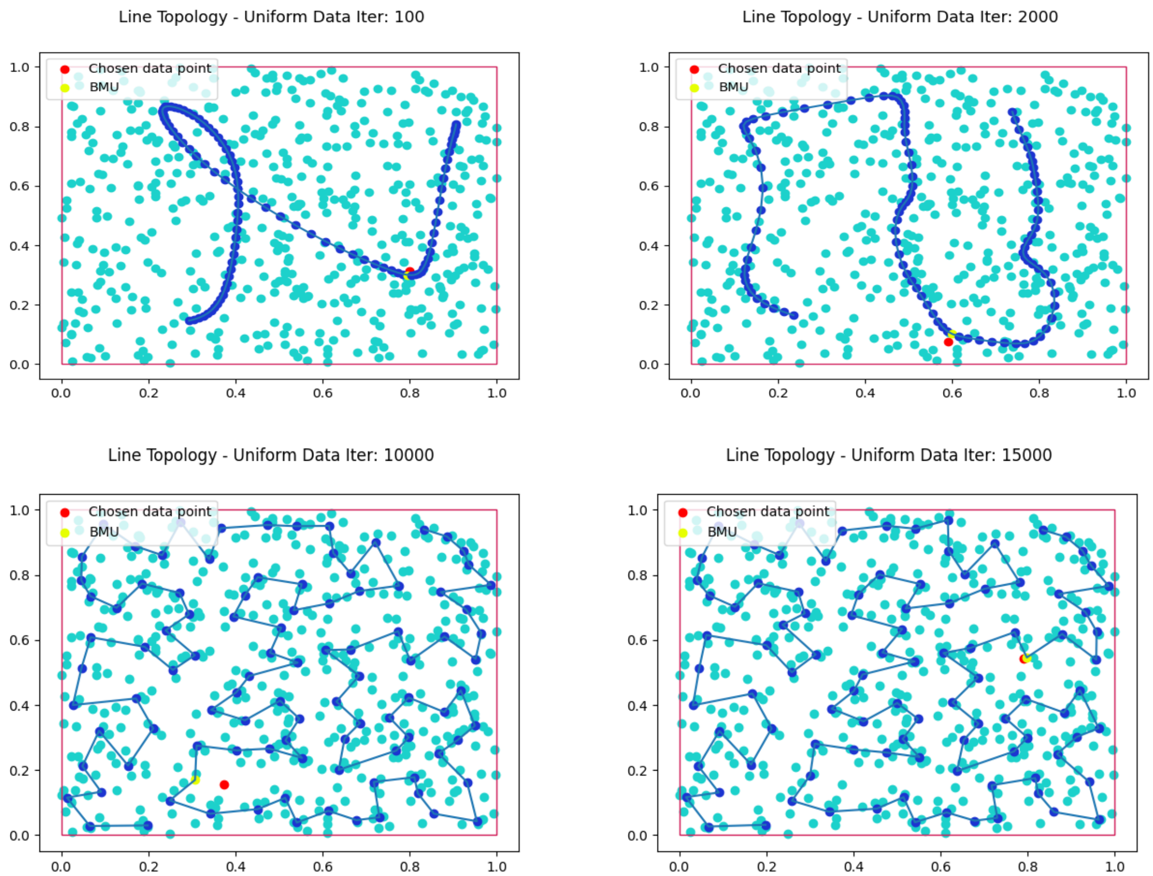
Project 2: part A and B

first name and T.Z. numbers
second name and T.Z. numbers

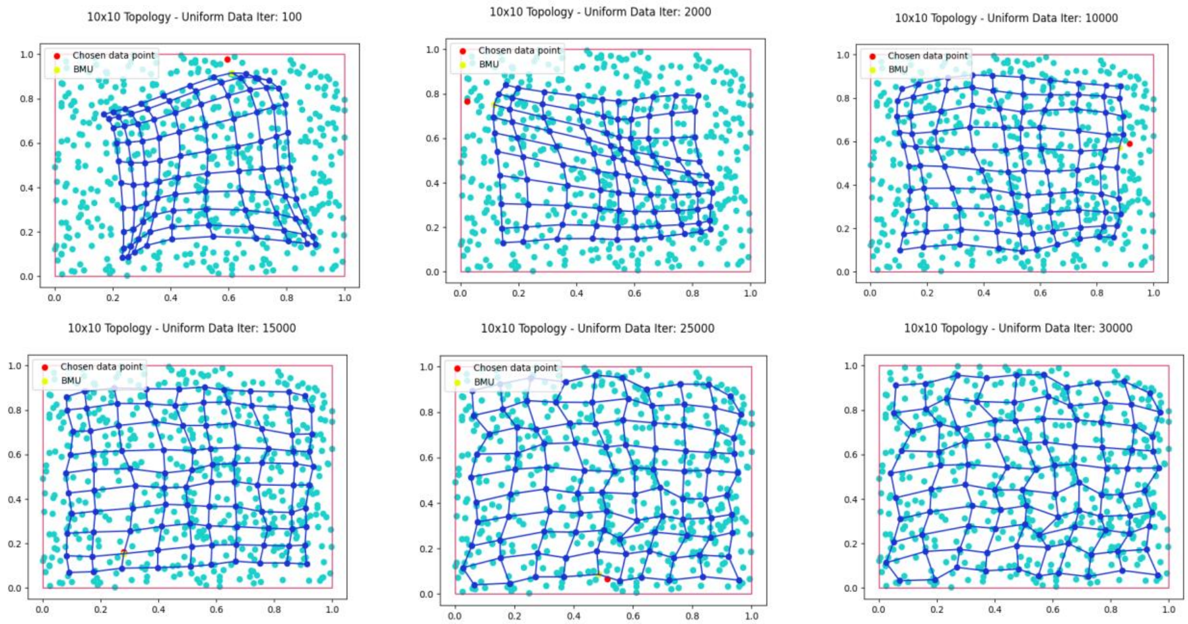
Part A:

1. **Discussions:**
   Describe what happens as the number of iterations of the algorithm increases?



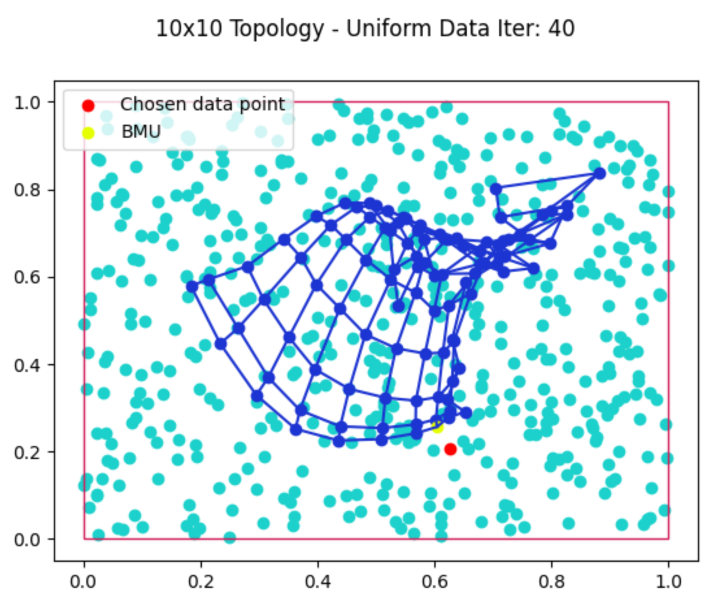
Link to images

By following the Kohonen algorithm (see image above), the neuron grid first disentangles, and then gradually fills in the entire space square so that neurons with adjacent indices are located close together. The new point P (chosen data point) attracts the nearest neuron (BMU) toward itself, but less so to its neighbors. Due to the large neighborhood distance at the beginning of the algorithm, large chunks of neighboring neurons in the input grid are pulled together toward P. With each subsequent stage, the neighborhood distance reduces, so that only the winner and possibly one or two very close neighbors are attracted to a new point. After completion (bottom right panel), individual neurons are close to a certain data area.



[Link to images]

Over time, the initial chaos slowly transforms into nearly perfect order, with the grid laid out uniformly in the data square, with only slight variations from a regular arrangement.

