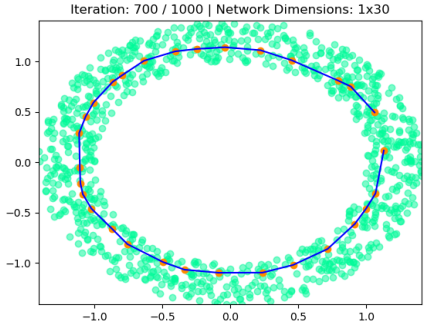
This is due to the fact that we reduce the radius and the learning rate as we progress in order not to harm what we have already mapped successfully and only to refine the mapping we have already managed to do.(this is relevant to all the sections in this project)

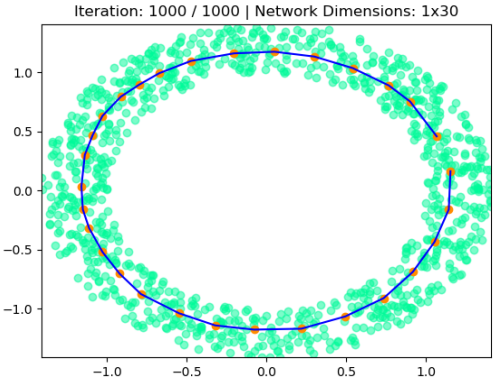
Part A3

part A, class 3 includes all the data points which satisfied the condition-{<x.y> | 2<= x^2 +y^2 <= 4}. We used our implementation of the algorithm “Kohonen (SOM)”. We ran the algorithms with a network of neurons in a 1X30 structure too.

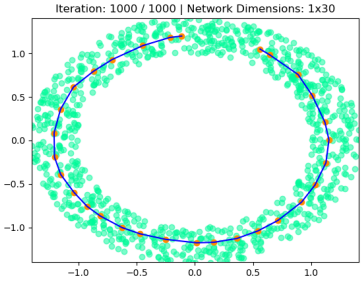








To create the effect of a circle we set that the distance between the last and first neuron is equal to 1 so that they affect each other like two neighboring neurons.
To compare we can see here what happens if we don't use this condition the neurons are far apart:



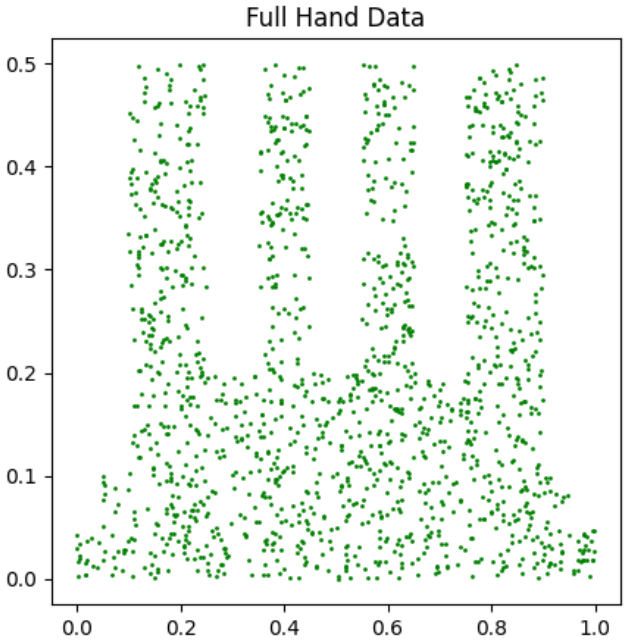
We can see over the iterations how the neurons are getting closer and closer to mapping the shape of the donut until they reach a perfect form.

Part B

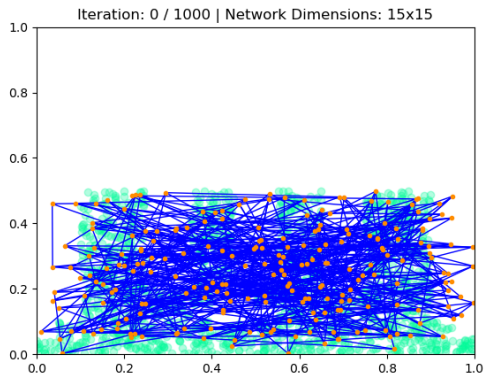
In this part, we reproduced the experiment of the “monkey hand” we discussed in class. For this part, we used our implementation of the Kohonen algorithm from part A.

