Neuro-Computation Project II Submission

Michael Kyuchka, ID: 315978403
Tal Goldberg, ID: 308361476
Rotem Halbreich, ID: 311549364
Ishay Levy, ID: 318439759

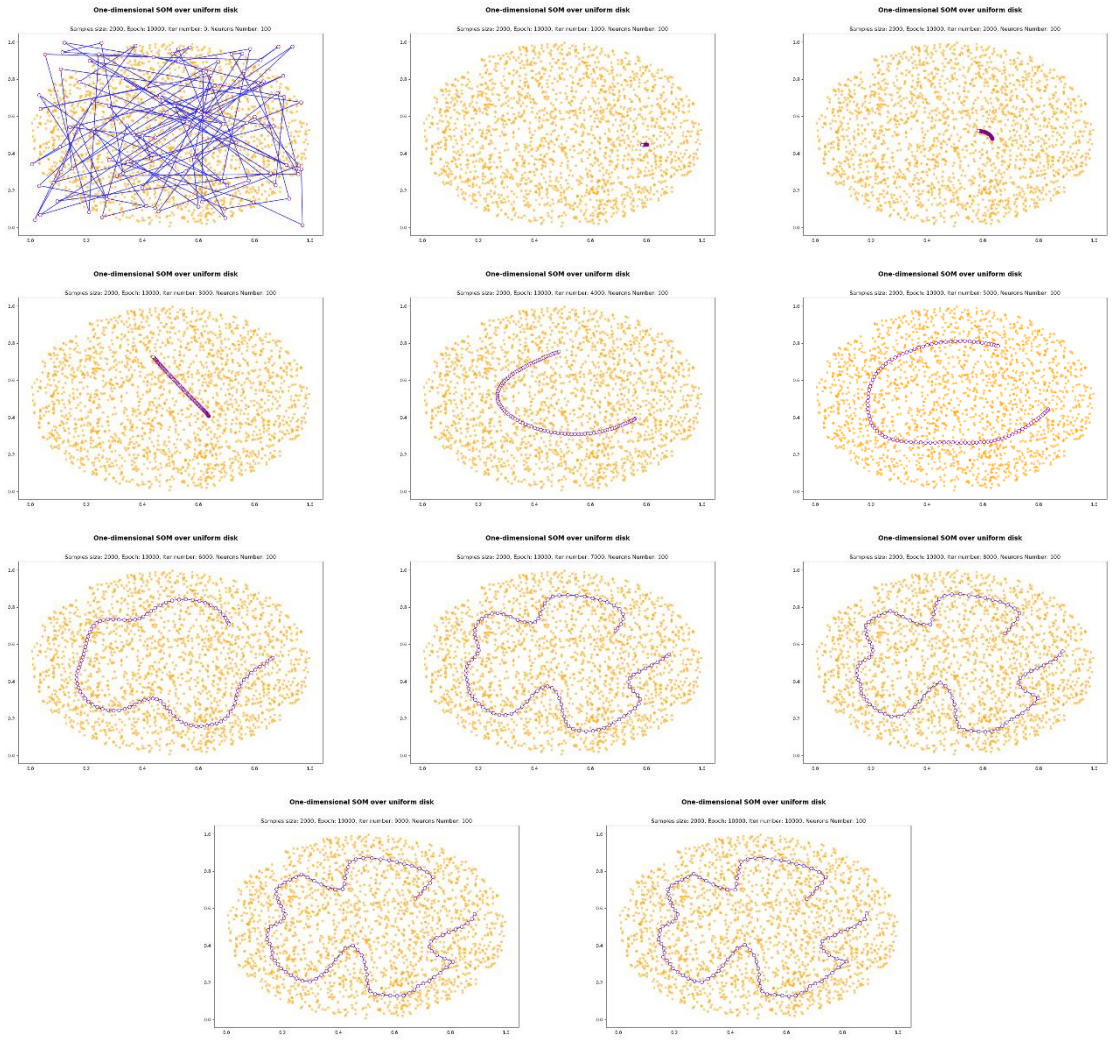
May 2022

Question A.1:

100 neurons in a one-dimensional SOM over uniform disk

Kohonen (SOM) Map Evolution:

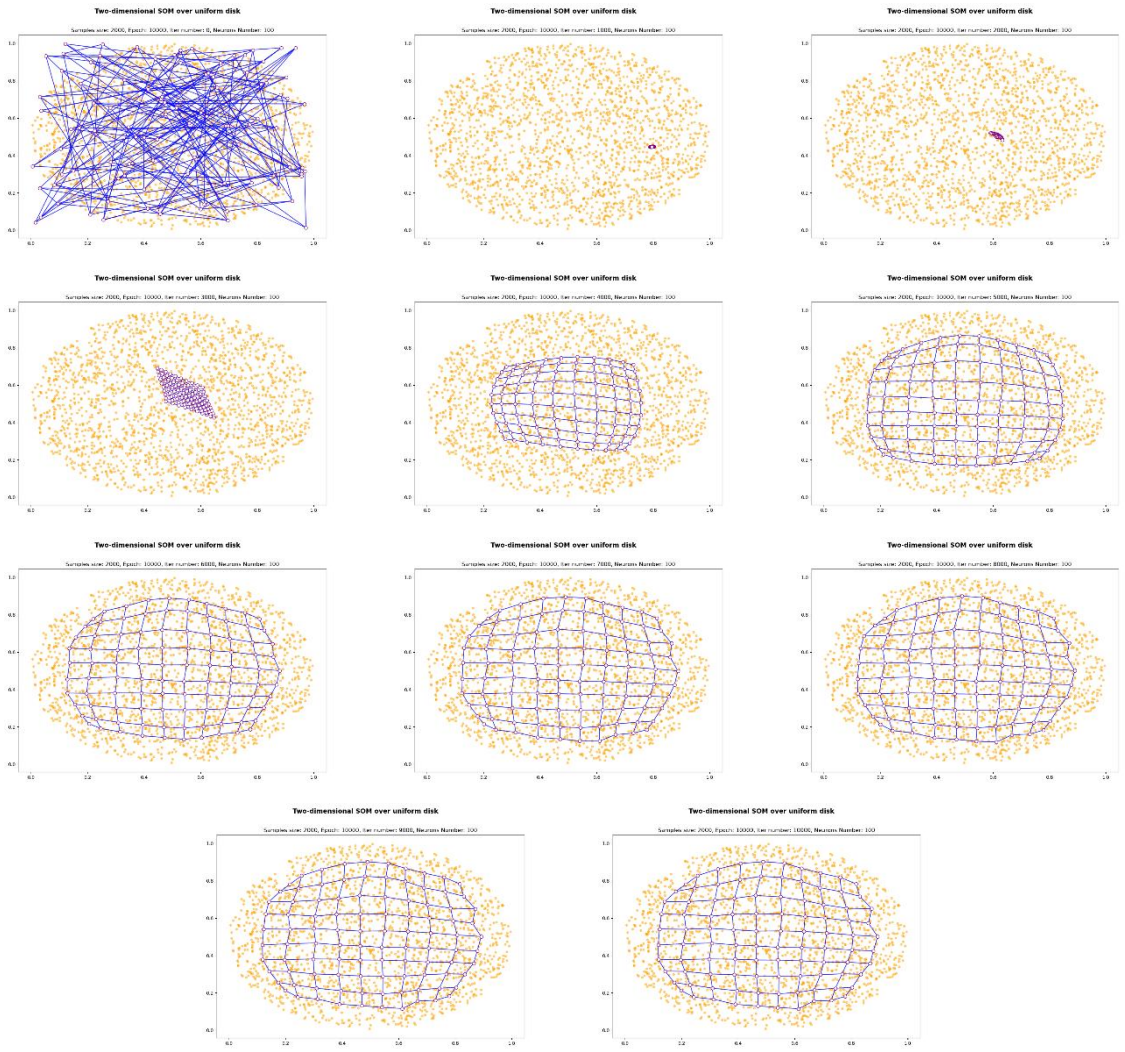
Fitting a line of 100 neurons in a one-dimensional SOM over a disk shape in a uniform way.