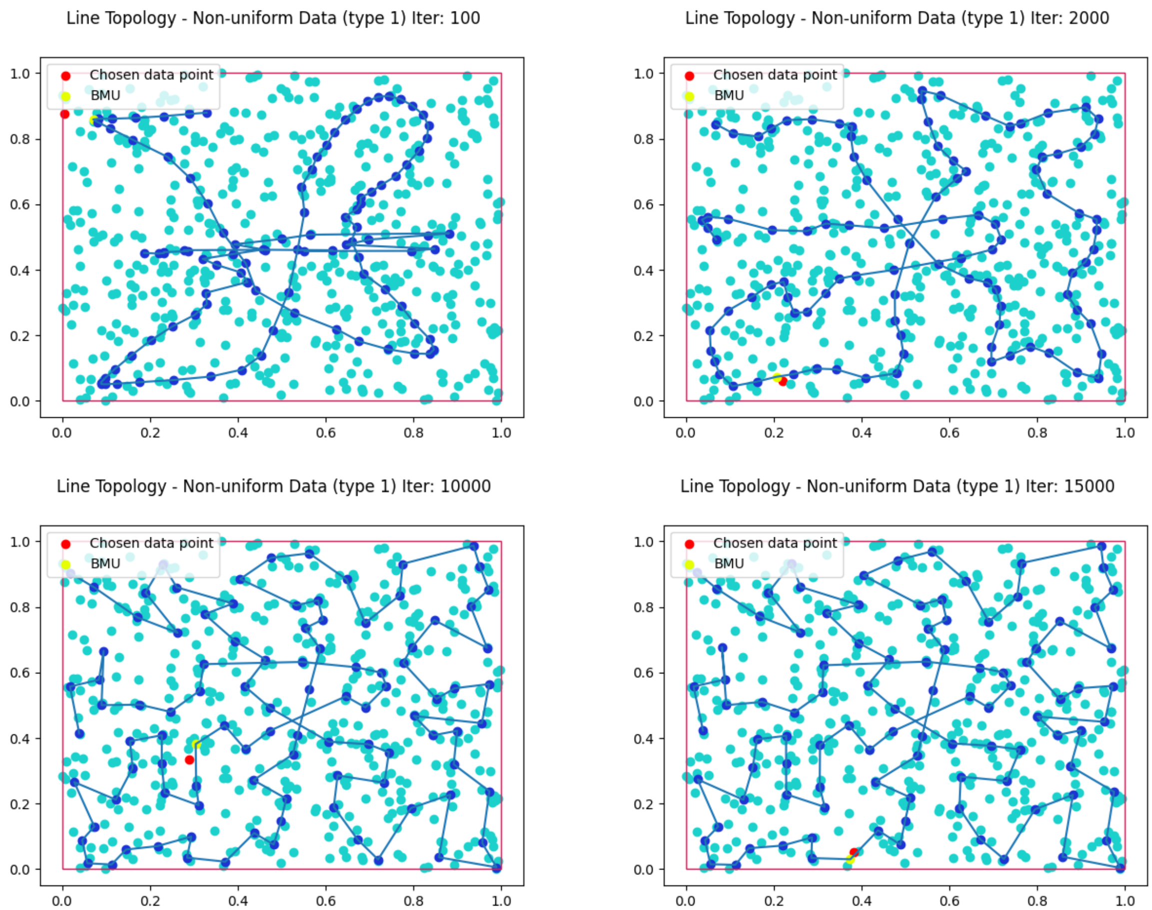


As we walk along, we notice a phenomenon called twist, in which the grid is crumpled. The effect will be more pronounced in non-uniform distributions.

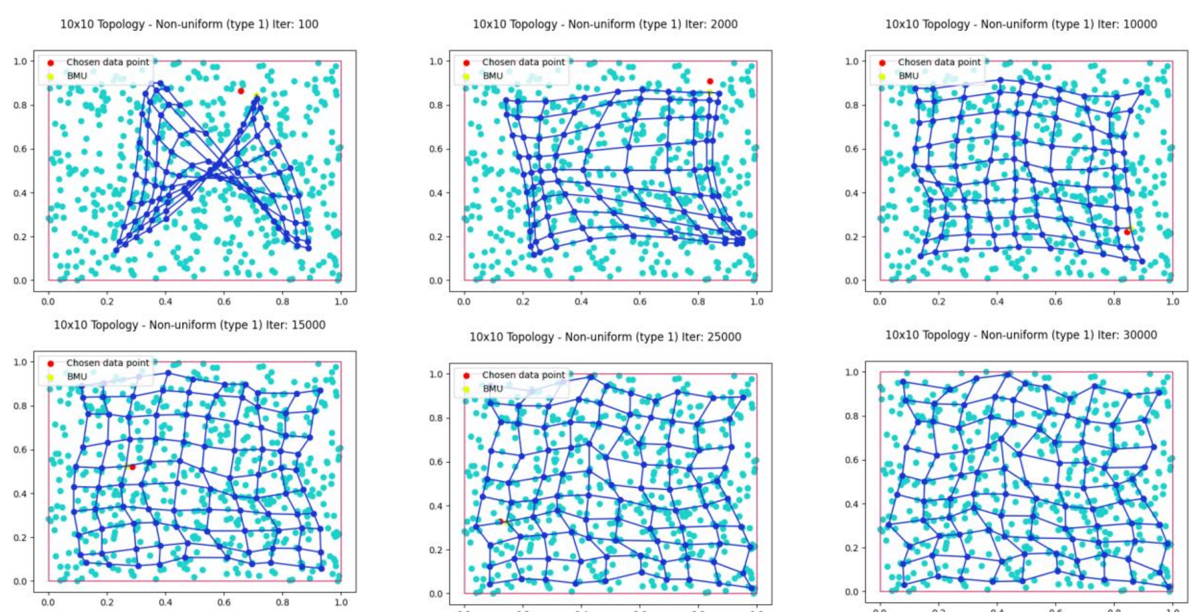
2. Discussions:

We did the same with two non-uniform distributions:

- [Dirichlet distribution] (*type 1*).

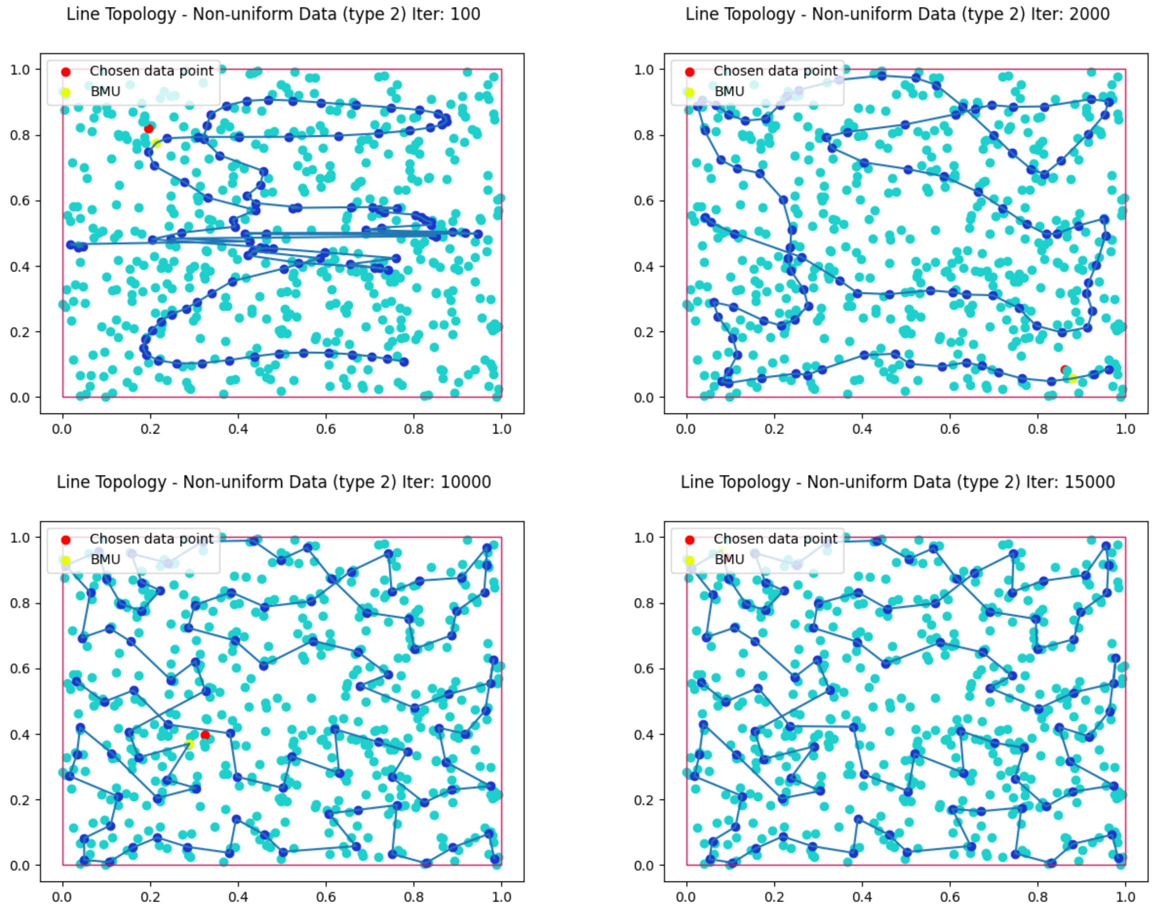


[Link to images]