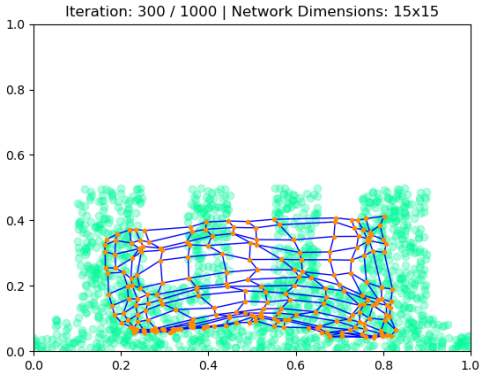
Full Hand:

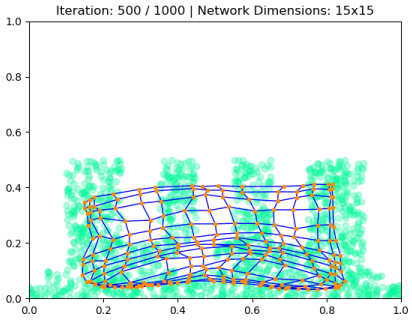
In this section, we built a Kohonen space with 225 neurons arranged in a 15X15 mesh with a data set where the data is <x,y> such that 0 <= x <= 1 and 0 <= y <= 1 where the points are inside the “hand”.

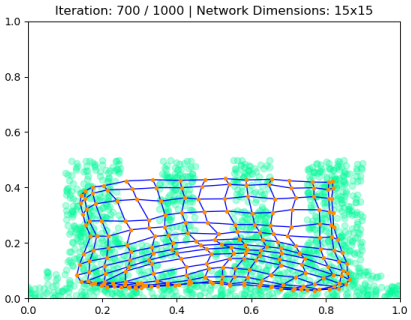


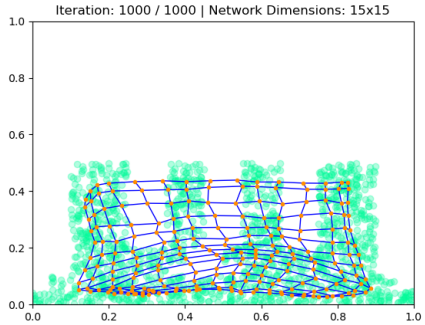
Below we present the fitting process- how the mesh is superimposed on the plane that contains the hand and how it changes over iterations.







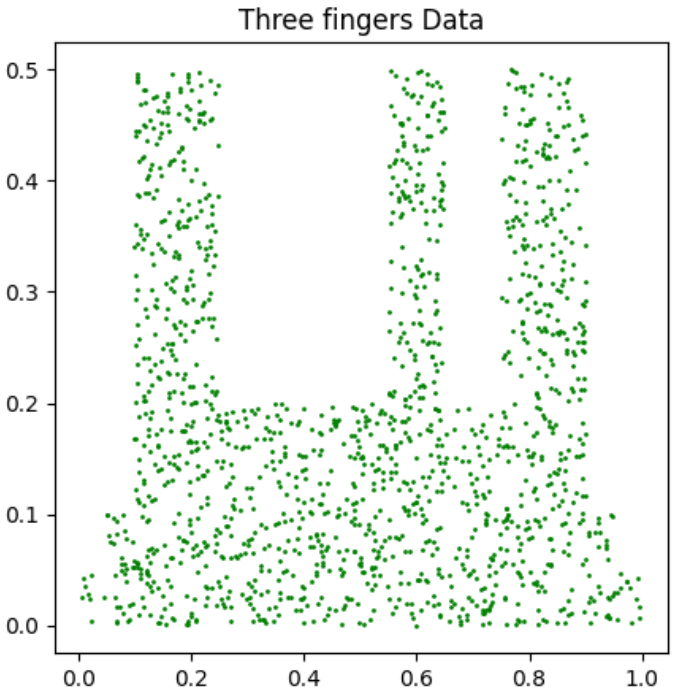




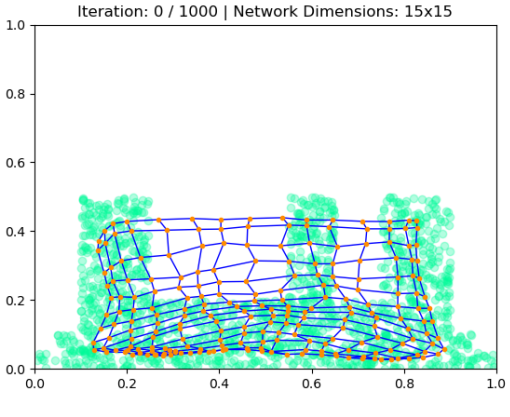
As can be seen, in the beginning, the neurons were spread randomly, and over the iterations, they converge into the shape of the hand.