10X10 neurons in a two-dimensional SOM over uniform disk

Kohonen (SOM) Map Evolution:

Fitting a matrix of 10x10 neurons in a two-dimensional SOM over a disk shape in a uniform way.



Describe what happens as the number of iterations of algorithm increases?

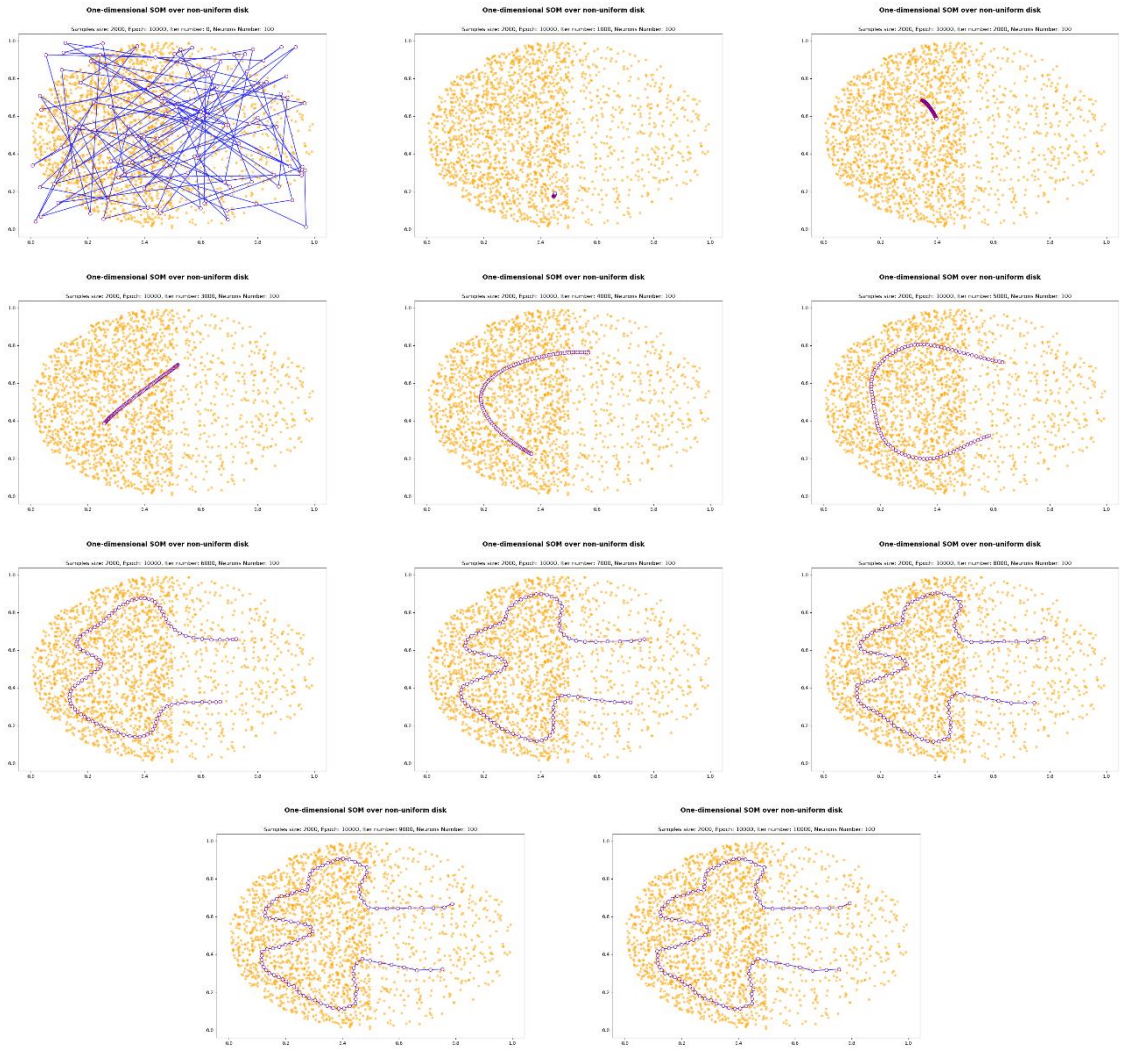
The result is as expected, as more as the number of iterations grow, each data point (input) "pulls" the nearest neuron (winner neuron) and his neighbor neurons. We've got a kind of shape that tries to spread over evenly on the area because we use a uniform distribution.

Question A.2:

100 neurons in a one-dimensional SOM over non-uniform-1 disk

Kohonen (SOM) Map Evolution:

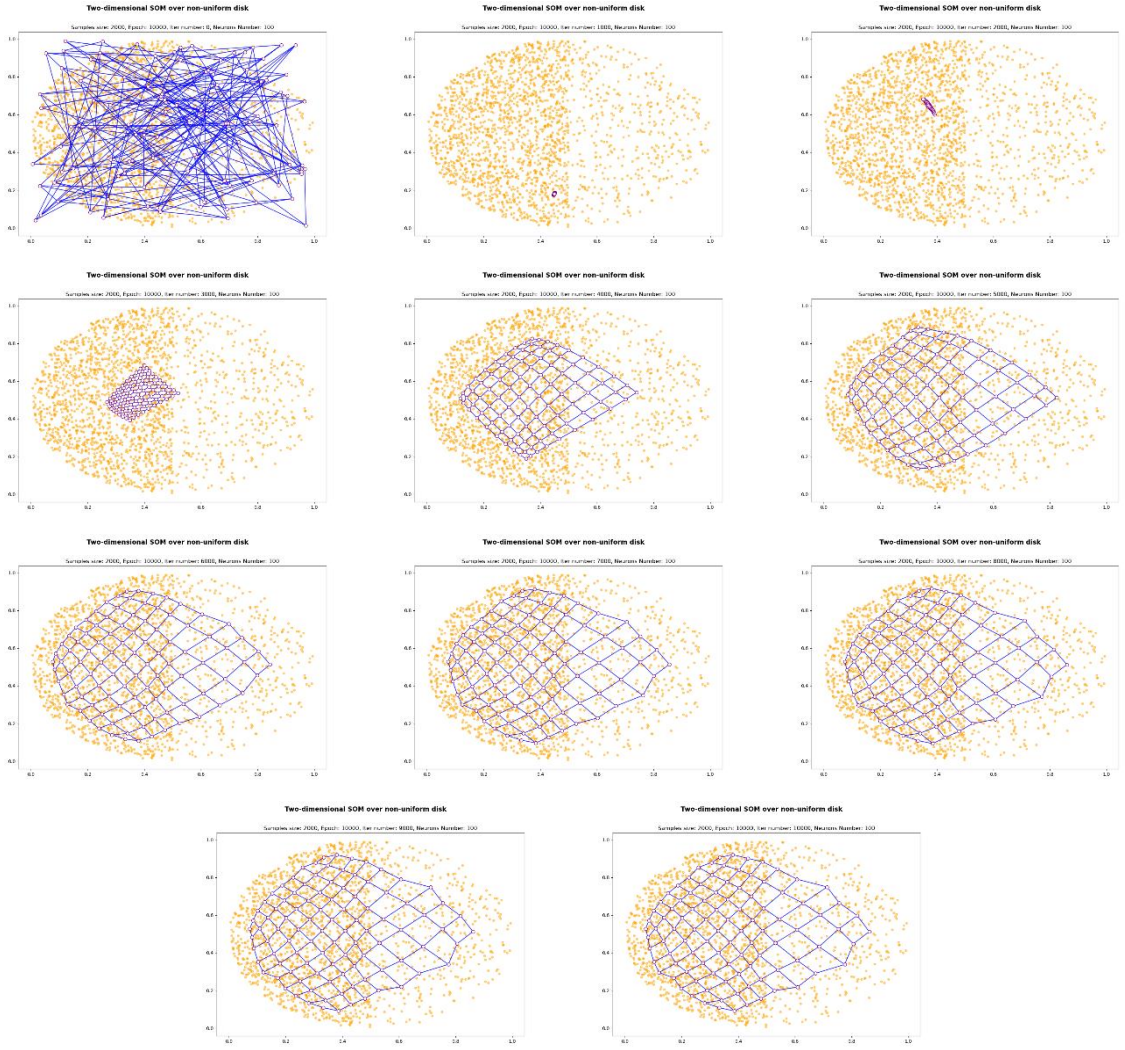
Fitting a line of 100 neurons in a one-dimensional SOM over a disk shape in a non-uniform way such that the likelihood of picking a point in the dataset is 80% higher at the left side of the disk.