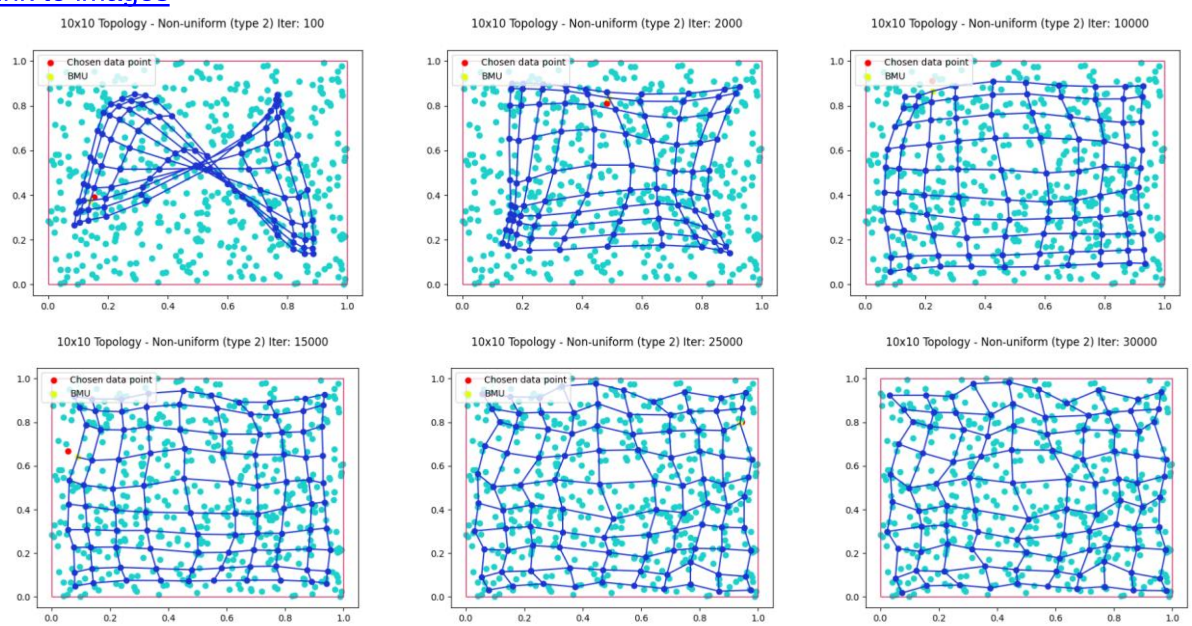


[Link to images]

- Depending on the distance from the square center, a point has a greater probability of being selected as a data point (*type 2*).



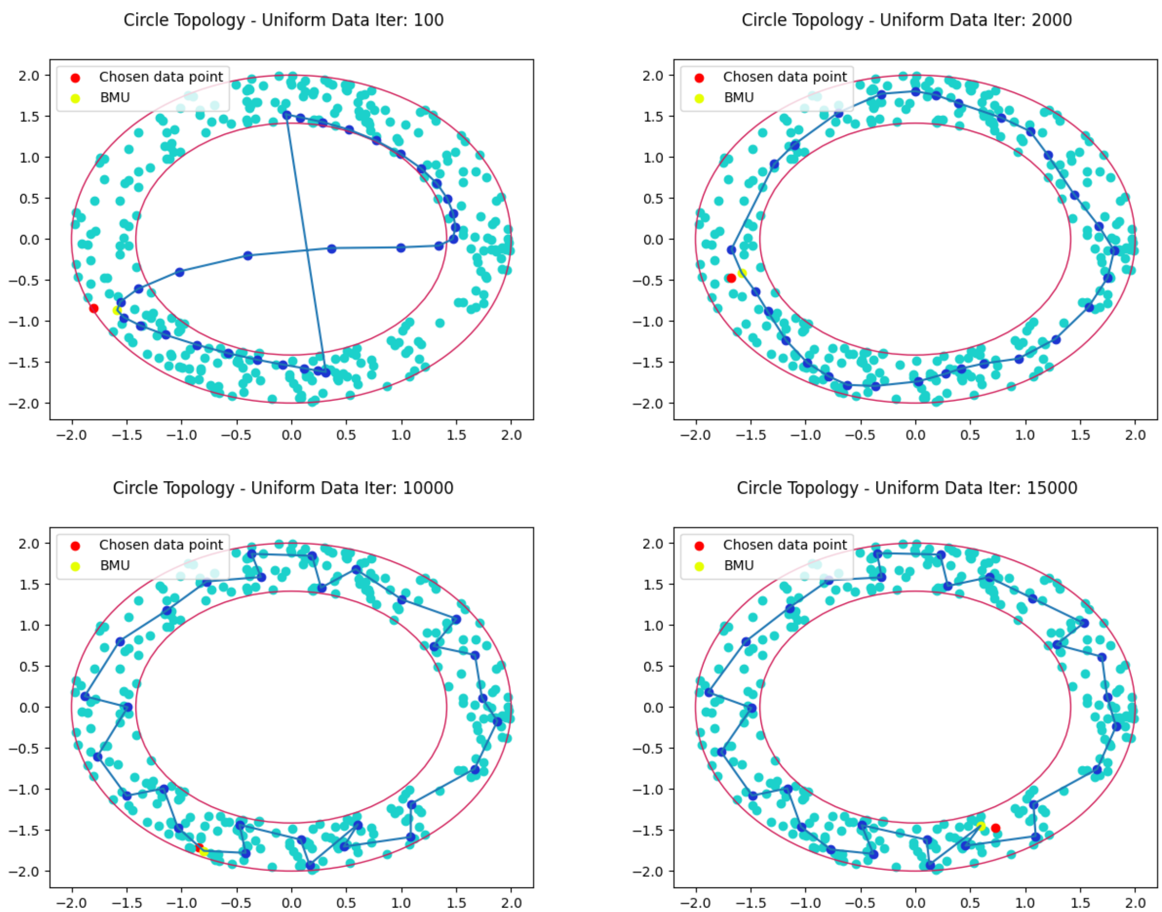
[Link to images]



[Link to images]

Following the Kohonen algorithm, the neural network is first unraveled, but unlike the uniform distribution, we see that they are concentrated in a certain place. After that, they gradually fill the entire square. As we wrote above, the twisting effect is more pronounced here.

3. **Discussions:**



[Link to images]()

According to Kohonen's algorithm (see image above), a ring-like neural network unravels before gradually filling the entire donut.

