Neuro Computation Ex2

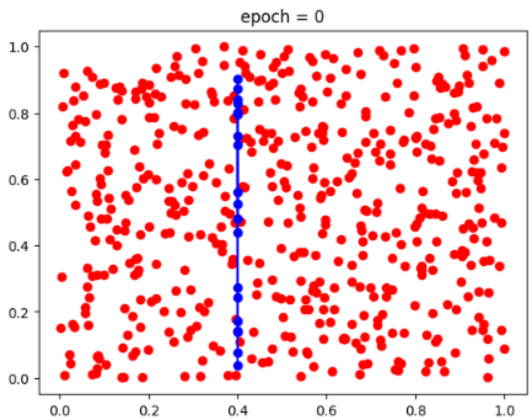
Submitters Names:

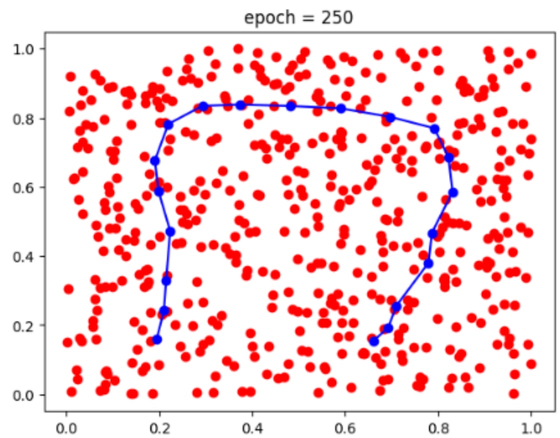
Yan Naigebaver, Eilon Barashi, Orel Zamler

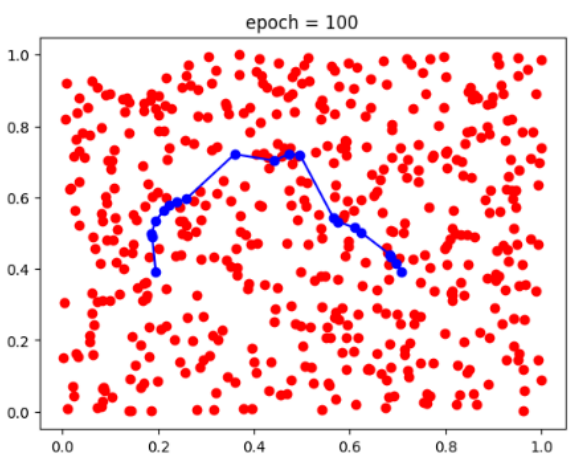
Contents

Question A

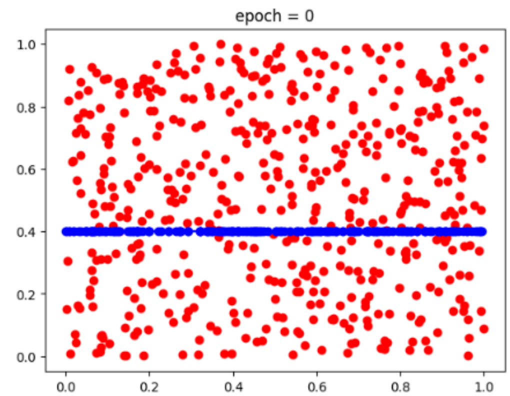
We drew a vertical line of 20 neurons, and after 250 epochs over dataset of 5 00 uniformly distributed 2d points between 0 and 1 in the first and second di mensions, the result was a n shaped structure of the neurons.

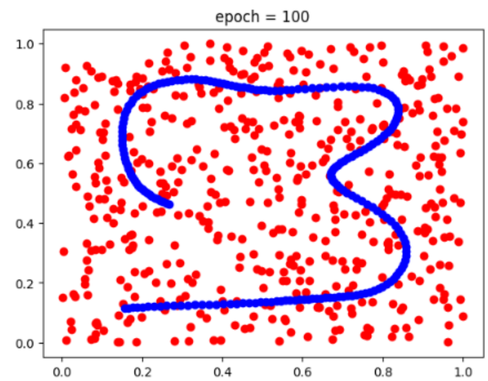


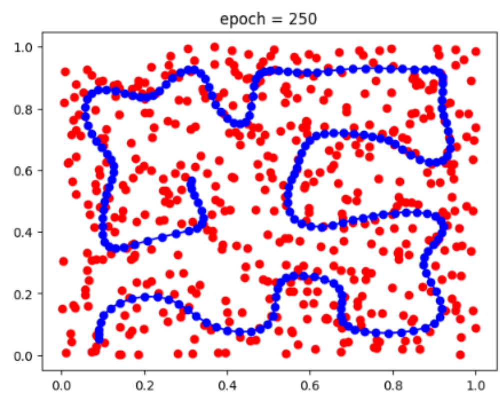




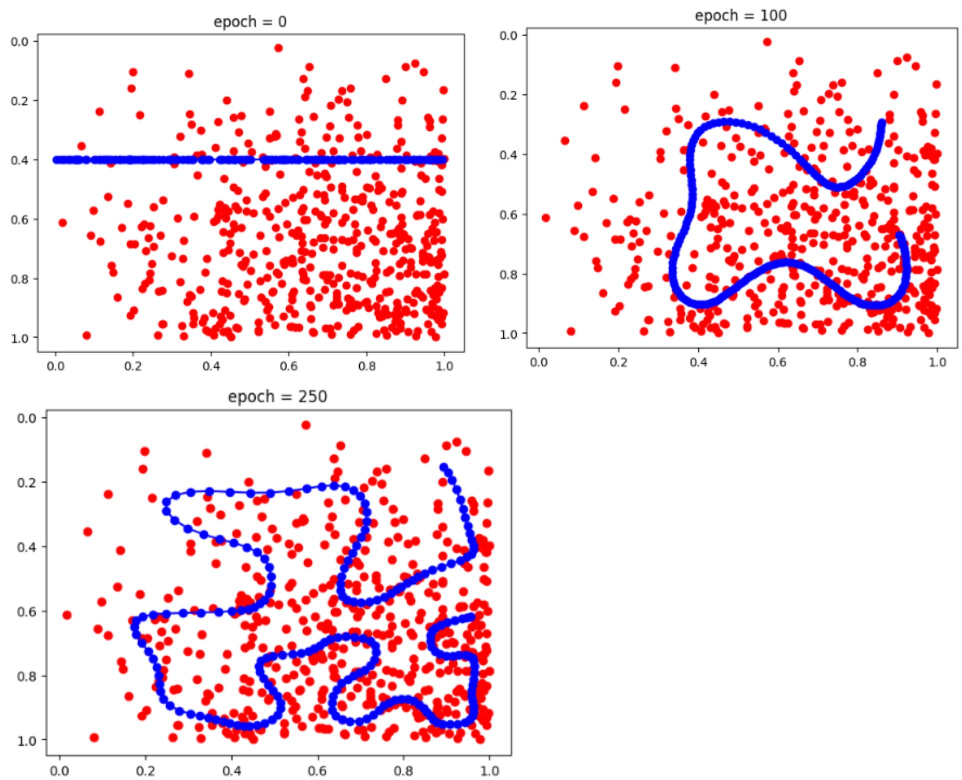
For 200 neurons, we drew a horizontal line with the neurons and we examined that the data is being distributed more accurately and takes more space than the former test.