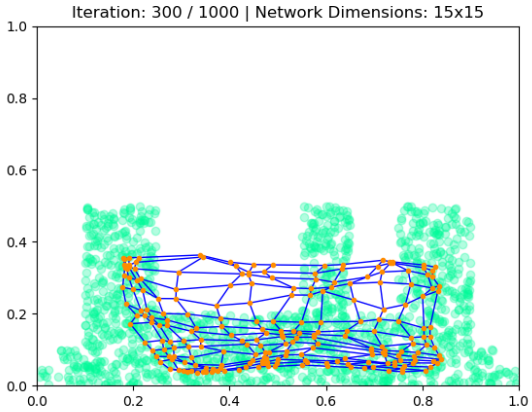
Three Fingers:

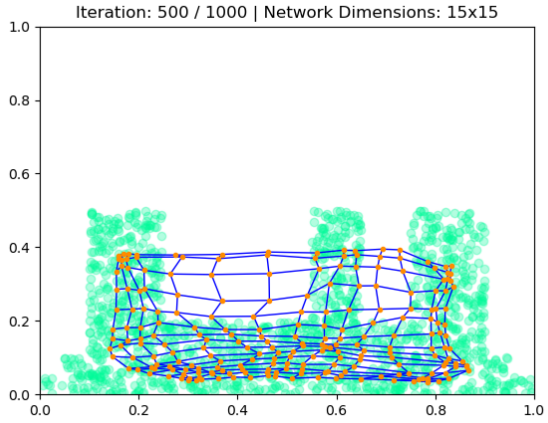
In this section. We took the SOM classifier we trained in the previous section and “cut off a finger” so the data points come from only three fingers, and then continued the training from where we stopped.

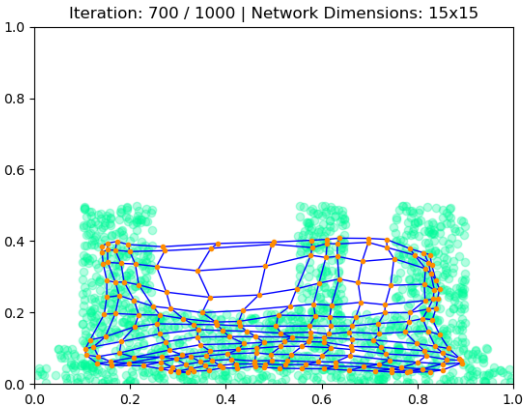


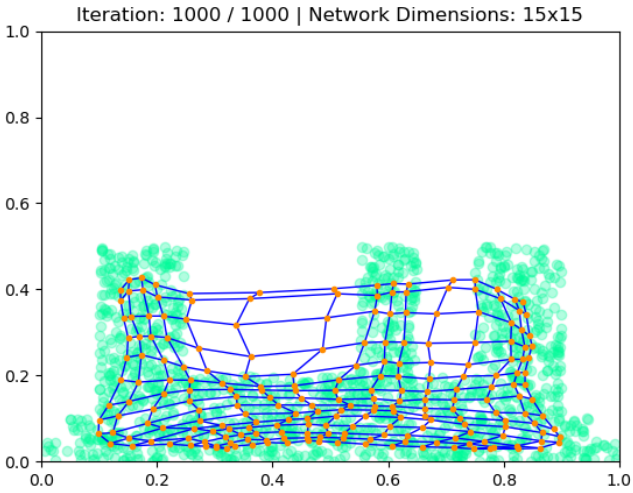
Below we present the fitting process- how the mesh is superimposed on the plane that contains the hand and how it changes over iterations.











As can be seen, in the beginning, the neurons were spread in the shape of a full hand, over the iterations they converged into the shape of three fingers.