- **Code:**

[Link to github]()

```
import sys
import math
import random
import numpy as np
import matplotlib.pyplot as plt
from numpy.random.mtrand import dirichlet

NEURONS_SET = 100
NEURONS_SMALL_SET = 30
```

```python
DATA_SET = 300
LAST_ITERATION = 1
MAXSIZE = sys.maxsize
LOWER_BOUND = 0
UPPER_BOUND = 1
LOWER_RADIUS = 2
UPPER_RADIUS = 4
LIST_PRINT = [100, 2000, 10000, 15000, 25000]

NEURON_COLOR = '#1c34d1'
POINT_COLOR = '#1cd1cb'
CIRCLE_COLOR = '#d11c58'
CHOSEN_POINT_COLOR = '#ff0000'
BMU_COLOR = '#e6ff00'

random.seed(47)

# --------------------------------------------------------------- Classes --------
---------------------------------------
class Point:
    def __init__(self, x=0, y=0, chosen=0):
        """
        :param x: X Value
        :param y: Y Value
        :param chosen: Conscience
        """
        self.x = x
        self.y = y
        self.chosen = chosen


class Index:
    def __init__(self, x=0, y=0):
        """
        :param x: X Value
        :param y: Y Value
        """
        self.x = x
        self.y = y


class Node:
    def __init__(self, point=Point(), index=Index(), adjacent=[]):
        """
        :param point: Point (X, Y)
```

```
        :param index: The point index in the matrix topology.
        :param adjacent: The current point neighbours - matrix topology.
        """
        self.point = point
        self.index = index
        self.adjacent = adjacent

class KohonenAlgorithm:
    def __init__(self, set=NEURONS_SET, lowerBound=LOWER_BOUND,
upperBound=UPPER_BOUND, learning_rate=.5, neighborhood_distance=3, shape="Line",
border=0, circleShape=0):
        if shape == "Line" or shape == "Circle":
            self.neurons = generateLine(set, lowerBound, upperBound)
        else:
            self.neurons = generateMatrix(lowerBound=lowerBound,
upperBound=upperBound, set=set)

        self.shape = shape
        self.border = border
        self.circleShape = circleShape

        self.eps = learning_rate   # initial learning speed
        self.de = neighborhood_distance   # initial neighborhood distance
        self.ste = 0    # inital number of carried out steps

    def phi(self, i, k, d):             # proximity function for line and circle
        return np.exp(-(i-k)**2/(2*d**2)) # Gaussian

    def phi2(self, ix, iy, kx, ky, d):  # proximity function for matrix
        return np.exp(-((ix-kx)**2+(iy-ky)**2)/(d**2))  # Gaussian


    def train(self, data, title, rounds=150, points=100, uniform=0):
        """
        Function to activate the Kohonen algorithm
        :param data: the data to be trained on.
        :param title: Title of the task.
        :param rounds: number of rounds.
        :param points: number of points in each round.
        :param uniform: if 0 preforms uniform distribution sampling of the data.
if 1/2 than non-uniform.
        :return:
        """
        if self.shape == "Line" or self.shape == "Circle":
            self.lineCircleTrain(data, title, rounds, points, uniform)
```

```
        else:
            self.matrixTrain(data, title, rounds, points, uniform)


    def getRandomIndex(self, probabilities, lenData, uniform):
        """
        Function returns probabilities for random choice
        :param probabilities: list of probabilities.
        :param lenData: range of data.
        :param uniform: if 0 preforms uniform distribution sampling of the data.
if 1/2 than non-uniform.
        :return: random index
        """
        if uniform == 0:
            return np.random.choice(range(lenData))  # returns a random number.
        elif uniform == 1:
            return np.random.choice(range(lenData), p=probabilities)
        else:
            return np.random.choice(range(lenData), p=probabilities)

    def getProbabilities(self, data, uniform):
        """
        Function returns probabilities for random choice
        :param data: the data to be trained on.
        :param uniform: if 0 preforms uniform distribution sampling of the data.
if 1/2 than non-uniform.
        :return: list of probabilities
        """
        if uniform == 1:
            return getDirichletProbabilities(len(data))
        elif uniform == 2:
            return getDistanceProbabilities(data)
        return []

    # the Euclidean distance between two points in Euclidean space is the length
of a line segment between the two points.
    # https://en.wikipedia.org/wiki/Euclidean_distance
    def euclideanDist(self, data):
        """
        Function to find the minimum distance
        The Euclidean distance between two points in Euclidean space is the
length of a line segment between the two points.
        https://en.wikipedia.org/wiki/Euclidean_distance
        :param data: The data to be trained on.
        :return: The closest neuron to the given point.
```

```
        """
        minimum = MAXSIZE
        if self.shape == "Line" or self.shape == "Circle":
            index = 0
            lenNeurons = len(self.neurons)

            for i in range(lenNeurons):
                if self.neurons[i].chosen == 0:
                    distance = math.sqrt((data.x - self.neurons[i].x)**2 +
(data.y - self.neurons[i].y)**2)
                    if distance < minimum:
                        minimum = distance
                        index = i
        else:
            index = Index()
            for i in range(len(self.neurons[0])):
                for j in range(len(self.neurons[0])):
                    if self.neurons[i][j].point.chosen == 0:
                        distance = math.sqrt((data.x -
self.neurons[i][j].point.x) ** 2 + (data.y - self.neurons[i][j].point.y) ** 2)
                        if distance < minimum:
                            minimum = distance
                            index = Index(i, j)

        return index


    def drow(self, data, title, chosenIndex, bmuIndex, border=0, circleShape=0,
done=0):
        """
        Function to draw the points and neurons.
        :param points: Array of points.
        :param neurons: Array of neurons.
        :param title: Title of the task.
        :param chosenIndex: The chosen point.
        :param bmuIndex: The Best Matching Unit.
        :param border: Border = 0 -> draw rectangle border | Border = 1 -> draw
ring border (2 circles).
        :param circleShape: circleShape = 0 -> line topology | circleShape = 1 ->
circle topology.
        :param done: Done = 0 -> draw the board and clear | Done = 1 -> last
iteration, show the board.
        """
        if self.shape == "Line" or self.shape == "Circle":
```

```
            drowPaintNeurons(data, self.neurons, title, chosenIndex, bmuIndex,
border, circleShape, done)
        else:
            drowPaintMatrix(data, self.neurons, title, chosenIndex, bmuIndex,
done)
    # ---------------------------------------------------------- Line / Circle ---
---------------------------------------------
    def lineCircleTrain(self, data, title, rounds=150, points=100, uniform=0):
        """
        Function to activate the Kohonen algorithm
        :param data: the data to be trained on.
        :param title: Title of the task.
        :param rounds: number of rounds.
        :param points: number of points in each round.
        :param uniform: if 0 preforms uniform distribution sampling of the data.
if 1/2 than non-uniform.
        :return:
        """
        lenData = len(data)
        probability = self.getProbabilities(data, uniform)
        lenNeurons = len(self.neurons)

        for _ in range(rounds):       # rounds
            self.eps = self.eps*.98
            self.de = self.de*.95
            for _ in range(points):   # repeat for rep points
                self.ste = self.ste+1
                chosenIndex = self.getRandomIndex(probability, lenData, uniform)
                bmuIndex = self.euclideanDist(data[chosenIndex])

                for index in range(lenNeurons):
                    self.neurons[index].x += self.eps*self.phi(bmuIndex, index,
self.de)*(data[chosenIndex].x - self.neurons[index].x)
                    self.neurons[index].y += self.eps*self.phi(bmuIndex, index,
self.de)*(data[chosenIndex].y - self.neurons[index].y)

                if  self.ste in LIST_PRINT:
                    self.drow(data, title + " Iter: " + str(self.ste),
chosenIndex=chosenIndex, bmuIndex=bmuIndex, border=self.border,
circleShape=self.circleShape)

        self.drow(data, title + " Iter: " + str(self.ste), chosenIndex=-1,
bmuIndex=-1, border=self.border, circleShape=self.circleShape,
done=LAST_ITERATION)
```

```
    # ------------------------------------------------------------- Matrix -----
---------------------------------------------
    def matrixTrain(self, data, title, rounds=100, points=300, uniform=0):
        """
        Function to activate the Kohonen algorithm
        :param data: the data to be trained on.
        :param title: Title of the task.
        :param rounds: number of rounds.
        :param points: number of points in each round.
        :param uniform: if 0 preforms uniform distribution sampling of the data.
if 1/2 than non-uniform.
        :return:
        """
        lenData = len(data)
        probability = self.getProbabilities(data, uniform)
        rows = len(self.neurons)
        cols = len(self.neurons[0])

        for _ in range(rounds):   # rounds
            self.eps = self.eps*.97
            self.de = self.de*.98
            for _ in range(points):    # repeat for rep points
                self.ste = self.ste+1
                chosenIndex = self.getRandomIndex(probability, lenData, uniform)
                bmuIndex = self.euclideanDist(data[chosenIndex])
                ind_i=bmuIndex.x
                ind_j=bmuIndex.y

                for j in range(rows):
                    for i in range(cols):
                        self.neurons[i][j].point.x +=
self.eps*self.phi2(ind_i,ind_j,i,j,self.de)*(data[chosenIndex].x -
self.neurons[i][j].point.x)
                        self.neurons[i][j].point.y +=
self.eps*self.phi2(ind_i,ind_j,i,j,self.de)*(data[chosenIndex].y -
self.neurons[i][j].point.y)

                if  self.ste in LIST_PRINT:
                    self.drow(data, title + " Iter: " + str(self.ste),
chosenIndex=chosenIndex, bmuIndex=bmuIndex, border=self.border,
circleShape=self.circleShape)

        self.drow(data, title + " Iter: " + str(self.ste), chosenIndex=-1,
bmuIndex=Index(-1, -1), done=LAST_ITERATION)
```

```python
# ----------------------------------------------------------- Circle ----------
------------------------------------------------
def generateCircle(radius1, radius2, set=DATA_SET):
    """
    Function generate the neuron circle.
    :param set: The number of neurons.
    :param radius1: Radius 1.
    :param radius2: Radius 2.
    :return: neurons
    """
    points = []

    for _ in range(set):
        x = random.uniform(-radius2, radius2)
        points.append(Point(x, generateCircleRing(x, radius1, radius2)))

    return points

def generateCircleRing(x, radius1, radius2=0):
    """
    Function to create points of data.
    Radius2 = 0 -> create points within a circle | Radius2 != 0 -> create points
within a ring.
    :param x: Random X value
    :param radius1: Radius 1
    :param radius2: Radius 2
    :return: Random Y value within the circle / ring
    """
    if radius2 == 0:
        y_ = random.uniform(-radius1, radius1)
        while y_ ** 2 + x ** 2 > radius1 ** 2:
            y_ = random.uniform(-radius1, radius1)
        return y_
    else:
        y_ = random.uniform(-radius2, radius2)
        while (y_ ** 2 + x ** 2 > radius2 ** 2) or (y_ ** 2 + x ** 2 < radius1 **
2):