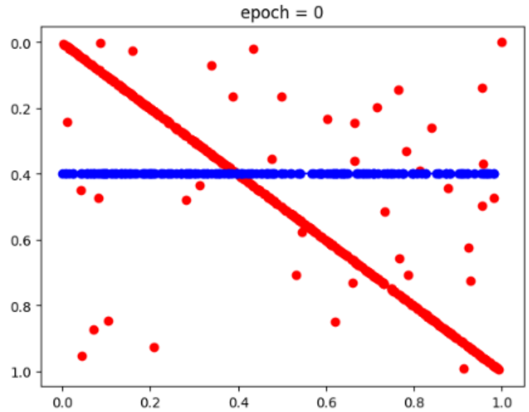


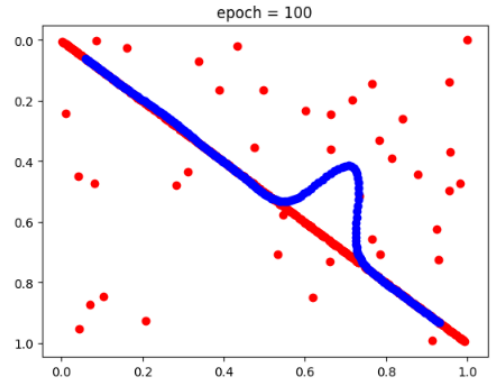


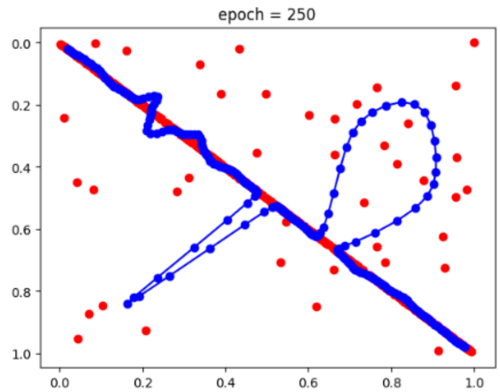
We made two more non uniform distributions as requested. The first distribution consists of data that is more likely to appear in the bottom right corner of the 1x1 square. |We tried to fit a topologically shaped line of neurons to this distribution, and these were the results



And the second distribution consist of data that is more likely to appear in a diagonal line across the top left corner to the bottom left corner. And we gave epsilon probability to appear anywhere else in the image. (I.E 0.9 change to appear on the diagonal line, and 0.1 chance to appear anywhere else on the 1x1 rectangle).

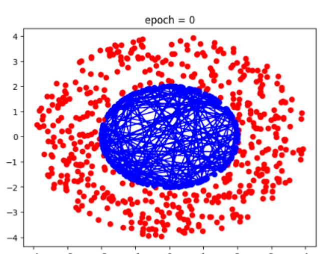


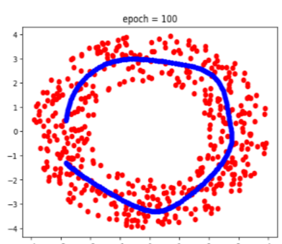


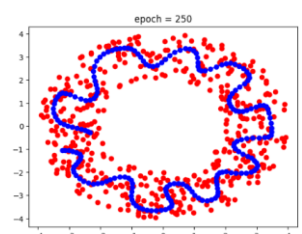


Question A.2

For the donut distribution we've created a topology of the neurons to be a circle with radius 2 around 0,0. To create the donut shaped data, we sampled 500 radiuses and corresponding angles, and then we converted this polar representation to cartesian with x,y coordinates to receive all the 500 points across the given range of circles. (I.E 16 > radiuses > 4) And the results are as follows:

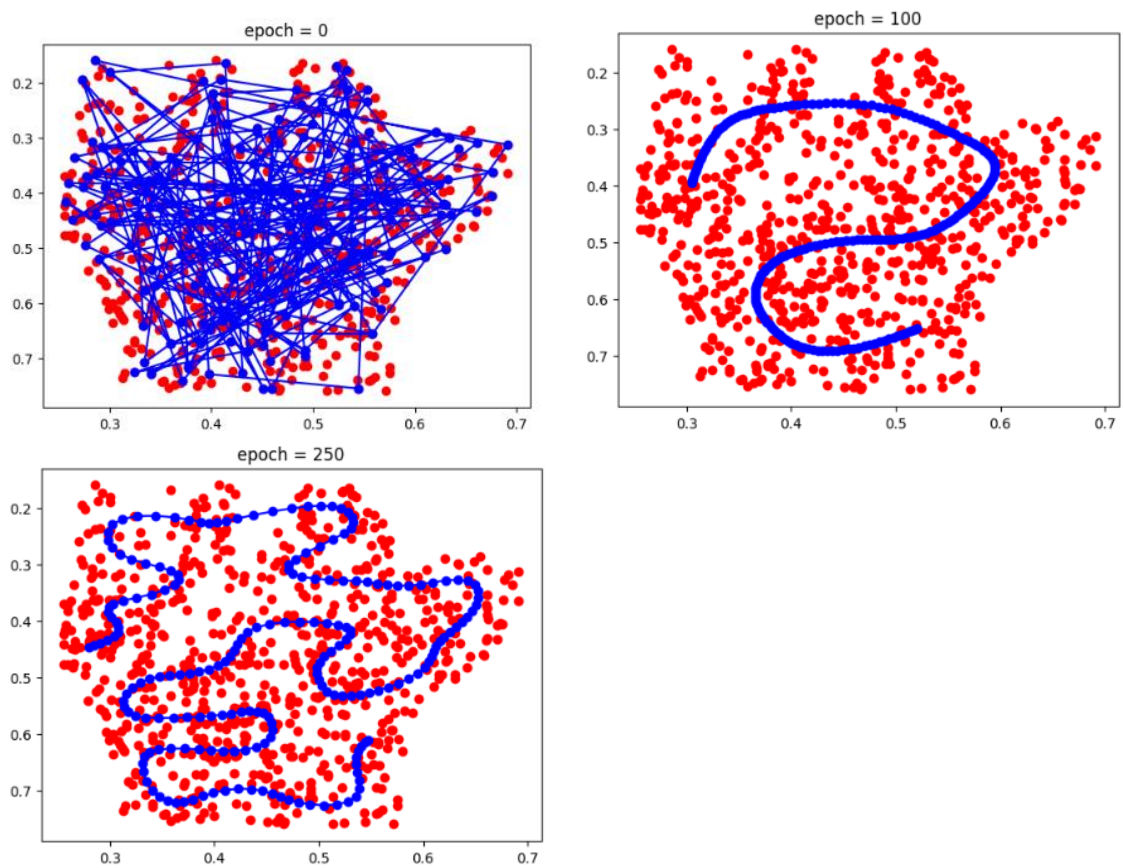






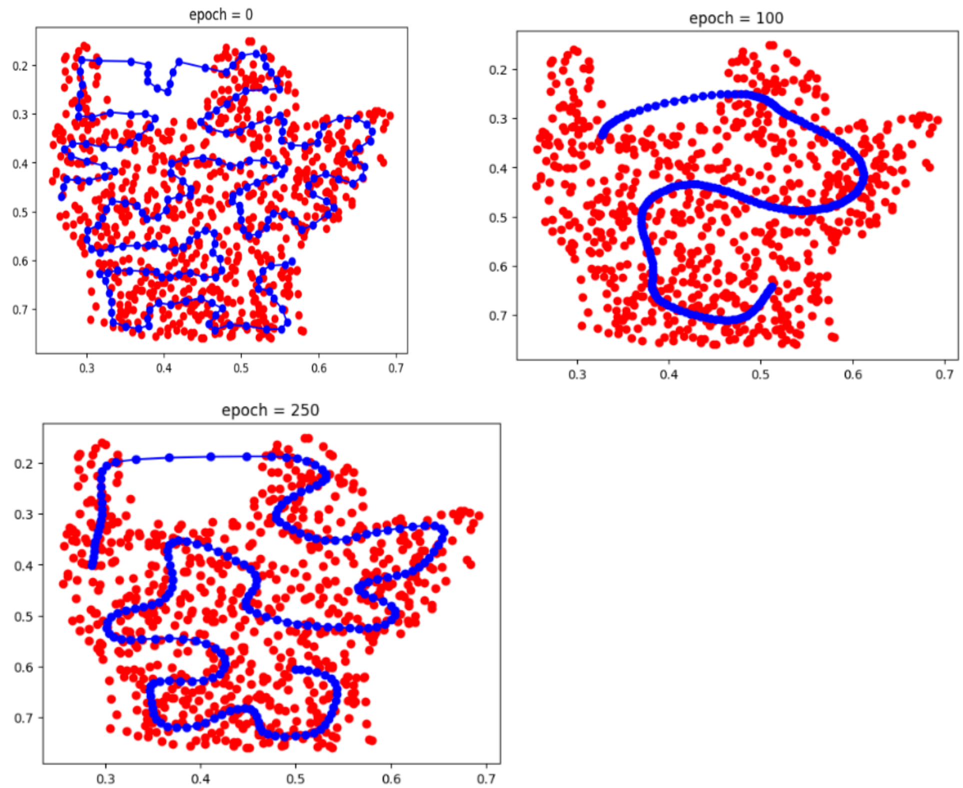
Question B

For the hand distribution we drew a hand in paint, and covered it with black color. Afterwards we've exported the image to png format and read it with opencv to a numpy array. Afterwards we could filter out all the white pixels and receive coordinates of the black pixels we've colored. Afterwards we've normalized the points to (0,1) range. Then we sampled 750 points to get the dataset of a hand. And we took 400 points uniformly at random to get the hand mesh. These are the results:



Question B.2

We've saved the neurons from part B and then we adapted them to a mesh without a middle left finger. This is the adaptation of the hand:



Inline Question:

What happens as the number of iterations of algorithm increases?

Answer:

As the number of iterations increases, the network's representation of the input data becomes more refined and accurate. The neurons align themselves in a way that reflects the distribution of the data and the boundaries between different clusters or regions. The network starts to capture the essential features and characteristics of the data.

We added the full implementation to this document but if you want to run it you can do so in this link, and please add a shortcut to the folder to "My Drive" folder in your drive: https://drive.google.com/drive/folders/1ZwNCDPIg2moWynqB8WziKW_XssaXxc00?usp=sharing

Code Implementation

Connecting Colab notebook to drive

```
# this mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'CV7062610/assignments/assignment3/'
FOLDERNAME = 'Neorio/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))
```

```
Mounted at /content/drive
```