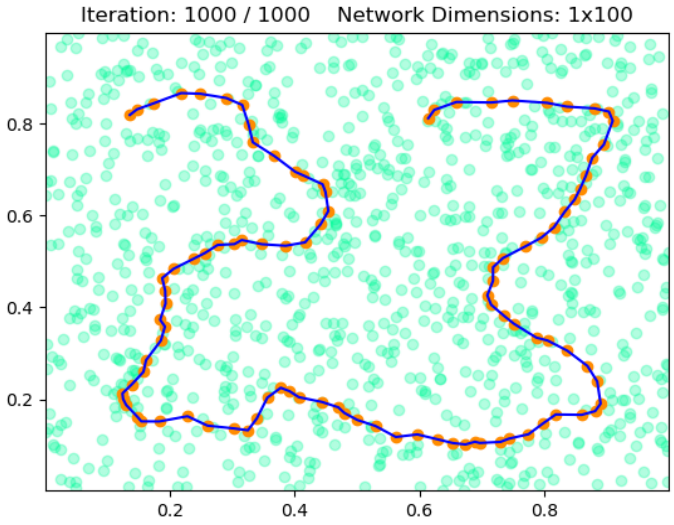
Conclusion

In this project, we implemented the SOM algorithm and examined its behavior on different types of data.

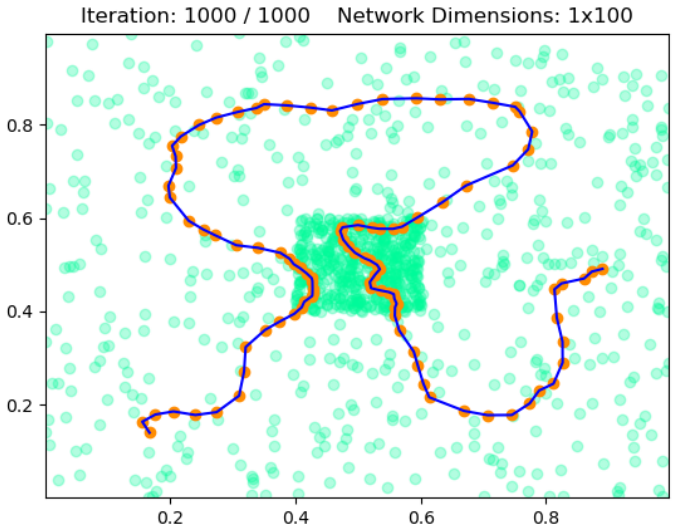
The algorithm is working in the following way:

1. Randomly select a weight vector for each neuron that represents a point in the data space.
2. Select a random data point from data and find which neuron weight vector is the closest neuron this neuron is called BMU(best matching unit).
3. Update the vector weights to be closer to the data point value.
4. Update the weights of all other neurons within the selected radius according to their distance from BMU (The closer they are the greater the change).
5. Repeat this process as the number of iterations we have defined.
6. Return the net with the new weight vectors for each neuron.

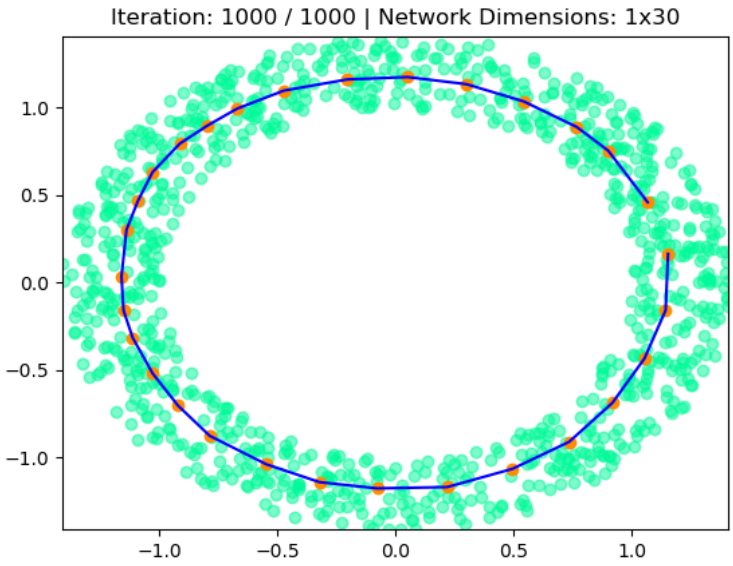
In part A we tested the algorithm in different scenarios.
We started with a uniform distribution of the data and saw how the neurons are distributed relatively evenly in space:



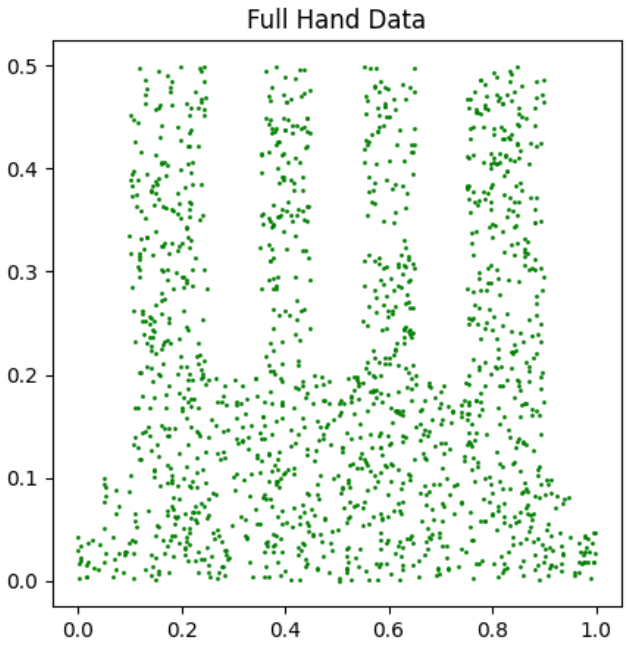
We then moved on to two examples of uneven distribution of data in space. We saw how a significant proportion of neurons are attracted there and their density in these areas:

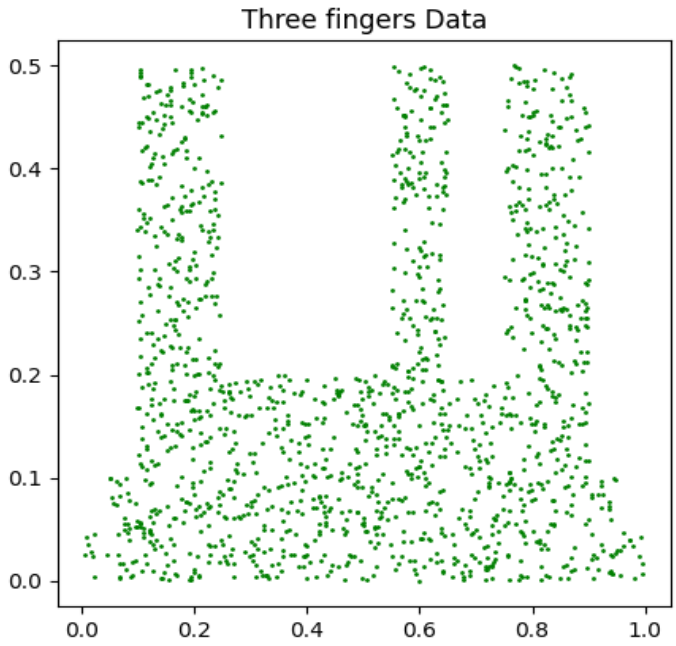


In the final scene, we mapped with 30 neurons in a form of a circle the donut shape
The algorithm was able to excellently map the shape of the donut as you can see here:

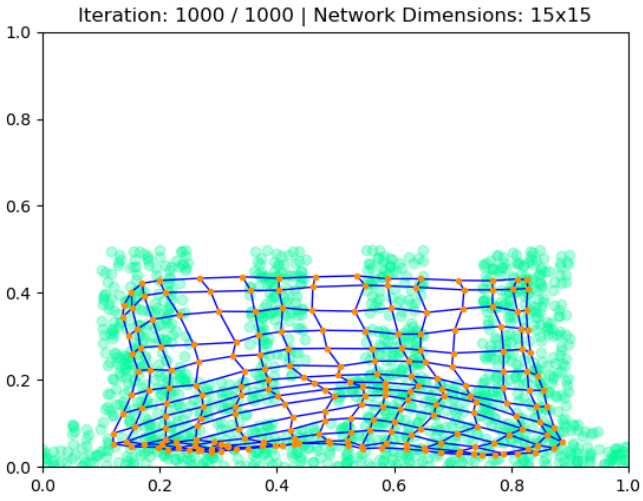


We reproduced the experiment on the “monkey finger” we talked about in class in part B. In this experiment, we built a SOM with 15X15 neurons (225) with the algorithm we produced by ourselves (we did not use a built-in or existing implementation from some library!), where the data is organized as a hand or a cut of hand with three fingers.