10X10 neurons in a two-dimensional SOM over non-uniform-1 disk

Kohonen (SOM) Map Evolution:

Fitting a matrix of 10x10 neurons in a two-dimensional SOM over a disk shape in a non-uniform way such that the likelihood of picking a point in the dataset is 80% higher at the left side of the disk.



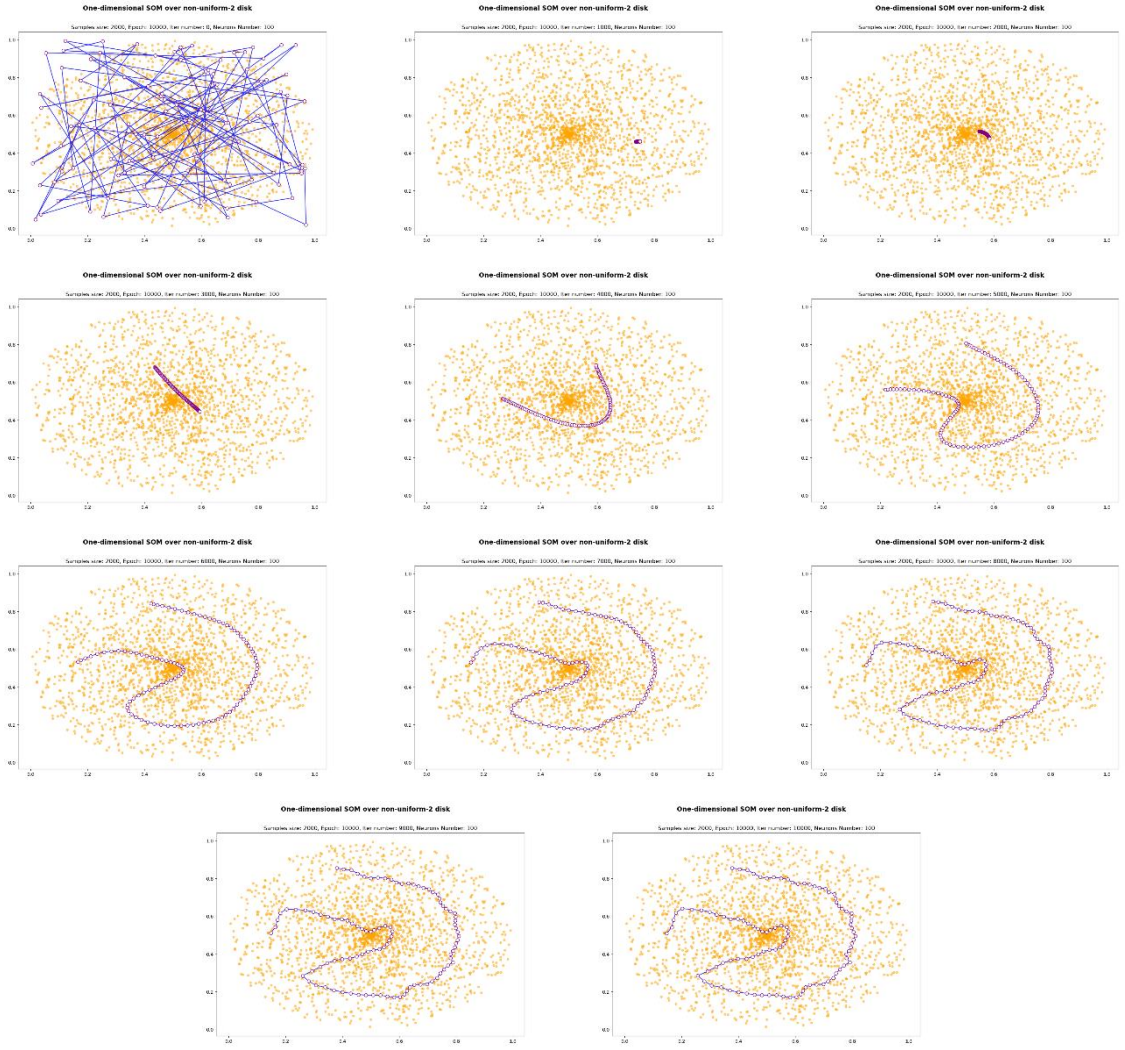
Describe what happens as the number of iterations of algorithm increases?

Most of the neurons stay at the left because most of the data points are this side of the disk, you can see that as more as the neurons are closer to the left side, the edges between them are shorter than on the right side.