            y_ = random.uniform(-radius2, radius2)
        return y_


# ----------------------------------------------------------- Matrix ----------
-------------------------------------------------
```

```python
def createTwoDimensionalArray(neurons, isqrt):
    """
    :param neurons: isqrtXisqrt neurons.
    :param isqrt: The number of neurons in one row/column.
    :return: Neurons arranged in a isqrtXisqrt topology.
    """
    matrix = [[Node() for i in range(isqrt)] for j in range(isqrt)]

    "Corners"
    matrix[0][0] = Node(neurons[0][0], Index(0, 0), [Index(0, 1), Index(1, 0)])
    matrix[0][isqrt-1] = Node(neurons[0][isqrt-1], Index(0, 4), [Index(0, 3),
Index(1, 4)])
    matrix[isqrt-1][0] = Node(neurons[isqrt-1][0], Index(4, 0), [Index(3, 0),
Index(4, 1)])
    matrix[isqrt-1][isqrt-1] = Node(neurons[isqrt-1][isqrt-1], Index(4, 4),
[Index(3, 4), Index(4, 3)])

    for i in range(1, isqrt-1):
        "Edges"
        matrix[0][i] = Node(neurons[0][i], Index(0, i), [Index(0, i - 1),
Index(1, i), Index(0, i + 1)])
        matrix[i][0] = Node(neurons[i][0], Index(i, 0), [Index(i - 1, 0),
Index(i, 1), Index(i + 1, 0)])
        matrix[isqrt-1][i] = Node(neurons[isqrt-1][i], Index(4, i), [Index(4, i -
1), Index(3, i), Index(4, i + 1)])
        matrix[i][isqrt-1] = Node(neurons[i][isqrt-1], Index(i, 4), [Index(i - 1,
4), Index(i, 3), Index(i + 1, 4)])

        "General Case"
        for j in range(1, isqrt-1):
            matrix[i][j] = Node(neurons[i][j], Index(i, j), [Index(i, j-1),
Index(i-1, j), Index(i, j+1), Index(i+1, j)])

    return matrix

def generateMatrix(lowerBound , upperBound, set=NEURONS_SET):
    """
    Function generate the neuron matrix.
    :param set: The number of neurons.
    :param lowerBound: The lower bound for a lower parameter in random.uniform
function.
    :param upperBound: The upper bound for a high parameter in random.uniform
function.
    :return: neurons
    """
```

```
    isqrt = math.isqrt(set)
    neurons = []
    neurons = [[Point() for i in range(isqrt)] for j in range(isqrt)]
    for i in range(isqrt):
        for j in range(isqrt):
            neurons[i][j] = Point(random.uniform(lowerBound, upperBound),
random.uniform(lowerBound, upperBound))
    matrix = createTwoDimensionalArray(neurons, isqrt)

    return matrix

# --------------------------------------------------------- Drow Function -------
-------------------------------------------------

def drowPaintNeurons(points, neurons, title, chosenIndex=-1, bmuIndex=-1,
border=0, circleShape=0, done=0):
    """
    Function to draw the points and neurons.
    :param points: Array of points.
    :param neurons: Array of neurons.
    :param title: Title of the task.
    :param chosenIndex: The chosen point.
    :param bmuIndex: The Best Matching Unit.
    :param border: Border = 0 -> draw rectangle border | Border = 1 -> draw ring
border (2 circles).
    :param circleShape: circleShape = 0 -> line topology | circleShape = 1 ->
circle topology.
    :param done: Done = 0 -> draw the board and clear | Done = 1 -> last
iteration, show the board.
    :return: None
    """
    neurons_x = []
    neurons_y = []

    for i in range(len(points)):
        if done == 0 and chosenIndex == i:
            plt.scatter(points[i].x, points[i].y, color=CHOSEN_POINT_COLOR,
label='Chosen data point')
        else:
            plt.scatter(points[i].x, points[i].y, color=POINT_COLOR)

    for i in range(len(neurons)):
        neurons_x.append(neurons[i].x)
        neurons_y.append(neurons[i].y)
        if done == 0 and bmuIndex == i:
            plt.scatter(neurons[i].x, neurons[i].y, color=BMU_COLOR, label='BMU')
```

```
        else:
            plt.scatter(neurons[i].x, neurons[i].y, color=NEURON_COLOR)
    if circleShape == 1:
        neurons_x.append(neurons_x[0]), neurons_y.append(neurons_y[0])

    if border == 0:
        # {(x,y) |  0 <= x <= 1, 0<=y<=1}
        rectangle = plt.Rectangle((0,0), 1, 1,  color=CIRCLE_COLOR, fill=False)
        ax = plt.gca()
        ax.add_patch(rectangle)
    elif border == 1:
        circle1 = plt.Circle((0, 0), math.sqrt(LOWER_RADIUS), color=CIRCLE_COLOR,
fill=False)
        circle2 = plt.Circle((0, 0), math.sqrt(UPPER_RADIUS), color=CIRCLE_COLOR,
fill=False)
        ax = plt.gca()
        ax.add_patch(circle1)
        ax.add_patch(circle2)

    plt.suptitle(title)
    plt.plot(neurons_x, neurons_y)

    if done == LAST_ITERATION:
        plt.show()
    else:
        plt.legend(loc="upper left")
        plt.draw()
        plt.pause(0.01)
        plt.clf()


def drowPaintMatrix(points, matrix, title, chosenIndex=-1, bmuIndex=Index(-1, -
1), done=0):
    """
    Function to draw the points and matrix.
    :param points: Array of points.
    :param matrix: Matrix of neurons.
    :param title: Title of the task.
    :param chosenIndex: The chosen point.
    :param bmuIndex: The Best Matching Unit.
    :param done: Done = 0 -> draw the matrix and clear | Done = 1 -> last
iteration, show the matrix.
    :return: None
    """
    neurons_x = [[] for _ in range(2 * len(matrix[0]))]
```

```
    neurons_y = [[] for _ in range(2 * len(matrix[0]))]

    for i in range(len(points)):
        if done == 0 and chosenIndex == i:
            plt.scatter(points[i].x, points[i].y, color=CHOSEN_POINT_COLOR,
label='Chosen data point')
        else:
            plt.scatter(points[i].x, points[i].y, color=POINT_COLOR)

    index = 0
    for i in range(len(matrix[0])):
        for j in range(len(matrix[0])):
            if done == 0 and bmuIndex.x == i and bmuIndex.y == j:
                plt.scatter(matrix[i][j].point.x, matrix[i][j].point.y,
color=BMU_COLOR, label='BMU')
            else:
                plt.scatter(matrix[i][j].point.x, matrix[i][j].point.y,
color=NEURON_COLOR)
            neurons_x[index].append(matrix[i][j].point.x)
            neurons_y[index].append(matrix[i][j].point.y)
        index += 1

    for i in range(len(matrix[0])):
        for j in range(len(matrix[0])):
            neurons_x[index].append(matrix[j][i].point.x)
            neurons_y[index].append(matrix[j][i].point.y)
        index += 1

    plt.suptitle(title)

    for i in range(len(neurons_x)):
        plt.plot(neurons_x[i], neurons_y[i], NEURON_COLOR)

    # {(x,y) |  0 <= x <= 1, 0<=y<=1}
    rectangle = plt.Rectangle((0,0), 1, 1,  color=CIRCLE_COLOR, fill=False)
    ax = plt.gca()
    ax.add_patch(rectangle)

    if done == LAST_ITERATION:
        plt.show()
    else:
        plt.legend(loc="upper left")
        plt.draw()
        plt.pause(0.01)
        plt.clf()
```

```
# ---------------------------------------------------------- Help Function -------
-------------------------------------------------
# https://en.wikipedia.org/wiki/Dirichlet_distribution
def getDirichletProbabilities(length):
    """
    Function generate the probability array by the dirichlet function.
    :param length: lenght of array.
    :return: probability array
    """
    probability = dirichlet([1] * length)  # uses the dirichlet function to
distribute probabilities.
    return probability

def getDistanceProbabilities(points):
    """
    Function generate the probability of a point being chosen as a data point is
proportional to the distance from the center of the disk.
    :param points: Array of points.
    :return: probability array
    """
    probability = []
    xCenter = (UPPER_BOUND - LOWER_BOUND)/2
    yCenter = xCenter

    for point in points:
        probability.append(math.dist([point.x, point.y], [xCenter, yCenter]))

    probability = np.asarray(probability)
    probability = (probability - min(probability)) / sum(probability -
min(probability))
    return probability

def generateLine(set, lowerBound , upperBound):
    """
    Function generate the neuron line.
    :param set: The number of neurons.
    :param lowerBound: The lower bound for a lower parameter in random.uniform
function.
    :param upperBound: The upper bound for a high parameter in random.uniform
function.
    :return: neurons
    """
    neurons = []
    y = (upperBound - lowerBound)/2
```

```
    for _ in range(set):
        neurons.append(Point(random.uniform(lowerBound, upperBound), y))
    return neurons

def generateSquare(set, lowerBound , upperBound):
    """
    Function generate the data points.
    :param set: The number of points.
    :param lowerBound: The lower bound for a lower parameter in random.uniform
function.
    :param upperBound: The upper bound for a high parameter in random.uniform
function.
    :return: The data points
    """
    squareData = []

    # the data set is {(x,y) |  0 <= x <= 1, 0<=y<=1}
    for i in range(set):
        squareData.append(Point(random.uniform(lowerBound, upperBound),
random.uniform(lowerBound, upperBound)))
    return squareData

def main():
    lowerBound = LOWER_BOUND
    upperBound = UPPER_BOUND
    # Part A.1
    # -------------------------------------------------------- Uniform Dat ------
---------------------------------------------------
    # ------------------------------------------------------- Line Topology -----
---------------------------------------------------
    # create points
    points = generateSquare(set=500, lowerBound=lowerBound,
upperBound=upperBound)

    # 20000 iterations
    somSquare = KohonenAlgorithm(set=NEURONS_SET, lowerBound=lowerBound,
upperBound=upperBound, neighborhood_distance=10, shape="Line", border=0,
circleShape=0)
    somSquare.train(points, "Line Topology - Uniform Data", rounds=150,
points=100, uniform=0)

    # ------------------------------------------------------- 10x10 Topology ----
---------------------------------------------------
    # 30000 iterations
```

```
    somSquare = KohonenAlgorithm(set=NEURONS_SET, lowerBound=lowerBound,
upperBound=upperBound, shape="Matrix")
    somSquare.train(points, "10x10 Topology - Uniform Data", rounds=100,
points=300, uniform=0)

    # Part A.2
    # ------------------------------------------------------ Non-uniform Dat ----
---------------------------------------------------
    # ------------------------------------------------------- Line Topology -----
---------------------------------------------------
    points = generateSquare(set=500, lowerBound=lowerBound,
upperBound=upperBound)

    # 20000 iterations
    somSquare = KohonenAlgorithm(set=NEURONS_SET, lowerBound=lowerBound,
upperBound=upperBound, shape="Line", border=0, circleShape=0)
    somSquare.train(points, "Line Topology - Non-uniform Data (type 1)",
rounds=150, points=100, uniform=1)

    # 20000 iterations
    somSquare = KohonenAlgorithm(set=NEURONS_SET, lowerBound=lowerBound,
upperBound=upperBound, shape="Line", border=0, circleShape=0)
    somSquare.train(points, "Line Topology - Non-uniform Data (type 2)",
rounds=150, points=100, uniform=2)

    # ------------------------------------------------------- 10x10 Topology ----
---------------------------------------------------
    # 30000 iterations
    somSquare = KohonenAlgorithm(set=NEURONS_SET, lowerBound=lowerBound,
upperBound=upperBound, shape="Matrix")
    somSquare.train(points, "10x10 Topology - Non-uniform (type 1)", rounds=100,
points=300, uniform=1)

    # 30000 iterations
    somSquare = KohonenAlgorithm(set=NEURONS_SET, lowerBound=lowerBound,
upperBound=upperBound, shape="Matrix")
    somSquare.train(points, "10x10 Topology - Non-uniform (type 2)", rounds=100,
points=300, uniform=2)

    # Part A.3
    # ------------------------------------------------------- Circle Topology ---
---------------------------------------------------
    points = generateCircle(math.sqrt(LOWER_RADIUS), math.sqrt(UPPER_RADIUS),
set=DATA_SET)
```