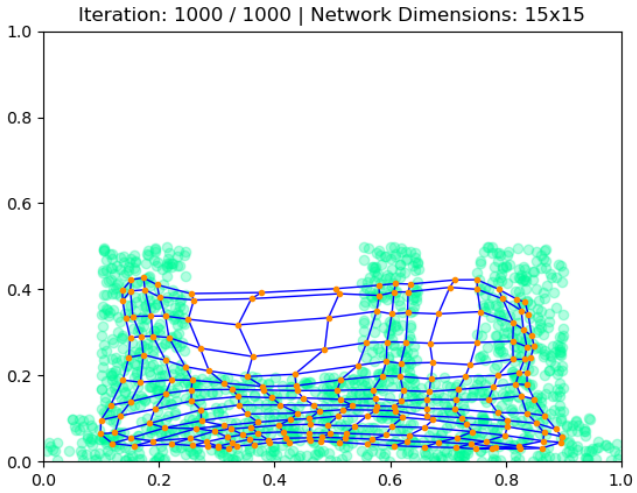




In section 1 we trained the SOM with the data of the “full” hand and as can be seen, we successfully received the desirable outcome:



In section 2, we trained the SOM with the data of the cut of hand, where the starting point was the SOM which we trained on the full hand data (section 1) to get the shape of the cut-off hand. To make things clear, the starting point was the full hand data after 1000 iterations (the plot above) as can be seen below, we successfully received the desirable outcome:



In conclusion, we can say confidently that the SOM algorithm we produced can map the neurons to their accurate position successfully in part A altogether with part B.

Code

Som.py

```
class Som:
    def __init__(self, data, n_iterations=1000, learning_rate=0.1, dima=15, dimb=15):
        self.n_iterations = n_iterations
        self.data = None
        self.learning_rate = learning_rate
        self.network_dimensions = np.array([dima, dimb])
        self.net, self.time_constant, self.init_radius, self.x_min, self.x_max, self.y_min, self.y_max = \
            self.initialization(data)

    def initialization(self, data):
        init_radius = max(self.network_dimensions[0], self.network_dimensions[1]) / 2  # radius
        self.data = data
        # radius decay parameter
        time_constant = self.n_iterations / np.log(init_radius)
        x_min = np.min(data[:, 0])
        y_min = np.min(data[:, 1])
        x_max = np.max(data[:, 0])
        y_max = np.max(data[:, 1])
        net = np.array([[(random.uniform(x_min + 0.001, x_max),
                          random.uniform(y_min + 0.001, y_max)) for _ in range(self.network_dimensions[0])] for _ in
                        range(self.network_dimensions[1])])  # initialize the network
        return net, time_constant, init_radius, x_min, x_max, y_min, y_max
```

```
def fit(self, data):
    for i in range(self.n_iterations + 1):
        # select a training example at random
        t = random.randint(0, len(data) - 1)  # random index
        # find its Best Matching Unit
        bmu_idx = np.array([0, 0])  # index of the best matching unit
        min_dist = np.inf  # initialize the minimum distance with infinity
        # calculate the distance between each neuron and the input
        for x in range(self.net.shape[0]):
            for y in range(self.net.shape[1]):
                w = np.linalg.norm(data[t] - self.net[x, y])  # euclidean distance
                if w < min_dist:
                    min_dist = w  # dist
                    bmu_idx = np.array([x, y])  # id
        # decay the SOM parameters
        r = self.init_radius * np.exp(-i / self.time_constant)  # radius
        l_rate = self.learning_rate * np.exp(-i / self.n_iterations)  # learning rate
        # update weight vector to move closer to input
        # and move its neighbours in 2-D vector space closer
        for x in range(self.net.shape[0]):
            for y in range(self.net.shape[1]):
                w = self.net[x, y]
                w_dist = np.sum((np.array([x, y]) - bmu_idx) ** 2)  # dist
                w_dist = np.sqrt(w_dist)  # dist
                if w_dist <= r:
                    # calculate the degree of influence (based on the 2-D distance)
                    influence = np.exp(-w_dist / (2 * (r ** 2)))
                    # new w = old w + (learning rate * influence * delta)
                    # where delta = input vector (t) - old w
                    new_w = w + (l_rate * influence * (data[t] - w))
                    self.net[x, y] = new_w  # update the weight vector
        if i % 100 == 0:  # plot the map every 100 iterations
            if self.network_dimensions[0] == 1:
                self.draw_map(self.x_max, self.x_min, i)
            else:
                self.plot_net(i)

    return self.net
```