100 neurons in a one-dimensional SOM over non-uniform-2 disk

Kohonen (SOM) Map Evolution:

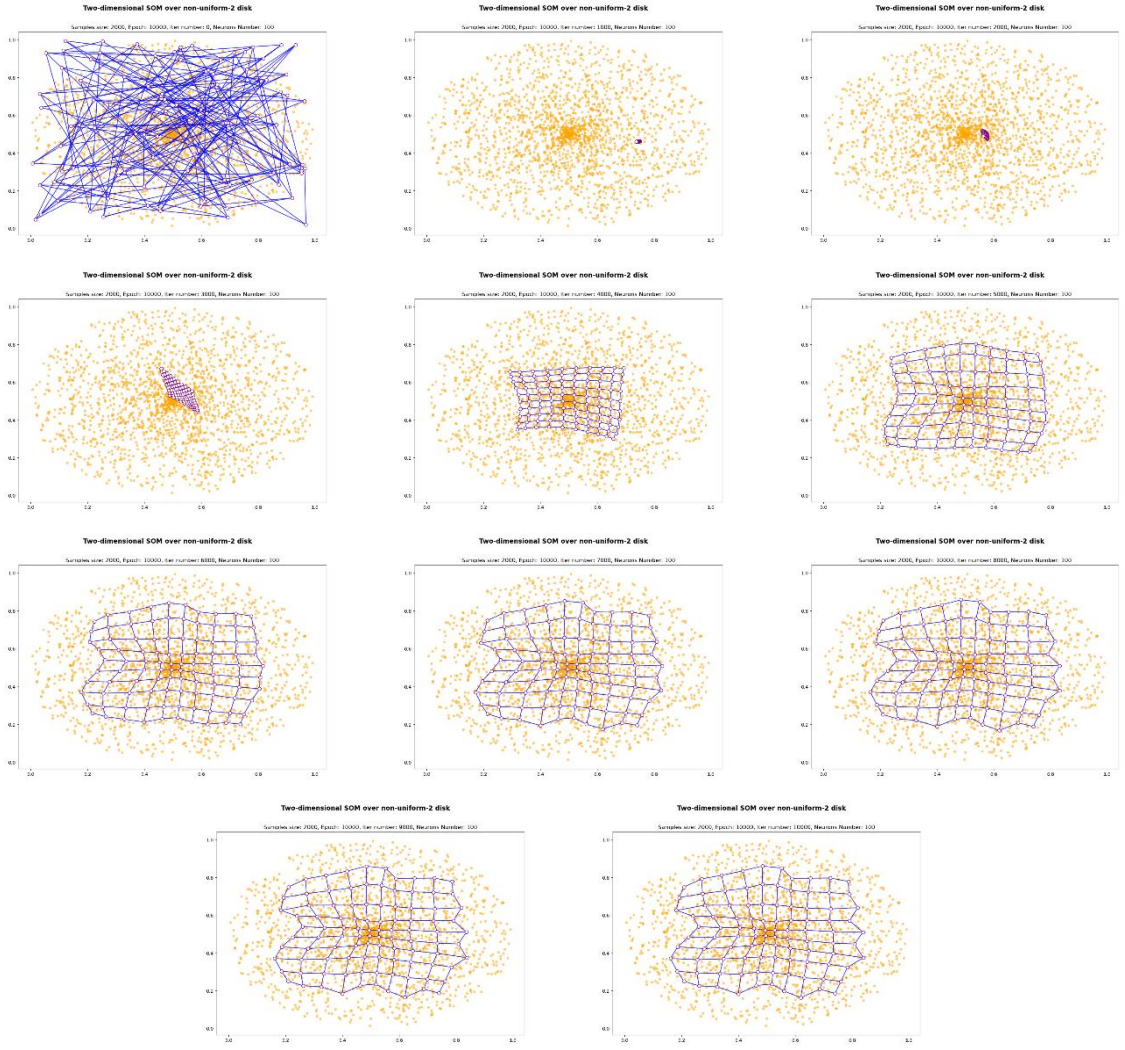
Fitting a line of 100 neurons in a one-dimensional SOM over a disk shape in a non-uniform way such that the likelihood of picking a point in the dataset is higher at the center of the disk.



10X10 neurons in a two-dimensional SOM over non-uniform-2 disk

Kohonen (SOM) Map Evolution:

Fitting a matrix of 10x10 neurons in a two-dimensional SOM over a disk shape in a non-uniform way such that the likelihood of picking a point in the dataset is higher at the center of the disk.



Describe what happens as the number of iterations of algorithm increases?

The neurons are pulled towards the center of the disk because most of the data points are in there. As for the line of neurons, you can see that at the beginning, the neurons are circling the center of the disk, but as the iterations grow, the neurons are spreading over the disk. The edges between the neurons are shorter as you proceed towards the center.

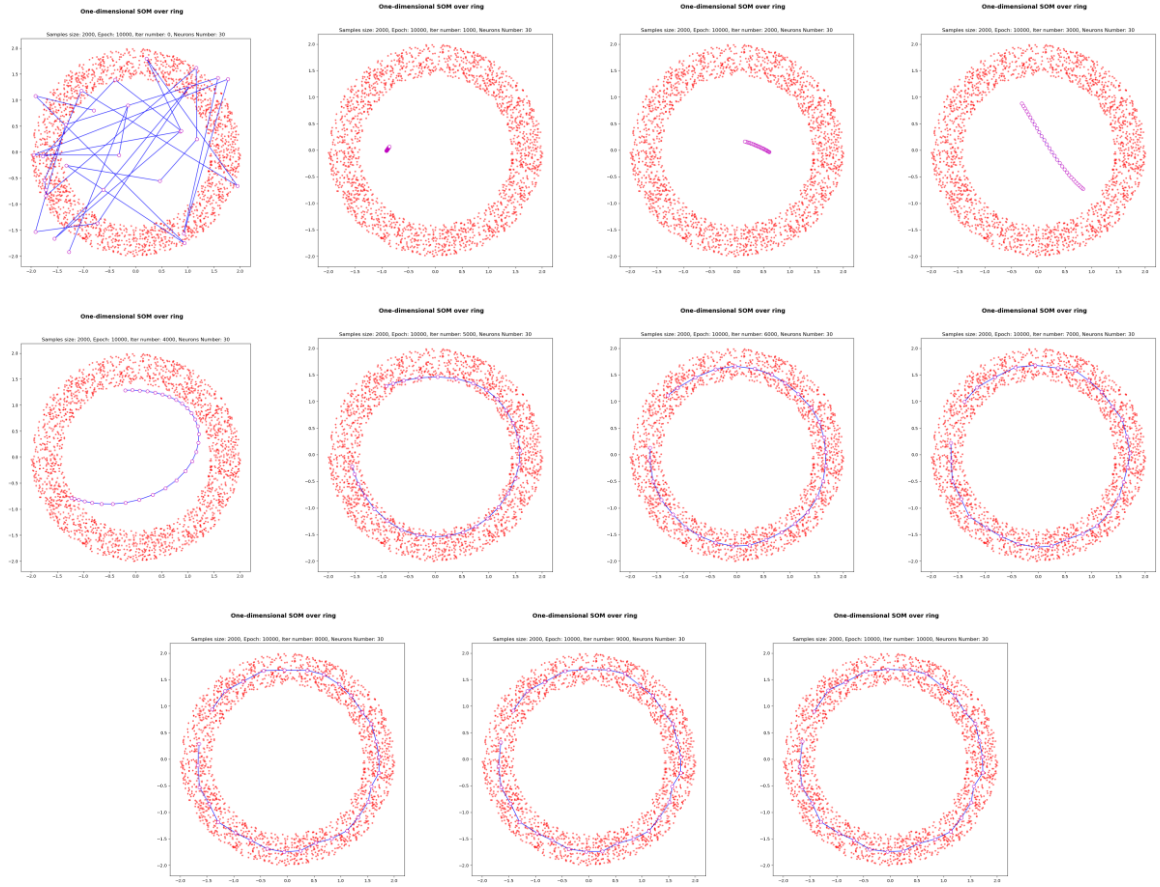
Question A.3:

30 neurons in a one-dimensional SOM over a ring

Kohonen (SOM) Map Evolution:

Fitting a line of 30 neurons in a one-dimensional SOM over a "ring" shape such that:

$$2 \leq x^2 + y^2 \leq 4$$



Describe what happens as the number of iterations of algorithm increases?