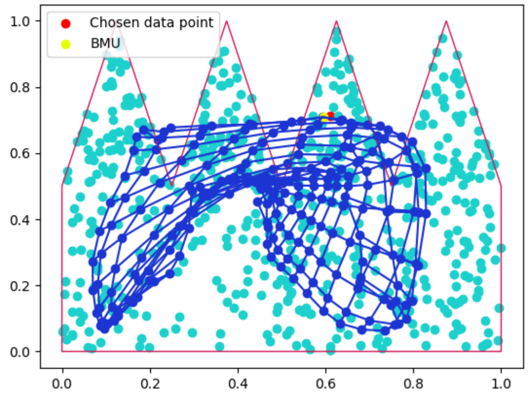
```
    # 20000 iterations
    somSquare = KohonenAlgorithm(set=NEURONS_SMALL_SET, lowerBound=lowerBound,
upperBound=upperBound, shape="Circle", border=1, circleShape=1)
    somSquare.train(points, "Circle Topology - Uniform Data", rounds=150,
points=100, uniform=0)


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

Part B:

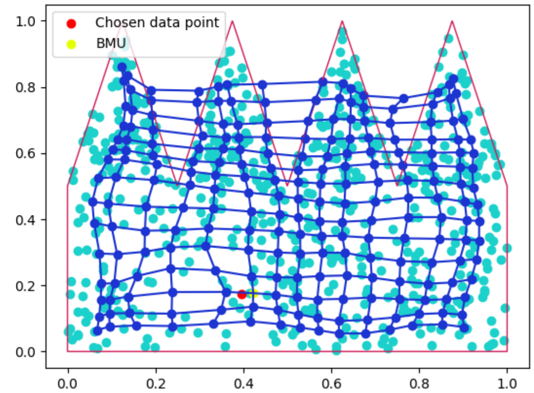
Our code was used for this part.

1. **Discussions:**

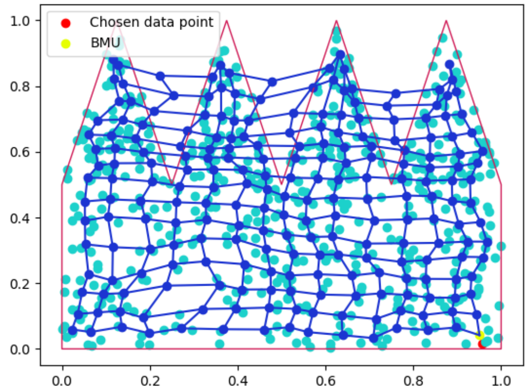
15x15 Topology - Uniform Data Iter: 100



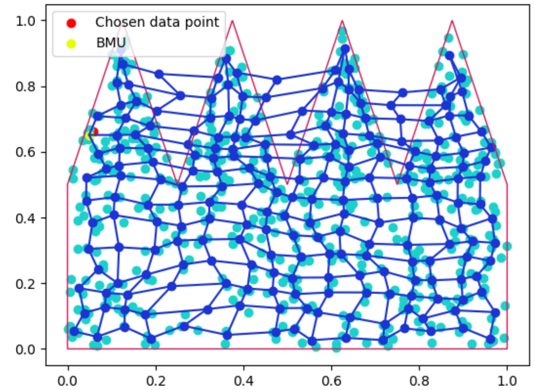
15x15 Topology - Uniform Data Iter: 10000



15x15 Topology - Uniform Data Iter: 20000



15x15 Topology - Uniform Data Iter: 30000



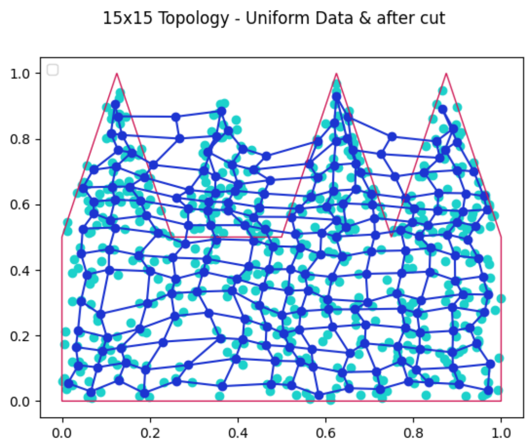
[Link to images]()

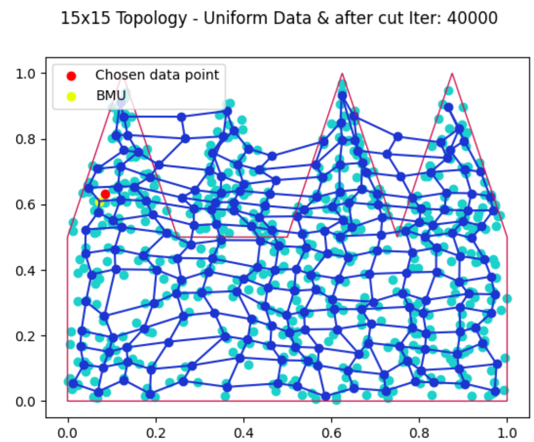
According to the figure above, the neurons haven't entirely been removed from the place between the fingers, but we're getting close.

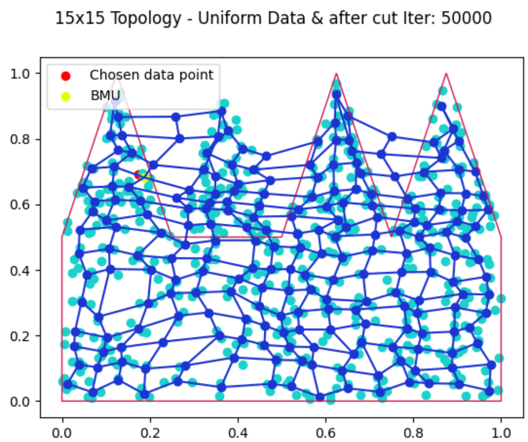
2. **Discussions:**

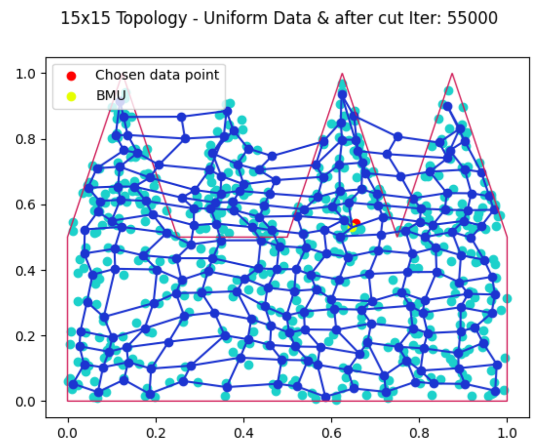
We have performed two experiments here:

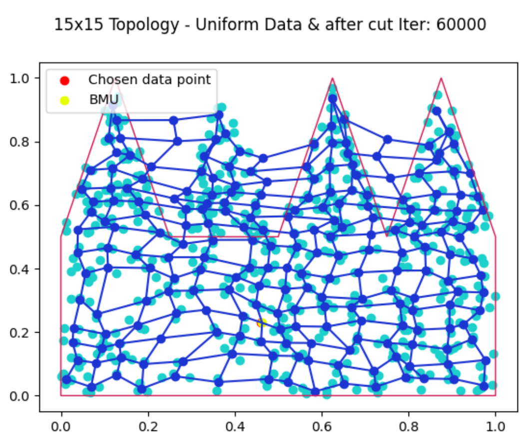
- Continuous iterations without changing the parameters associated with the number of neighbors and the learning rate.

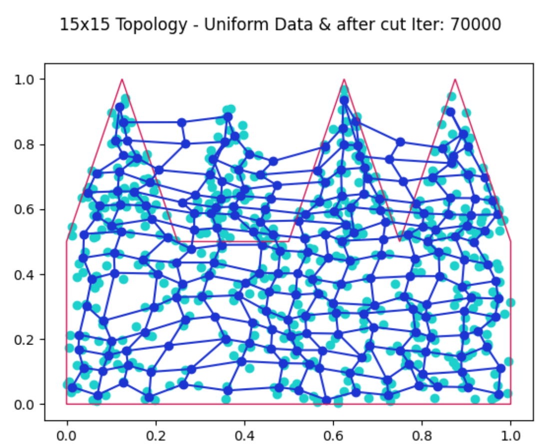








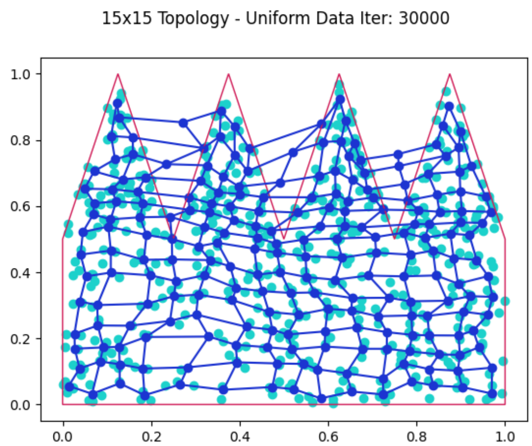


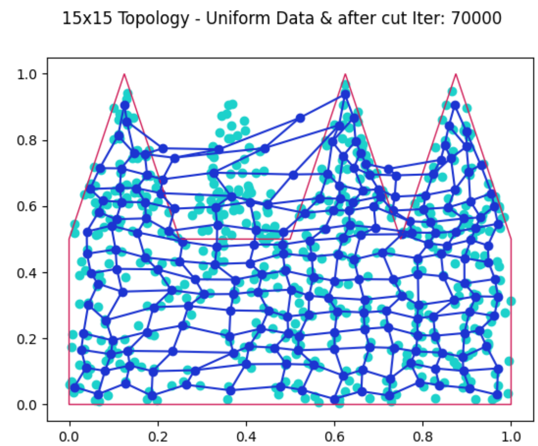


[Link to images]

It was not possible to remove the neurons from the area of the severed finger, as we can see above.

- Iterations continued, but parameters associated with the number of neighbors and learning rate were reset to the original values.





[Link to images](#)

We were able to practically remove neurons from the area of the severed finger in this case.

- **Code:**

[Link to github](#)