Second try implementation

```
def create_neurons(neurons_num):
    neurons = {}
    for i in range(neurons_num):
        neurons.update({i: [0.45 + i/neurons_num * 0.1, 0.5]})

    return neurons

# Selecting the winning neuron - the closest neuron via Euclidean distance
def decide_winner(p, N):
    min_dist = float("inf")
    winner_index = 0
    # going over all neurons
    for n in N.keys():
        dist = 0
```

```
        # going over all elements
        for i in range(len(p)):
            dist += (p[i] - N[n][i])**2
        # formula for Euclidean distance
        dist = math.sqrt(dist)
        # updating the current winner
        if dist < min_dist:
            winner_index = n
            min_dist = dist
    return winner_index

def create_data(sample_num):
    data = {}
    for i in range(sample_num):
        x, y = np.random.uniform(size=2)
        while (x-0.5)**2 + (y-0.5)**2 > 0.25:
            x, y = np.random.uniform(size=2)
        data.update({i: (x, y)})

    return data

# N := all neurons
# c := winning neuron's index
# X := current input
# alpha := current learning rate
# sigma := current neighborhood size
# radius := current neighborhood radius
def update_weights(N, c, X, alpha, sigma, radius):
    # go over all neurons with index in radius r from the winning neuron
    for j in range(c-radius, c+radius+1):
        # if the index exists:
        if j in N:
            # get current topological neighborhood (will be equal to 1 if the
neuron is the winner)
            h = math.exp(-((c - j)**2)/(2*(sigma**2)))
            # go over all elements
            for i in range(len(X)):
                # update weight according to the formula
                N[j][i] = N[j][i] + alpha * h * (X[i] - N[j][i])
    # return the new weights
    return N

def train(P, N, epoches, learning_rate, neighborhood_size, neighborhood_radiu
s):
    for t in range(epoches):
        # going over all input vectors
        if t % 50 == 0:
            display(P, N, t)
        for p in range(len(P)):
            # update alpha, sigma, and radius
```

```
            alpha = learning_rate * (1-(t/epoches))
            sigma = neighborhood_size * (1-(t/epoches))
            radius = round(neighborhood_radius * (1-(t/epoches)))
            # check the winning neuron
            c = decide_winner(P[p], N)
            # update the weights
            N = update_weights(N, c, P[p], alpha, sigma, radius)

def display(P, N, t):
    px, py = [], []
    for i in P.keys():
        px.append(P[i][0])
        py.append(P[i][1])

    nx, ny = [], []
    for i in N.keys():
        nx.append(N[i][0])
        ny.append(N[i][1])

    plt.plot(px, py, 'ro')
    plt.plot(nx, ny, 'bo-')
    plt.gca().invert_yaxis()
    plt.title("epoch = " + str(t))
    plt.show()

def plotGraph(data, neurons=None):
  # Plotting the final Kohonen map
  plt.scatter(data[:,0], data[:,1], c='blue')
  if neurons is not None:
    plt.plot(neurons[:, 0], neurons[:, 1], c='red', marker='o')
  plt.gca().invert_yaxis()
  plt.show()
```

Dataset

```
def createRectangleDist(data_size,p1: np.ndarray = np.array([0,0]),p2: np.nda
rray = np.array([1,1])):

  points = np.random.rand(data_size,2)
  m = [np.min([p1[0],p2[0]]), np.min([p1[1],p2[1]])]
  points *= (p2 - p1)
```

```
  points += np.array(m)
  return points

import math
import random
def createDonutDist(data_size):
  # {<x.y> | 4<= x^2 +y^2 <= 16}
  radiuses = np.sqrt(np.random.uniform(4,16, data_size))
  thetas = np.random.uniform(0, 2*np.pi,data_size)
  data = np.array([radiuses * np.sin(thetas), radiuses * np.cos(thetas)]).T
  return data

data = createDonutDist(100)
# print(data)
i = 0
for x,y in data:
  if x **2 + y**2 > 16 or 4 > x **2 + y**2:
    i+=1
print('x **2 + y**2 > 16 or 4 > x **2 + y**2' if i != 0 else "4 <= x **2 + y*
*2 <= 16")
```

```
4 <= x **2 + y**2 <= 16
```

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def createDistByPath(data_size, image_path):
  hand_img = cv2.imread(image_path)
  hand_img = cv2.cvtColor(hand_img, cv2.COLOR_BGR2GRAY)
  # plt.imshow(hand_img, cmap='gray')
  # plt.axis('off')  # Remove the axes
  # plt.show()
  print(hand_img.shape)
  # print(np.indices(hand_img.shape)[hand_img != 255])
  black_pixels = (hand_img != 255).nonzero()
  black_pixels = ( black_pixels[1] / hand_img.shape[1],black_pixels[0] / hand
_img.shape[0])
  data = np.vstack(black_pixels).T
  return data[np.random.choice(data.shape[0],data_size, replace=False)]

def createHandDist(data_size):
  image_path = '/content/drive/MyDrive/Neorio/best_hand.png'
  return createDistByPath(data_size,image_path)

def createHandNoFingerDist(data_size):
  image_path = '/content/drive/MyDrive/Neorio/best_hand_no_finger.png'
  return createDistByPath(data_size,image_path)

def getNonUniformDist2(rect_size):
    rect_indices = np.arange(rect_size * rect_size)
```

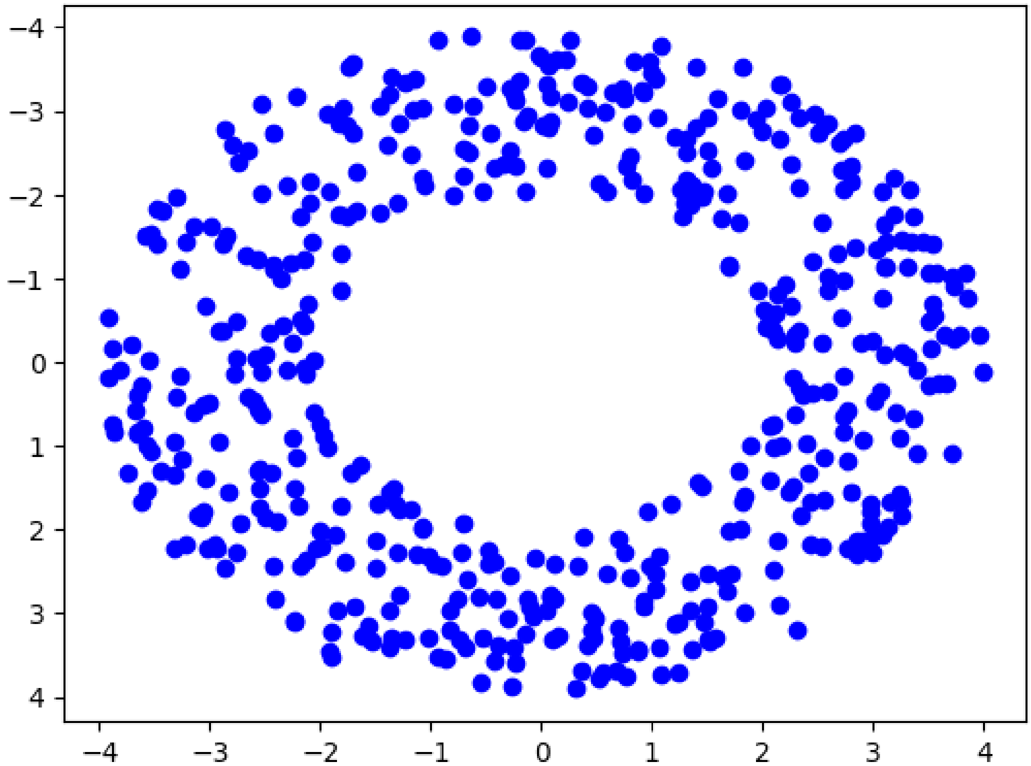
```
    rect_dist = np.ones((rect_size * rect_size,))
    rect_dist /= (rect_size * rect_size) - rect_size
    rect_dist /= 10
    for i in range(rect_size):
        rect_dist[i * rect_size + i] = 9 / (10 * rect_size)
    return rect_dist / np.sum(rect_dist)