The neurons are pulled towards the ring as the number of iterations grows. At the beginning of the process, we can see that that the line of neurons is starting to spread over. As we proceed with the iterations, the edges between the neurons are getting longer and the neurons are pulled to the middle of the data points on the ring, in such a way that a circle is closing.

Kohonen (SOM Algorithm):

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


def find_max_min(my_data):

    maxX = my_data[:,0].max()
    maxY = my_data[:,1].max()

    minX = my_data[:,0].min()
    minY = my_data[:,1].min()

    return maxX, maxY, minX, minY


def Gaussian(neurons_index,center=None,sigma=0.5):

    if center == None:
        center = np.array(list(neurons_index))//2
    d = 0
    for i in range(len(neurons_index)):
        d += ((center[i]-neurons_index[i])/float(len(neurons_index[i])))**2

    dist_sqrt = np.sqrt(d)/np.sqrt(len(neurons_index))
    h_x = np.exp(-dist_sqrt**2/sigma**2)

    return h_x

def draw_report(self, num_of_iter, epochs, opt):

    fig = plt.figure(figsize=(12, 8))
    fig.suptitle(opt, fontsize=14, fontweight='bold')
    axes = fig.add_subplot()

    x, y = self.samples[:, 0], self.samples[:, 1]
    plt.scatter(x, y, alpha=0.5, color='orange', marker='.', s=80)

    x, y = self.som_weight[..., 0], self.som_weight[..., 1]

    if len(self.som_weight.shape) > 2:

        for i in range(self.som_weight.shape[0]):
            plt.plot(x[i, :], y[i, :], 'b', alpha=0.8)
        for i in range(self.som_weight.shape[1]):
            plt.plot(x[:, i], y[:, i], 'b', alpha=0.8)
    else:
        plt.plot(x, y, 'b', alpha=0.8)

    plt.scatter(x, y,  s=50, facecolor='w', edgecolor='purple', zorder=10)
    axes.set_title("Samples size: " + str(len(self.samples)) + ", "
                   + "Epoch: " + str(epochs) + ", "
                   + "Iter number: " + str(num_of_iter) + ", "
                   + "Neurons Number: " + str(self.som_weight.shape[0]))
    save_file = opt + "_{}.png"
    plt.savefig(save_file.format(str(num_of_iter)))
    plt.show()


def init_som_weight(maxX, maxY, minX, minY, shape):
    if len(shape) > 2:
        som_weight = np.array([[[np.random.uniform(minX, maxX), np.random.uniform(minY, maxY)] for i in
range(shape[0])] for j in range(shape[1])])
    else:
        som_weight = np.array([[np.random.uniform(minX, maxX), np.random.uniform(minY, maxY)] for i in
range(shape[0])])
    return som_weight


class SOM:

    def __init__(self, *args):
        ''' Initialize som '''

        last_element_index = len(args) - 1
        tup = args[:last_element_index]
        self.som_shape = np.zeros(tup)
        self.samples = args[last_element_index]

        maxX, maxY, minX, minY = find_max_min(args[last_element_index])
        self.som_weight = init_som_weight(maxX, maxY, minX, minY, self.som_shape.shape)
```

```
def learn(self, epochs=10000, sigma=(10, 0.001), lrate=(1.0, 0.001), report_iter=1000, opt=None):

    sigma_i, sigma_f = sigma
    lrate_i, lrate_f = lrate

    for i in range(epochs):
        # Adjust learning rate and neighborhood
        t = i/float(epochs)

        lrate = lrate_i*(lrate_f/float(lrate_i))**t
        sigma = sigma_i*(sigma_f/float(sigma_i))**t

        # Get random sample
        index = np.random.randint(0,self.samples.shape[0])
        data = self.samples[index]

        # Get index of nearest node (minimum distance)

        min_dist = ((data-self.som_weight)**2).sum(axis=-1)
        winner = np.unravel_index(np.argmin(min_dist), min_dist.shape)

        if len(min_dist.shape) > 1:
            rows = np.arange(min_dist.shape[0]).reshape(-1, 1) + np.zeros((1, min_dist.shape[0]))
            cols = np.arange(float(0), float(min_dist.shape[0])) + np.zeros((min_dist.shape[0], 1))
            neurons_index = [rows, cols]
        else:
            line = np.arange(float(min_dist.shape[0]))
            neurons_index = [line]

        # Generate a Gaussian centered on winner
        G = Gaussian(neurons_index, winner, sigma)
        G = np.nan_to_num(G)

        # Move nodes towards sample according to Gaussian
        delta = data-self.som_weight

        if i % report_iter == 0 or i+1 == epochs:
            if i+1 == epochs:
                i = epochs
            draw_report(self, i, epochs, opt)

        for x in range(self.som_weight.shape[-1]):
            self.som_weight[..., x] += lrate * G * delta[..., x]
```

Main:

```
import random
import numpy as np
from Kohonen import SOM
import math
from numpy.random.mtrand import dirichlet


def NonUniformRandom2(radius=1):
    r = radius * math.sqrt(np.random.random()) * 0.5
    theta = 2 * np.pi * np.random.rand()

    return 0.5 + r * np.cos(theta), 0.5 + r * np.sin(theta)


def UniformRandom(radius=1):
    r = radius * math.sqrt(np.random.random()) * 0.5
    theta = 2 * np.pi * np.random.rand()

    return 0.5 + r * np.cos(theta), 0.5 + r * np.sin(theta)


def NonUniformRandom1(radius=1):
    r = radius * np.random.random() * 0.5
    theta = 2 * np.pi * np.random.rand()

    return 0.5 + r * np.cos(theta), 0.5 + r * np.sin(theta)


def ReplicateNTimes(func, rad, Ntrials=1000):
    xpoints, ypoints = [], []
    for _ in range(Ntrials):
        xp, yp = func(rad)
        xpoints.append(xp)
        ypoints.append(yp)
    dist = (xpoints, ypoints)
    return dist


def ReplicateNTimes2(func, rad, Ntrials=1000):
    i = 0
    xpoints, ypoints = [], []
    while i < Ntrials:
        xp, yp = func(rad)
        if xp <= 0.5 and i <= Ntrials * 0.8:
            xpoints.append(xp)
            ypoints.append(yp)
            i = i + 1
        if xp > 0.5 and i > Ntrials * 0.8:
            xpoints.append(xp)
            ypoints.append(yp)
            i = i + 1
    dist = (xpoints, ypoints)
    return dist


def InitializesSamples(size=2000, task=1.0):
    samples = np.zeros((size, 2))
    np.random.seed(11)
    Ntrials = size
    radius = 1

    # uniform disk
    if task == 1.0:
        dist = ReplicateNTimes(UniformRandom, radius, Ntrials=Ntrials)
        samples[:, 0] = dist[0]
        samples[:, 1] = dist[1]

    # ring
    if task == 1.1:
        c = 0
        while c < size:
            x = random.uniform(-2, 2)
            y = random.uniform(-2, 2)
            if 2 <= x ** 2 + y ** 2 <= 4:
                samples[c, 0] = x
                samples[c, 1] = y
                c = c + 1