Plot the results:

```
def draw_map(self, max, min, t):
    x_n = []  # x-coordinates of the neurons
    y_n = []  # y-coordinates of the neurons
    for i in range(self.net.shape[0]):
        for j in range(self.net.shape[1]):
            x_n.append(self.net[i, j, 0])
            y_n.append(self.net[i, j, 1])
    fig, ax = plt.subplots()
    ax.set_xlim(min, max)
    ax.set_ylim(min, max)
    ax.plot(x_n, y_n, 'b-')
    ax.scatter(self.data[:, 0], self.data[:, 1], alpha=0.3, c='mediumspringgreen')  # plot the data
    ax.scatter([x_n], [y_n], c='darkorange')  # plot the neurons
    ax.set_title(
        f'Iteration: {t} / {self.n_iterations} | Network Dimensions: {self.network_dimensions[0]}x{self.network_dimensions[1]}')
    plt.show()

def plot_net(self, t):
    neurons_x = self.net[:, :, 0]  # x-coordinates of the neurons
    neurons_y = self.net[:, :, 1]  # y-coordinates of the neurons
    fig, ax = plt.subplots()
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    for i in range(self.network_dimensions[0]):
        xh = []
        yh = []
        xs = []
        ys = []
        for j in range(self.network_dimensions[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, 'b-', markersize=0, linewidth=1)
        ax.plot(xh, yh, 'b-', markersize=0, linewidth=1)
    ax.plot(neurons_x, neurons_y, color='darkorange', marker='o', linewidth=0, markersize=3)
    ax.scatter(self.data[:, 0], self.data[:, 1], c='mediumspringgreen', alpha=0.3)
    ax.set_title(
        f'Iteration: {t} / {self.n_iterations} | Network Dimensions: {self.network_dimensions[0]}x{self.network_dimensions[1]}')
    plt.show()
```

Monkey hand:

```
    def fit_and_show_changes(self, data, data_type):
        net, time_constant, init_radius, x_min, x_max, y_min, y_max = self.som.initialization(data)
        # print(net.shape)
        for i in range(self.som.n_iterations + 1):
            # select a training example at random
            t = random.randint(0, len(data) - 1)
            # find its Best Matching Unit
            bmu_idx = np.array([0, 0])
            min_dist = np.inf
            # calculate the distance between each neuron and the input
            for x in range(net.shape[0]):
                for y in range(net.shape[1]):
                    w = np.linalg.norm(data[t] - self.som.net[x, y])
                    if w < min_dist:
                        min_dist = w  # dist
                        bmu_idx = np.array([x, y])  # id
            # decay the SOM parameters
            r = init_radius * np.exp(-i / time_constant)
            l = self.som.learning_rate * np.exp(-i / self.som.n_iterations)
            # update weight vector to move closer to input
            # and move its neighbours in 2-D vector space closer
            for x in range(net.shape[0]):
                for y in range(net.shape[1]):
                    w = self.som.net[x, y]
                    w_dist = np.sum((np.array([x, y]) - bmu_idx) ** 2)  # dist
                    w_dist = np.sqrt(w_dist)  # dist
                    if w_dist <= r:
                        # calculate the degree of influence (based on the 2-D distance)
                        influence = np.exp(-w_dist / (2 * (r ** 2)))
                        # new w = old w + (learning rate * influence * delta)
                        # where delta = input vector (t) - old w
                        new_w = w + (l * influence * (data[t] - w))
                        # print(w.shape)
                        self.som.net[x, y] = new_w
            if (i % 100 == 0):
                self.som.plot_net(i)
        return self.som.net
```

```python
def create_data(data_size, data_kind=1):
    data = np.zeros((data_size, 2))
    if data_kind == 1:
        for i in range(data_size):
            data[i, 0] = random.uniform(0, 1000) / 1000
            data[i, 1] = random.uniform(0, 1000) / 1000
    elif data_kind == 2:
        # create values between 0.4 and 0.6
        for i in range(data_size // 2):
            data[i, 0] = random.uniform(0, 1000) / 1000
            data[i, 1] = random.uniform(0, 1000) / 1000
        for i in range(data_size // 2, data_size):
            data[i, 0] = random.uniform(0.4, 0.6)
            data[i, 1] = random.uniform(0.4, 0.6)
    elif data_kind == 3:
        for i in range(data_size // 2):
            data[i, 0] = random.uniform(0, 1000) / 1000
            data[i, 1] = random.uniform(0, 1000) / 1000
        for i in range(data_size // 2, data_size):
            data[i, 0] = random.uniform(500, 750) / 750
            data[i, 1] = random.uniform(500, 750) / 750
    elif data_kind == 4:  # create donut shape
        n = 0
        while n < data_size:
            x = random.uniform(-2, 2)
            y = random.uniform(-2, 2)
            if 1 <= x ** 2 + y ** 2 <= 2:
                data[n, 0] = x
                data[n, 1] = y
                n += 1
    return data
```