# Define the parameters
num_neurons = 20  # Number of neurons
num_iterations = 100  # Number of iterations
learning_rate = 0.1  # Learning rate

rect_size = 500
donut_size = 500
hand_size = 750
finger_size = 750


# Generate the training data (uniform distribution)
# data = np.random.rand(1000, 2)
rect_data = createRectangleDist(rect_size)
rect_data_non_uniform1 = np.zeros((rect_size,2))
rect_data_non_uniform1[:,0] = np.random.choice(np.linspace(0, 1, rect_size),
size=rect_size, p=np.linspace(0, 1, rect_size)/(rect_size//2))
rect_data_non_uniform1[:,1] = np.random.choice(np.linspace(0, 1, rect_size),
size=rect_size, p=np.linspace(0, 1, rect_size)/(rect_size//2))

rect_data_non_uniform2 = np.zeros((rect_size,2))
probabilities = getNonUniformDist2(rect_size)

point_indices = np.random.choice(np.linspace(1, rect_size * rect_size, rect_s
ize*rect_size), size=rect_size, p=probabilities)
rect_data_non_uniform2[:,0] = point_indices // rect_size
rect_data_non_uniform2[:,1] = point_indices % rect_size
rect_data_non_uniform2 /= rect_size

donut_data = createDonutDist(donut_size)
hand = createHandDist(hand_size)
no_finger_hand = createHandNoFingerDist(finger_size)
```

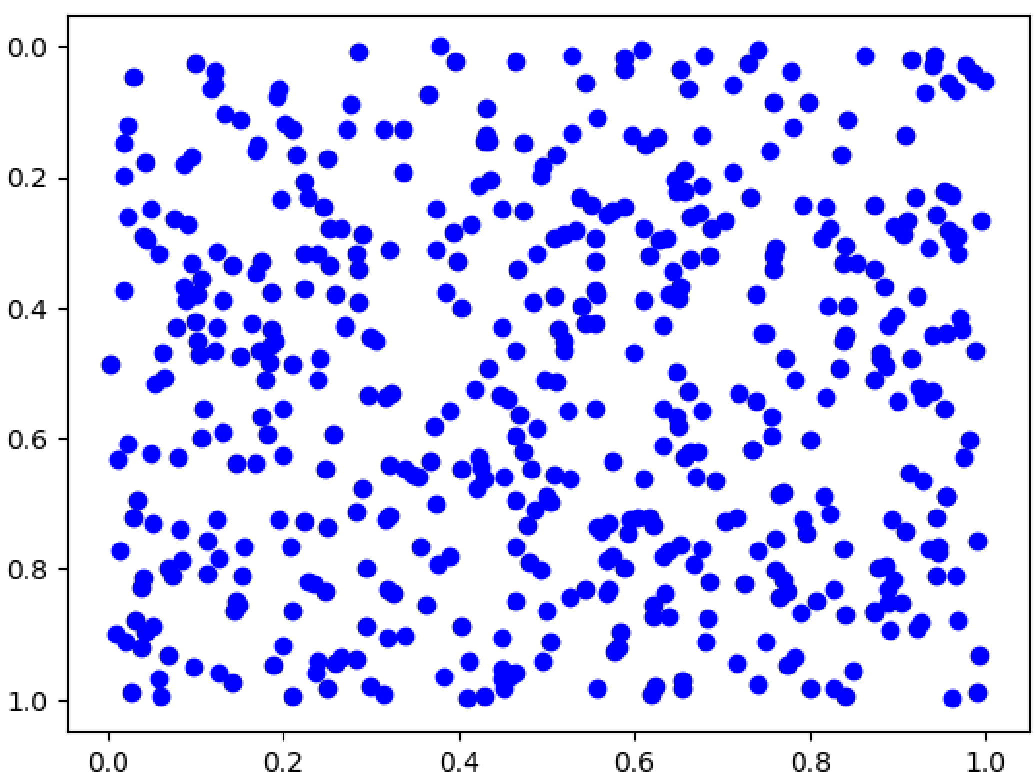
```
(617, 933)
(617, 933)
```

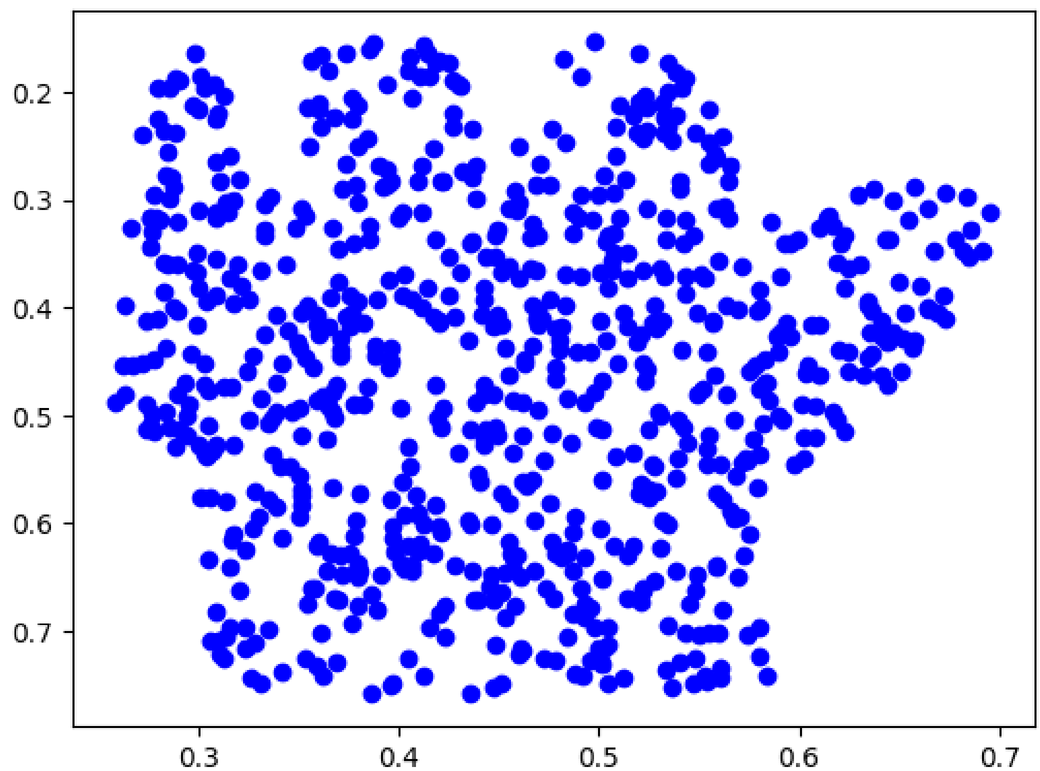
Plot diffrent distributions

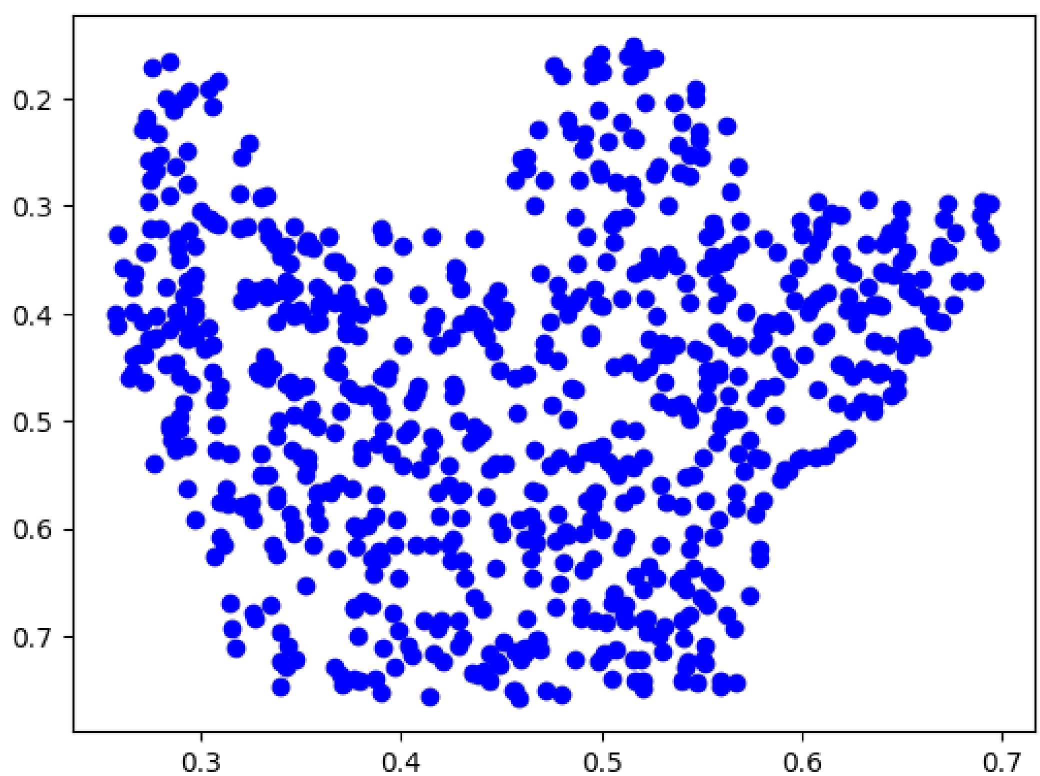
```
plotGraph(donut_data)