    # non-uniform half disk : 80% in left
    if task == 1.2:
        dist = ReplicateNTimes2(NonUniformRandom2, radius, Ntrials=Ntrials)
        samples[:, 0] = dist[0]
        samples[:, 1] = dist[1]
```

```
    # non-uniform disk : most of the points in the center
    if task == 1.3:
        dist = ReplicateNTimes(NonUniformRandom1, radius, Ntrials=Ntrials)
        samples[:, 0] = dist[0]
        samples[:, 1] = dist[1]

    return samples


def main():
    samples0 = InitializesSamples()
    samples1 = InitializesSamples(task=1.1)
    samples2 = InitializesSamples(task=1.2)
    samples3 = InitializesSamples(task=1.3)

    op0 = "One-dimensional SOM over uniform disk"
    op1 = "One-dimensional SOM over ring"
    op2 = "One-dimensional SOM over non-uniform disk"
    op3 = "One-dimensional SOM over non-uniform-2 disk"
    op2D_0 = "Two-dimensional SOM over uniform disk"
    op2D_1 = "Two-dimensional SOM over ring"
    op2D_2 = "Two-dimensional SOM over non-uniform disk"
    op2D_3 = "Two-dimensional SOM over non-uniform-2 disk"


    """ QUESTION A.1: """
    # som = SOM(100, 2, samples0)
    # som.learn(opt=op0)

    # som = SOM(10, 10, 2, samples0)
    # som.learn(opt=op2D_0)

    """ QUESTION A.2: """
    # non-uniform disk
    # som = SOM(100, 2, samples2)
    # som.learn(opt=op2)

    # som = SOM(10, 10, 2, samples2)
    # som.learn(opt=op2D_2)

    # non-uniform-2 disk
    # som = SOM(100, 2, samples3)
    # som.learn(opt=op3)

    # som = SOM(10, 10, 2, samples3)
    # som.learn(opt=op2D_3)

    """ QUESTION A.3: """
    som = SOM(30, 2, samples1)
    som.learn(opt=op1)


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

1 Question B.1

1.1 Tables and Graphs

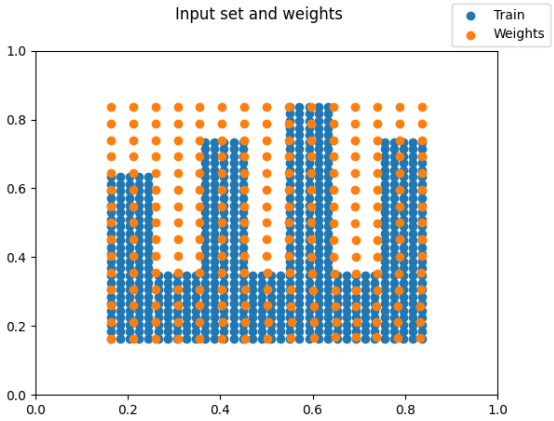


Figure 1: Sampled hand and weights

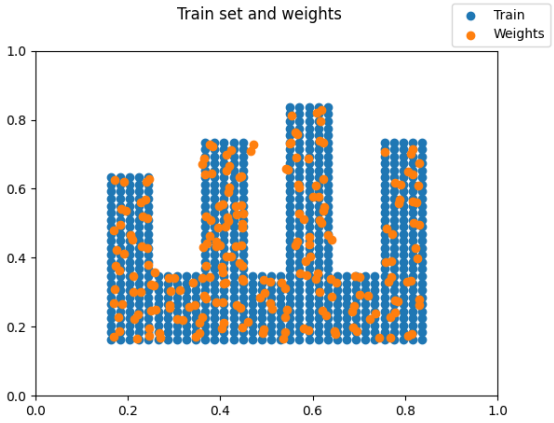


Figure 2: Final output

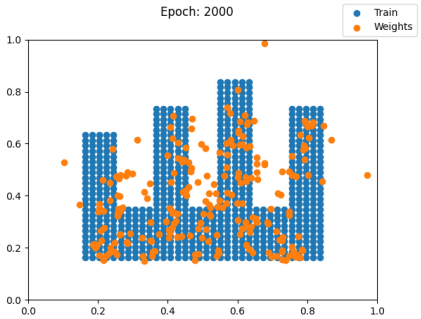


Figure 3: Iteration 2000

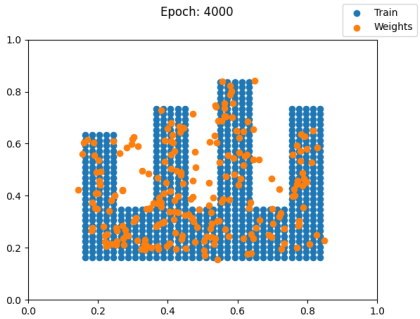


Figure 4: Iteration 4000

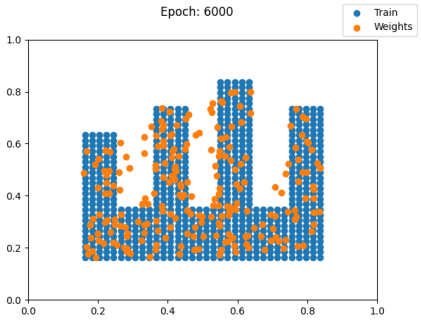


Figure 5: Iteration 6000

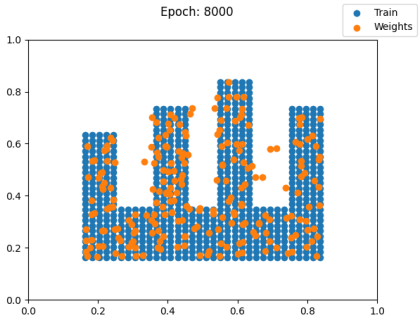


Figure 6: Iteration 8000

Figure 7: Weight change over 10000 iterations

2 Discussion & Results

A key observation found in this project, is that self-organizing maps are very sensitive to initialization parameters, and we have plenty of parameters. For this question, a Gaussian neighbourhood function was used, with exponential weight decay for the scalars. We had 4 scalars in total: learning rate and it's decay, neighbourhood scale and it's decay. Setting the neighbourhood function impact scalar too high resulted in many of the weights clumping up together. Setting it too low yielded many points being outside the hand. Even though the learning rate was simpler to adjust, it indirectly impacted the neighbourhood function. An another observation is that the initial position of the weights in Kohonen's algorithm is very important. Since the algorithm is based around weight distances to the input, a bad starting distribution of the weights may yield bad results, since this will require much more iterations, and our parameters to decay over time. Overall self organizing maps have shown to be very effective in learning the shape of the input data. Though it does need much time to converge and is very sensitive to initialization parameters.

2.1 Code

A package code was used as a basis for the Kohonen algorithm: https://github.com/nicomignoni/SOM
Though 90% of the code was re-written.