```
import sys
import math
import random
import numpy as np
import matplotlib.pyplot as plt

NEURONS_SET = 225
DATA_SET = 600
LAST_ITERATION = 1
DEFAULT_ITERATIONS = 10
MAXSIZE = sys.maxsize
LOWER_BOUND = 0
UPPER_BOUND = 1
LIST_PRINT = [100, 10000, 20000, 30000, 40000, 50000, 55000, 60000]

NEURON_COLOR = '#1c34d1'
POINT_COLOR = '#1cd1cb'
CIRCLE_COLOR = '#d11c58'
CHOSEN_POINT_COLOR = '#ff0000'
BMU_COLOR = '#e6ff00'

random.seed(47)


# ------------------------------------------------------------- Classes --------
----------------------------------------
class Point:
```

```python
    def __init__(self, x=0, y=0, chosen=0):
        """
        :param x: X Value
        :param y: Y Value
        :param chosen: Conscience
        """
        self.x = x
        self.y = y
        self.chosen = chosen


class Index:
    def __init__(self, x=0, y=0):
        """
        :param x: X Value
        :param y: Y Value
        """
        self.x = x
        self.y = y


class Node:
    def __init__(self, point=Point(), index=Index(), adjacent=[]):
        """
        :param point: Point (X, Y)
        :param index: The point index in the matrix topology.
        :param adjacent: The current point neighbours - matrix topology.
        """
        self.point = point
        self.index = index
        self.adjacent = adjacent

class KohonenAlgorithm:
    def __init__(self, set=NEURONS_SET, lowerBound=LOWER_BOUND,
upperBound=UPPER_BOUND, learning_rate=.5, neighborhood_distance=3):
        self.neurons = generateMatrix(lowerBound=lowerBound,
upperBound=upperBound, set=set)

        self.eps = learning_rate   # initial learning speed
        self.de = neighborhood_distance   # initial neighborhood distance
        self.ste = 0    # inital number of carried out steps

    def phi2(self, ix, iy, kx, ky, d):  # proximity function for matrix
        return np.exp(-((ix-kx)**2+(iy-ky)**2)/(d**2))  # Gaussian
```

```
    def train(self, data, title, rounds=100, points=300, cutOffFinger=0):
        """
        Function to activate the Kohonen algorithm
        :param data: the data to be trained on.
        :param title: Title of the task.
        :param rounds: number of rounds.
        :param points: number of points in each round.
        :param cutOffFinger: cutOffFinger = 0 -> draw the hand | cutOffFinger = 1
-> draw the hand with cut off a finger.
        :return:
        """
        lenData = len(data)
        rows = len(self.neurons)
        cols = len(self.neurons[0])
        if cutOffFinger == 1:
            self.eps = .5
            self.de = 3

        self.drow(data, title, chosenIndex=-1, bmuIndex=Index(-1, -1),
cutOffFinger=cutOffFinger)
        for _ in range(rounds):   # rounds
            self.eps = self.eps*.97
            self.de = self.de*.98
            for _ in range(points):    # repeat for rep points
                self.ste = self.ste+1

                if cutOffFinger == 1:
                    chosenIndex = randomPointInside(lenData, data)
                else:
                    chosenIndex = np.random.choice(range(lenData))
                bmuIndex = self.euclideanDist(data[chosenIndex])

                ind_i=bmuIndex.x
                ind_j=bmuIndex.y

                for j in range(rows):
                    for i in range(cols):
                        self.neurons[i][j].point.x +=
self.eps*self.phi2(ind_i,ind_j,i,j,self.de)*(data[chosenIndex].x -
self.neurons[i][j].point.x)
                        self.neurons[i][j].point.y +=
self.eps*self.phi2(ind_i,ind_j,i,j,self.de)*(data[chosenIndex].y -
self.neurons[i][j].point.y)

                if  self.ste in LIST_PRINT:
```

```
                    self.drow(data, title + " Iter: " + str(self.ste),
chosenIndex=chosenIndex, bmuIndex=bmuIndex, cutOffFinger=cutOffFinger)

        self.drow(data, title + " Iter: " + str(self.ste), chosenIndex=-1,
bmuIndex=Index(-1, -1), cutOffFinger=cutOffFinger, done=LAST_ITERATION)


    # the Euclidean distance between two points in Euclidean space is the length
of a line segment between the two points.
    # https://en.wikipedia.org/wiki/Euclidean_distance
    def euclideanDist(self, data):
        """
        Function to find the minimum distance
        The Euclidean distance between two points in Euclidean space is the
length of a line segment between the two points.
        https://en.wikipedia.org/wiki/Euclidean_distance
        :param data: The data to be trained on.
        :return: The closest neuron to the given point.
        """
        minimum = MAXSIZE
        index = Index()
        for i in range(len(self.neurons[0])):
            for j in range(len(self.neurons[0])):
                if self.neurons[i][j].point.chosen == 0:
                    distance = math.sqrt((data.x - self.neurons[i][j].point.x) **
2 + (data.y - self.neurons[i][j].point.y) ** 2)
                    if distance < minimum:
                        minimum = distance
                        index = Index(i, j)
        return index


    def drow(self, points, title, chosenIndex=-1, bmuIndex=Index(-1, -1),
cutOffFinger=0, done=0):
        """
        Function to draw the points and matrix.
        :param points: Array of points.
        :param title: Title of the task.
        :param chosenIndex: The chosen point.
        :param bmuIndex: The Best Matching Unit.
        :param cutOffFinger: cutOffFinger = 0 -> draw the hand | cutOffFinger = 1
-> draw the hand with cut off a finger.
        :param done: Done = 0 -> draw the matrix and clear | Done = 1 -> last
iteration, show the matrix.
        :return: None
```