Fit for the donut shape

```
def fit_donut(self, data):
    for i in range(self.n_iterations + 1):
        # select a training example at random
        t = random.randint(0, len(data) - 1)  # random index
        # find its Best Matching Unit
        bmu_idx = np.array([0, 0])  # index of the best matching unit
        min_dist = np.inf  # initialize the minimum distance with infinity
        # calculate the distance between each neuron and the input
        for x in range(self.net.shape[0]):
            for y in range(self.net.shape[1]):
                w = np.linalg.norm(data[t] - self.net[x, y])  # euclidean distance
                if w < min_dist:
                    min_dist = w  # dist
                    bmu_idx = np.array([x, y])  # id
        # decay the SOM parameters
        r = self.init_radius * np.exp(-i / self.time_constant)
        l_rate = self.learning_rate * np.exp(-i / self.n_iterations)  # learning rate
        # update weight vector to move closer to input
        # and move its neighbours in 2-D vector space closer
        if bmu_idx[0] == 0 and bmu_idx[1] == 0:
            for x in range(self.net.shape[0]):
                for y in range(self.net.shape[1]):
                    w = self.net[x, y]
                    if x == self.net.shape[0] - 1:
                        w_dist = 1
                    else:
                        w_dist = np.sum((np.array([x, y]) - bmu_idx) ** 2)  # dist
                        w_dist = np.sqrt(w_dist)  # dist
                    if w_dist <= r:
                        # calculate the degree of influence (based on the 2-D distance)
                        influence = np.exp(-w_dist / (2 * (r ** 2)))
                        # new w = old w + (learning rate * influence * delta)
                        # where delta = input vector (t) - old w
                        new_w = w + (l_rate * influence * (data[t] - w))
                        self.net[x, y] = new_w  # update the weight vector
        elif bmu_idx[0] == self.net.shape[0] - 1:
            for x in range(self.net.shape[0]):
                for y in range(self.net.shape[1]):
                    w = self.net[x, y]
                    if x == 0:
                        w_dist = 1
                    else:
```

```
    elif bmu_idx[0] == self.net.shape[0] - 1:
        for x in range(self.net.shape[0]):
            for y in range(self.net.shape[1]):
                w = self.net[x, y]
                if x == 0:
                    w_dist = 1
                else:
                    w_dist = np.sum((np.array([x, y]) - bmu_idx) ** 2)  # dist
                    w_dist = np.sqrt(w_dist)  # dist
                if w_dist <= r:
                    # calculate the degree of influence (based on the 2-D distance)
                    influence = np.exp(-w_dist / (2 * (r ** 2)))
                    # new w = old w + (learning rate * influence * delta)
                    # where delta = input vector (t) - old w
                    new_w = w + (l_rate * influence * (data[t] - w))
                    self.net[x, y] = new_w  # update the weight vector
    else:
        for x in range(self.net.shape[0]):
            for y in range(self.net.shape[1]):
                w = self.net[x, y]
                w_dist = np.sum((np.array([x, y]) - bmu_idx) ** 2)  # dist
                w_dist = np.sqrt(w_dist)  # dist
                if w_dist <= r:
                    # calculate the degree of influence (based on the 2-D distance)
                    influence = np.exp(-w_dist / (2 * (r ** 2)))
                    # new w = old w + (learning rate * influence * delta)
                    # where delta = input vector (t) - old w
                    new_w = w + (l_rate * influence * (data[t] - w))
                    self.net[x, y] = new_w  # update the weight vector
    if (i % 100 == 0):  # plot the map every 100 iterations
        if self.network_dimensions[0] == 1:
            self.draw_map(self.x_max, self.x_min, i)
        else:
            self.plot_net(i)

return self.net
```