2.1.1 File som/mapping.py:

```
import torch
import numpy as np
from math import exp
from tqdm import trange
import matplotlib.pyplot as plt



class SOM:
    def __init__(
        self,
        nFeatures,
        gridShape,
        weights,
        alpha0=0.5,
        t_alpha=25,
        sigma0=2,
        t_sigma=25,
        scale=True,
        history=True,
        shuffle=True,
    ):

        self.nFeatures = nFeatures
        self.gridShape = gridShape
        self.weights = weights
        self.nWeights = self.weights.shape[0]

        self._unitGrid = self._GenerateUnitGrid()

        self.alpha = alpha0
        self.alphaMaxTime = t_alpha
        self.sigma = sigma0
        self.sigmaMaxTime = t_sigma
        self.allowScale = scale
        self.allowHistory = history
        self.allowShuffle = shuffle
```

```
def _GenerateUnitGrid(self):
    x = np.arange(self.gridShape[0])
    y = np.arange(self.gridShape[1])
    XX, YY = np.meshgrid(x, y)

    unitGrid = np.vstack([XX.reshape(−1), YY.reshape(−1)]).transpose()
        ↪ .reshape(self.gridShape[0],self.gridShape[1],self.nFeatures)
    unitGrid = torch.tensor(unitGrid)

    return unitGrid

def ScalarUpdate(self, value, currentTime, maxTime):
    return value ∗ exp(−currentTime / maxTime)

def GuassianNeighbourhood(self, BMUGridCoords, sigma):

    distanceMatrix = torch.zeros(self.gridShape)

    distanceMatrix = torch.abs((self._unitGrid − torch.Tensor(
        ↪ BMUGridCoords)))
    distanceMatrix = distanceMatrix.sum(axis=2)
    distanceMatrix = torch.pow(distanceMatrix,2)

    return torch.exp(−distanceMatrix / (2 ∗ sigma∗∗2)).reshape(
        (self.nWeights, 1)
    )


def fit(self, input, epochs):
    n_samples, n_attributes = input.shape

    # From numpy conversion
    input = torch.from_numpy(input).type(torch.double)

    # Shuffling
    if self.allowShuffle:
        indices = torch.randperm(n_samples)
        input = input[indices, :]

    # Scaling W in the same range as X
    if self.allowScale:
        self.weights = self.weights ∗ (
            torch.max(input) − torch.min(input)
        ) + torch.min(input)

    # Record each W for each t (debugging)
```

```
        if self.allowHistory:
            self.history = self.weights.reshape(1, self.nWeights, n_attributes)

        # The training loop
        for epochIndex in trange(epochs):
            sample = input[epochIndex % n_samples, :]
            distances = sample - self.weights

            # Find the winning point
            euclideanDistances = torch.pow((distances), 2).sum(axis=1)
            BMUIndex = torch.argmin(euclideanDistances)
            BMUGridCoords = (int(BMUIndex/self.gridShape[0]),int(
                ↪ BMUIndex%self.gridShape[0]))

            # Update the learning rate
            alpha = self.ScalarUpdate(self.alpha, epochIndex, self.
                ↪ alphaMaxTime)

            # Update the neighborhood size
            sigma = self.ScalarUpdate(self.sigma, epochIndex, self.
                ↪ sigmaMaxTime)

            # Evaluate the topological neighborhood
            changeRate = self.GuassianNeighbourhood(BMUGridCoords,
                ↪ sigma)

            # Update weights
            self.weights += alpha * changeRate * (distances)

            if self.allowHistory:
                self.history = torch.cat(
                    (
                        self.history,
                        self.weights.reshape(1, self.nWeights, n_attributes),
                    ),
                    axis=0,
                )
```

2.1.2 File ShapeGen.py:

```
import re
from typing import List



class Shape():
    def __init__(self) -> None:
        pass

    def DecidePoint(self,point):
        raise NotImplementedError()

class Rectangle():
    def __init__(self,point1,point2) -> None:

        self._point_1 = point1
        self._point_2 = point2

        if self._point_1[0] > self._point_2[0]:
            temp = self._point_2[0]
            self._point_2[0] = self._point_1[0]
            self._point_1[0] = temp

        if self._point_1[1] > self._point_2[1]:
            temp = self._point_2[1]
            self._point_2[1] = self._point_1[1]
            self._point_1[1] = temp


    def DecidePoint(self,point):
        isOk = True

        isOk = isOk and point[0] >= self._point_1[0] and point[0] <= self.
            ↪ _point_2[0]
        isOk = isOk and point[1] >= self._point_1[1] and point[1] <= self.
            ↪ _point_2[1]

        return isOk


class ShapeGen():
    def __init__(self,shapes: List[Shape]) -> None:
        self._shapes = shapes
```

```
def DecidePoint(self,point):

    pointIsOk: bool = False

    for shape in self._shapes:
        pointIsOk = pointIsOk or shape.DecidePoint(point)

    return pointIsOk

def FilterPoints(self, points):
    outpoints = []

    for iterPoint in points:
        if self.DecidePoint(iterPoint): outpoints.append(iterPoint)

    return outpoints
```

2.1.3 File Question B.py:

```
import matplotlib.pyplot as plt
import numpy as np
import torch
from ShapeGen import ShapeGen

from ShapeGen import Rectangle, ShapeGen
from som.mapping import SOM


SHOW_HISTORY = True
HISTORY_STEPSIZE = 500
SHOW_OUTPUT = True
SHOW_INPUT = False

USE_EXAMPLE_DATA = False

GRID_SHAPE = (15, 15)
INPUT_DIM = 50
EPOCHS = 10000
ALPHA_SCALE = 1
SIGMA_SCALE = 1
ALPHA = 2
SIGMA = 1


x = np.linspace(0, 1, INPUT_DIM)
y = np.linspace(0, 1, INPUT_DIM)
XX, YY = np.meshgrid(x, y)

data = np.vstack([XX.reshape(-1), YY.reshape(-1)]).transpose()


palmShape = Rectangle((0.15, 0.15), (0.85, 0.35))
finger1 = Rectangle((0.15, 0.25), (0.25, 0.65))
finger2 = Rectangle((0.35, 0.35), (0.45, 0.75))
finger3 = Rectangle((0.55, 0.35), (0.65, 0.85))
finger4 = Rectangle((0.75, 0.35), (0.85, 0.75))

allShapes = [palmShape, finger1, finger2, finger3, finger4]

shapeFilter = ShapeGen(allShapes)

data = np.array(shapeFilter.FilterPoints(data))
```

```
x = np.linspace(0, 1, GRID_SHAPE[0])
y = np.linspace(0, 1, GRID_SHAPE[1])
XX, YY = np.meshgrid(x, y)

weights = np.vstack([XX.reshape(−1), YY.reshape(−1)]).transpose()
weights = torch.tensor(weights)


if USE_EXAMPLE_DATA:

    from sklearn.datasets import load_iris
    from sklearn.decomposition import PCA

    dataset = load_iris()
    train = dataset.data
    pca = PCA(n_components=2)
    train_pca = pca.fit_transform(train)
    data = train_pca


model = SOM(
    nFeatures=2,
    gridShape=GRID_SHAPE,
    weights=weights,
    alpha0=ALPHA,
    t_alpha=ALPHA_SCALE * EPOCHS,
    sigma0=SIGMA,
    t_sigma=SIGMA_SCALE * EPOCHS,
    scale=True,
    history=True,
    shuffle=True,
)

if SHOW_INPUT:
    weights = model.weights

    fig, ax = plt.subplots()
    fig.suptitle("Input_set_and_weights")

    t = ax.scatter(data[:, 0], data[:, 1])
    w = ax.scatter(weights[:, 0], weights[:, 1])

    fig.legend((t, w), ("Train", "Weights"))
    plt.xlim((0, 1))
    plt.ylim((0, 1))
    plt.show()
```

```
model.fit(data, EPOCHS)

if SHOW_OUTPUT:
    weights = model.weights

    fig, ax = plt.subplots()
    fig.suptitle("Train set and weights")

    t = ax.scatter(data[:, 0], data[:, 1])
    w = ax.scatter(weights[:, 0], weights[:, 1])

    fig.legend((t, w), ("Train", "Weights"))
    plt.xlim((0, 1))
    plt.ylim((0, 1))
    plt.show()


if SHOW_HISTORY:

    # Plot the train dataset and the weights
    historyNum = model.history.shape[0]

    for historyIndex in range(0, historyNum, HISTORY_STEPSIZE):
        weights = model.history[historyIndex, :, :]

        fig, ax = plt.subplots()
        fig.suptitle("Epoch: " + historyIndex.__str__())

        t = ax.scatter(data[:, 0], data[:, 1])
        w = ax.scatter(weights[:, 0], weights[:, 1])

        fig.legend((t, w), ("Train", "Weights"))
        plt.xlim((0, 1))
        plt.ylim((0, 1))
        plt.show()

torch.save(model.weights, "./model.pt")
```