```
        """
        neurons_x = [[] for _ in range(2 * len(self.neurons[0]))]
        neurons_y = [[] for _ in range(2 * len(self.neurons[0]))]

        for i in range(len(points)):
            if done == 0 and chosenIndex == i:
                plt.scatter(points[i].x, points[i].y, color=CHOSEN_POINT_COLOR,
label='Chosen data point')
            else:
                plt.scatter(points[i].x, points[i].y, color=POINT_COLOR)

        index = 0
        for i in range(len(self.neurons[0])):
            for j in range(len(self.neurons[0])):
                if done == 0 and bmuIndex.x == i and bmuIndex.y == j:
                    plt.scatter(self.neurons[i][j].point.x,
self.neurons[i][j].point.y, color=BMU_COLOR, label='BMU')
                else:
                    plt.scatter(self.neurons[i][j].point.x,
self.neurons[i][j].point.y, color=NEURON_COLOR)
                neurons_x[index].append(self.neurons[i][j].point.x)
                neurons_y[index].append(self.neurons[i][j].point.y)
            index += 1

        for i in range(len(self.neurons[0])):
            for j in range(len(self.neurons[0])):
                neurons_x[index].append(self.neurons[j][i].point.x)
                neurons_y[index].append(self.neurons[j][i].point.y)
            index += 1

        if cutOffFinger == 0:
            polygon1 = plt.Polygon([(0,0), (0,0.5), (0.125,1), (0.25,0.5),
(0.375,1), (0.5,0.5), (0.625,1), (0.75,0.5), (0.875,1), (1,0.5), (1,0),],
color=CIRCLE_COLOR, fill=False)
            ax = plt.gca()
            ax.add_patch(polygon1)
        else:
            polygon1 = plt.Polygon([(0,0), (0,0.5), (0.125,1), (0.25,0.5),
(0.5,0.5), (0.625,1), (0.75,0.5), (0.875,1), (1,0.5), (1,0),],
color=CIRCLE_COLOR, fill=False)
            ax = plt.gca()
            ax.add_patch(polygon1)

        plt.suptitle(title)
```

```
        for i in range(len(neurons_x)):
            plt.plot(neurons_x[i], neurons_y[i], NEURON_COLOR)

        if done == LAST_ITERATION:
            plt.show()
        else:
            plt.legend(loc="upper left")
            plt.show()
            # plt.draw()
            # plt.pause(0.01)
            # plt.clf()


# -------------------------------------------------------------- Matrix ---------
----------------------------------------
def createTwoDimensionalArray(neurons, isqrt):
    """
    :param neurons: isqrtXisqrt neurons.
    :param isqrt: The number of neurons in one row/column.
    :return: Neurons arranged in a isqrtXisqrt topology.
    """
    matrix = [[Node() for i in range(isqrt)] for j in range(isqrt)]

    "Corners"
    matrix[0][0] = Node(neurons[0][0], Index(0, 0), [Index(0, 1), Index(1, 0)])
    matrix[0][isqrt-1] = Node(neurons[0][isqrt-1], Index(0, 4), [Index(0, 3),
Index(1, 4)])
    matrix[isqrt-1][0] = Node(neurons[isqrt-1][0], Index(4, 0), [Index(3, 0),
Index(4, 1)])
    matrix[isqrt-1][isqrt-1] = Node(neurons[isqrt-1][isqrt-1], Index(4, 4),
[Index(3, 4), Index(4, 3)])

    for i in range(1, isqrt-1):
        "Edges"
        matrix[0][i] = Node(neurons[0][i], Index(0, i), [Index(0, i - 1),
Index(1, i), Index(0, i + 1)])
        matrix[i][0] = Node(neurons[i][0], Index(i, 0), [Index(i - 1, 0),
Index(i, 1), Index(i + 1, 0)])
        matrix[isqrt-1][i] = Node(neurons[isqrt-1][i], Index(4, i), [Index(4, i -
1), Index(3, i), Index(4, i + 1)])
        matrix[i][isqrt-1] = Node(neurons[i][isqrt-1], Index(i, 4), [Index(i - 1,
4), Index(i, 3), Index(i + 1, 4)])

        "General Case"
        for j in range(1, isqrt-1):
```

```
            matrix[i][j] = Node(neurons[i][j], Index(i, j), [Index(i, j-1),
Index(i-1, j), Index(i, j+1), Index(i+1, j)])

    return matrix

def generateMatrix(lowerBound , upperBound, set=NEURONS_SET):
    """
    Function generate the neuron matrix.
    :param set: The number of neurons.
    :param lowerBound: The lower bound for a lower parameter in random.uniform
function.
    :param upperBound: The upper bound for a high parameter in random.uniform
function.
    :return: neurons
    """
    isqrt = math.isqrt(set)
    neurons = []
    neurons = [[Point() for i in range(isqrt)] for j in range(isqrt)]
    for i in range(isqrt):
        for j in range(isqrt):
            neurons[i][j] = Point(random.uniform(lowerBound, upperBound),
random.uniform(lowerBound, upperBound))
    matrix = createTwoDimensionalArray(neurons, isqrt)

    return matrix

# ---------------------------------------------------------- Help Function -------
--------------------------------------------------
def area(x1, y1, x2, y2, x3, y3):
    return abs((x1 * (y2 - y3) + x2 * (y3 - y1)
                + x3 * (y1 - y2)) / 2.0)


def isInside(x1, y1, x2, y2, x3, y3, x, y):
    # Calculate area of triangle ABC
    A = area(x1, y1, x2, y2, x3, y3)

    # Calculate area of triangle PBC
    A1 = area(x, y, x2, y2, x3, y3)

    # Calculate area of triangle PAC
    A2 = area(x1, y1, x, y, x3, y3)

    # Calculate area of triangle PAB
    A3 = area(x1, y1, x2, y2, x, y)
```

```
    # Check if sum of A1, A2 and A3
    # is same as A
    if(A == A1 + A2 + A3):
        return True
    else:
        return False


def randomPointInside(lenPoints, points):
    while True:
        chosenIndex = np.random.choice(range(lenPoints))
        x, y = points[chosenIndex].x, points[chosenIndex].y
        inside = isInside(0.25,0.5, 0.375,1, 0.5,0.5, x, y)
        if inside==False:
            return chosenIndex


def pointOnTriangle(pt1, pt2, pt3):
    """
    Random point on the triangle with vertices pt1, pt2 and pt3.
    """
    x, y = random.random(), random.random()
    q = abs(x - y)
    s, t, u = q, 0.5 * (x + y - q), 1 - 0.5 * (q + x + y)
    return (
        s * pt1[0] + t * pt2[0] + u * pt3[0],
        s * pt1[1] + t * pt2[1] + u * pt3[1],
    )


def generateHand(set, lowerBound, upperBound):
    """
    Function generate the data points.
    :param set: The number of points.
    :param lowerBound: The lower bound for a lower parameter in random.uniform
function.
    :param upperBound: The upper bound for a high parameter in random.uniform
function.
    :return: The data points
    """
    handData = []
    palm = int(set/2)
    fingers = int(set/8)
```

```python
    # the data set is {(x,y) |  0 <= x <= 1, 0<=y<=1}
    for _ in range(palm):
        handData.append(Point(random.uniform(lowerBound, upperBound),
random.uniform(lowerBound, upperBound/2)))

    # first finger
    pt1 = (lowerBound, upperBound/2)
    pt2 = (upperBound/8, upperBound)
    pt3 = (upperBound/4, upperBound/2)
    for _ in range(fingers):
        point = pointOnTriangle(pt1, pt2, pt3)
        handData.append(Point(point[0], point[1]))

    # second finger
    pt1 = (upperBound/4, upperBound/2)
    pt2 = (upperBound/4+upperBound/8, upperBound)
    pt3 = (upperBound/2, upperBound/2)
    for _ in range(fingers):
        point = pointOnTriangle(pt1, pt2, pt3)
        handData.append(Point(point[0], point[1]))

    # third finger
    pt1 = (upperBound/2, upperBound/2)
    pt2 = (upperBound/2+upperBound/8, upperBound)
    pt3 = (upperBound-upperBound/4, upperBound/2)
    for _ in range(fingers):
        point = pointOnTriangle(pt1, pt2, pt3)
        handData.append(Point(point[0], point[1]))

    # fourth finger
    pt1 = (upperBound-upperBound/4, upperBound/2)
    pt2 = (upperBound-upperBound/8, upperBound)
    pt3 = (upperBound, upperBound/2)
    for _ in range(fingers):
        point = pointOnTriangle(pt1, pt2, pt3)
        handData.append(Point(point[0], point[1]))

    return handData


def main():
    lowerBound = LOWER_BOUND
    upperBound = UPPER_BOUND
    # Part B.1
```

```
    # -------------------------------------------------------- 15x15 Topology ----
---------------------------------------------------
    points = generateHand(set=DATA_SET, lowerBound=lowerBound,
upperBound=upperBound)

    # 30000 iterations
    somSquare = KohonenAlgorithm(set=NEURONS_SET, lowerBound=lowerBound,
upperBound=upperBound)
    somSquare.train(points, cutOffFinger=0, rounds=100, points=300, title="15x15
Topology - Uniform Data")

    # Part B.2
    # -------------------------------------------------------- 15x15 Topology ----
---------------------------------------------------
    # 30000 iterations
    somSquare.train(points, cutOffFinger=1, rounds=100, points=400, title="15x15
Topology - Uniform Data & after cut")



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