3 Question B.2

3.1 Tables and Graphs

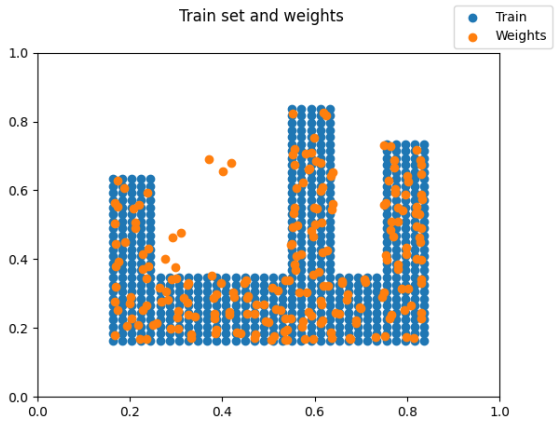


Figure 8: Re-trained output after removing one finger

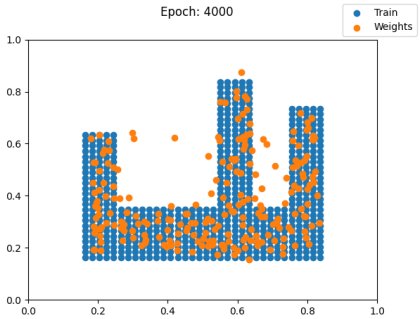


Figure 9: Iteration 4000

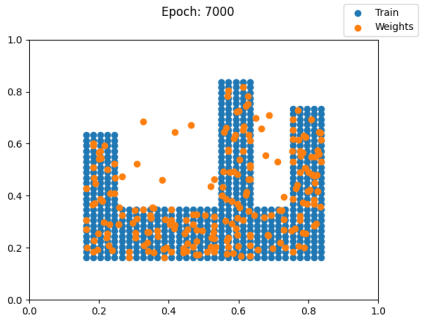


Figure 10: Iteration 7000

4 Discussion & Results

It is observerd that the mesh re-arranges itself to fit the new input data quite well. Since no inputs are recieved from the removed finger, the weights in the same area are redistributed to other parts by proximity. The distribution is also rather disorganized. Due to the stochastic nature of the algorithm, the selected points to distribute and their neighbourhood functions may affect points that are already in a rather good position. Though, with enough iteration, the algorithm does converge eventually to a good decision mesh. Overall, self organizing maps are very flexible and can be adjusted on the fly to new data changes, just like the brain.

4.1 Code

Previous code from Question B was used as a basis.

4.1.1 File Question B2.py:

```
import matplotlib.pyplot as plt
import numpy as np
import torch
from ShapeGen import ShapeGen

from ShapeGen import Rectangle, ShapeGen
from som.mapping import SOM


SHOW_HISTORY = True
HISTORY_STEPSIZE = 500
SHOW_OUTPUT = True
SHOW_INPUT = False

USE_EXAMPLE_DATA = False

GRID_SHAPE = (15, 15)
INPUT_DIM = 50
EPOCHS = 10000
ALPHA_SCALE = 1
SIGMA_SCALE = 1
ALPHA = 2
SIGMA = 1



x = np.linspace(0, 1, INPUT_DIM)
y = np.linspace(0, 1, INPUT_DIM)
XX, YY = np.meshgrid(x, y)

data = np.vstack([XX.reshape(-1), YY.reshape(-1)]).transpose()

palmShape = Rectangle((0.15, 0.15), (0.85, 0.35))
finger1 = Rectangle((0.15, 0.25), (0.25, 0.65))
finger2 = Rectangle((0.35, 0.35), (0.45, 0.75))
finger3 = Rectangle((0.55, 0.35), (0.65, 0.85))
finger4 = Rectangle((0.75, 0.35), (0.85, 0.75))

allShapes = [palmShape, finger1, finger3, finger4]

shapeFilter = ShapeGen(allShapes)
```

```
data = np.array(shapeFilter.FilterPoints(data))

x = np.linspace(0, 1, GRID_SHAPE[0])
y = np.linspace(0, 1, GRID_SHAPE[1])
XX, YY = np.meshgrid(x, y)

weights = torch.load("model.pt")


if USE_EXAMPLE_DATA:

    from sklearn.datasets import load_iris
    from sklearn.decomposition import PCA

    dataset = load_iris()
    train = dataset.data
    pca = PCA(n_components=2)
    train_pca = pca.fit_transform(train)
    data = train_pca


model = SOM(
    nFeatures=2,
    gridShape=GRID_SHAPE,
    weights=weights,
    alpha0=ALPHA,
    t_alpha=ALPHA_SCALE * EPOCHS,
    sigma0=SIGMA,
    t_sigma=SIGMA_SCALE * EPOCHS,
    scale=True,
    history=True,
    shuffle=True,
)

if SHOW_INPUT:
    weights = model.weights

    fig, ax = plt.subplots()
    fig.suptitle("Input set and weights")

    t = ax.scatter(data[:, 0], data[:, 1])
    w = ax.scatter(weights[:, 0], weights[:, 1])

    fig.legend((t, w), ("Train", "Weights"))
    plt.xlim((0, 1))
```

```
    plt.ylim((0, 1))
    plt.show()

model.fit(data, EPOCHS)

if SHOW_OUTPUT:
    weights = model.weights

    fig, ax = plt.subplots()
    fig.suptitle("Train set and weights")

    t = ax.scatter(data[:, 0], data[:, 1])
    w = ax.scatter(weights[:, 0], weights[:, 1])

    fig.legend((t, w), ("Train", "Weights"))
    plt.xlim((0, 1))
    plt.ylim((0, 1))
    plt.show()


if SHOW_HISTORY:

    # Plot the train dataset and the weights
    historyNum = model.history.shape[0]

    for historyIndex in range(0, historyNum, HISTORY_STEPSIZE):
        weights = model.history[historyIndex, :, :]

        fig, ax = plt.subplots()
        fig.suptitle("Epoch: " + historyIndex.__str__())

        t = ax.scatter(data[:, 0], data[:, 1])
        w = ax.scatter(weights[:, 0], weights[:, 1])

        fig.legend((t, w), ("Train", "Weights"))
        plt.xlim((0, 1))
        plt.ylim((0, 1))
        plt.show()
```