plotGraph(rect_data)
plotGraph(hand)
plotGraph(no_finger_hand)
```

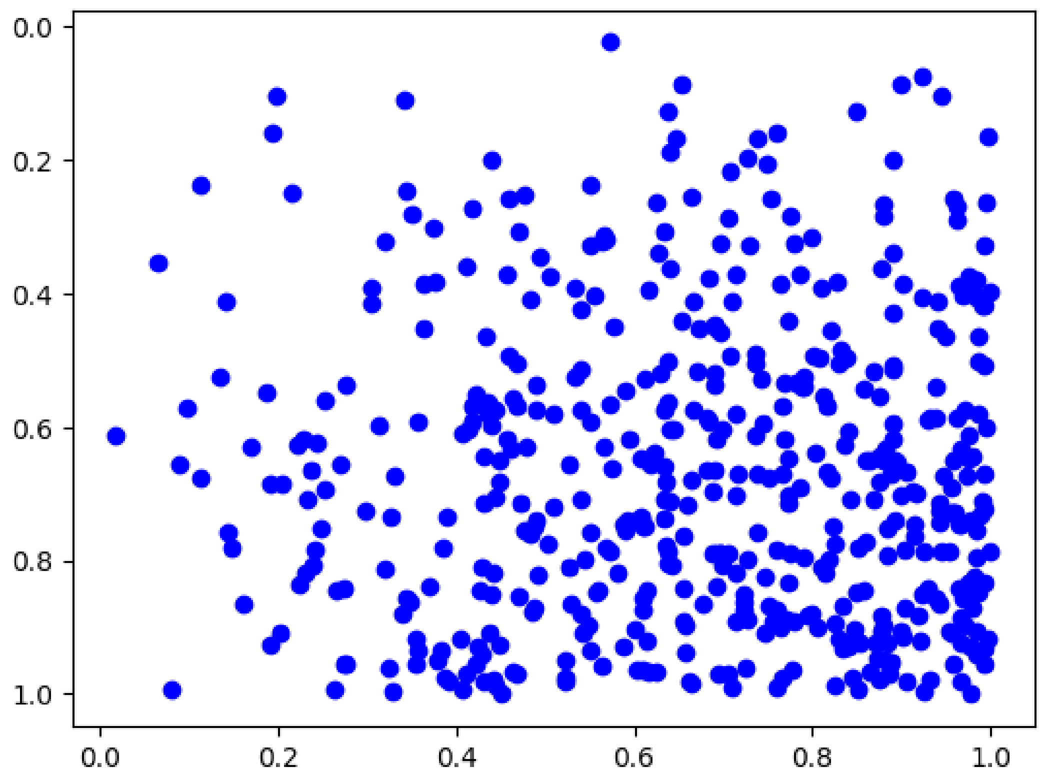
```
plotGraph(rect_data_non_uniform1)
plotGraph(rect_data_non_uniform2)
```

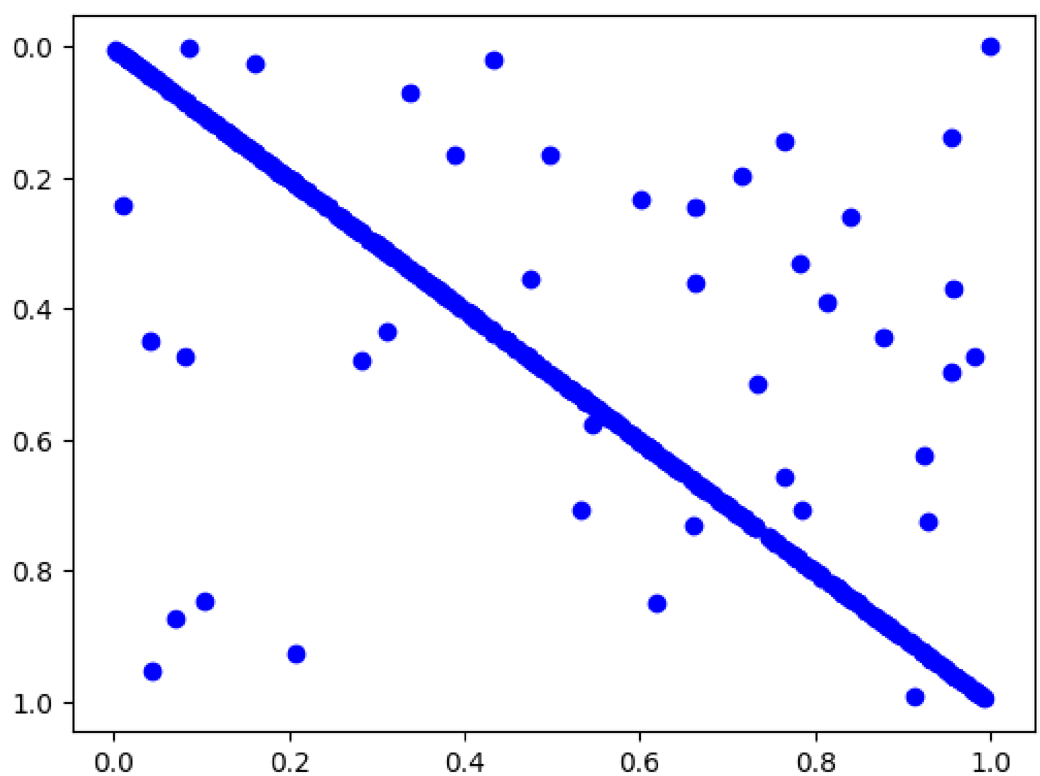





0.2
0.3
0.4
0.5
0.6
0.7
0.3
0.4
0.5
0.6
0.7


0.2
0.3
0.4
0.5
0.6
0.7
0.3
0.4
0.5
0.6
0.7


0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0


0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

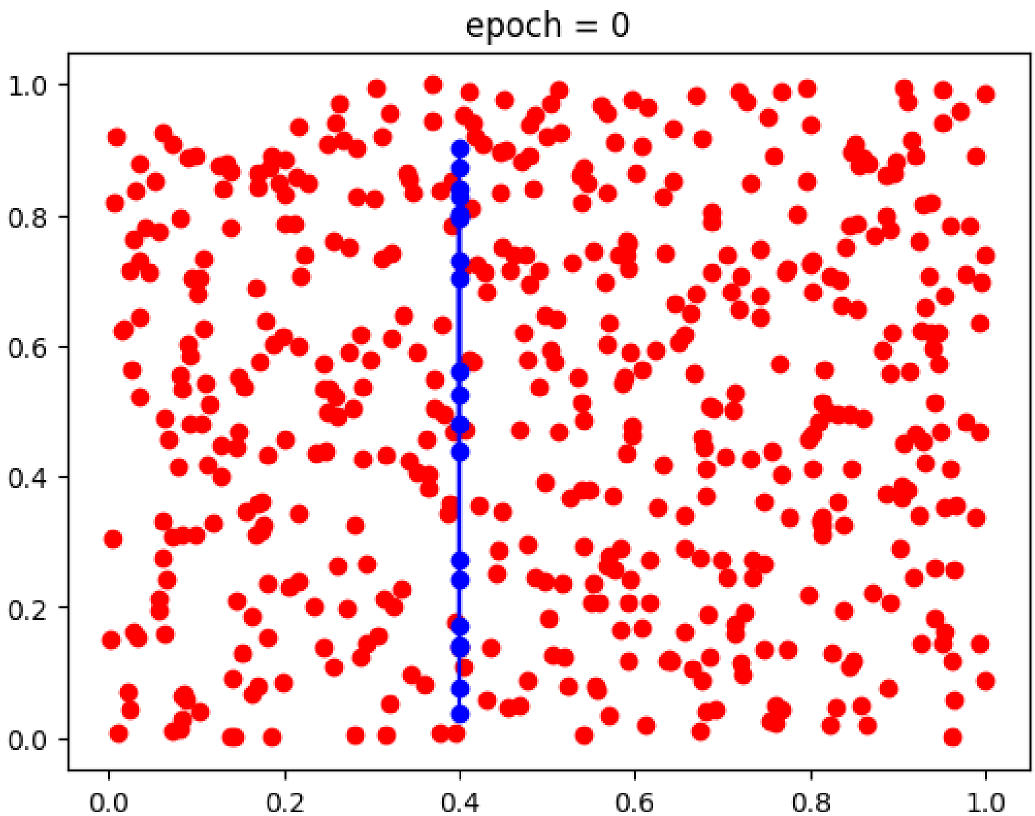
Part A

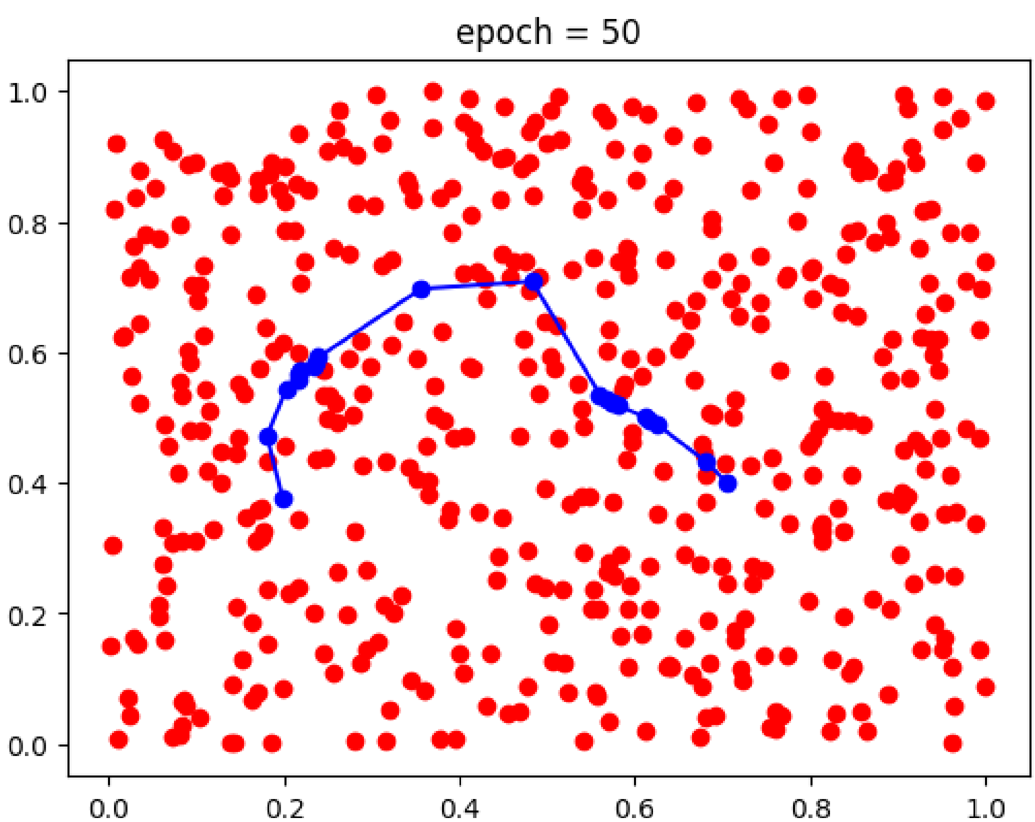
###Create the topological order of the neurons

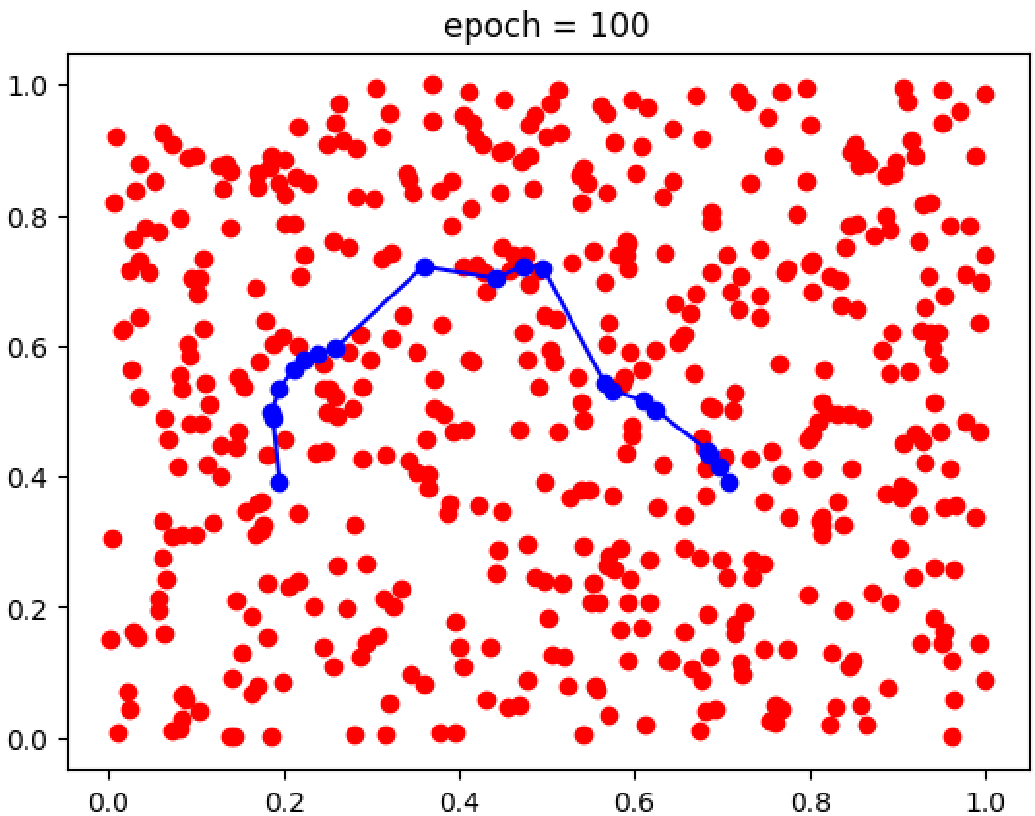
```
def makeNeuronsByDist(dist, num_neurons, lst_size = None):
  neurons = None
  if dist == 'line':
    neurons = np.zeros((num_neurons,2))
    neurons[:,1] = 0.4
    neurons[:,0] = np.random.rand(num_neurons)
  elif dist == 'circle':
     neurons = np.zeros((num_neurons,2))
     radius = 2
     thetas = np.random.uniform(0,2 * np.pi, num_neurons)
     neurons[:,0] = np.sin(thetas) * radius
     neurons[:,1] = np.cos(thetas) * radius
  elif dist == 'hand':
    neurons = hand[np.random.randint(0,hand.shape[0], num_neurons),:]

  if neurons is None:
    return None
  if lst_size is None:
    return neurons

  lst = []
  for i in range(lst_size):
    lst.append(neurons.copy())
  return lst
```

Running Kohonan algorithm with 20 neurons on a line to fit the rectangle Distribution

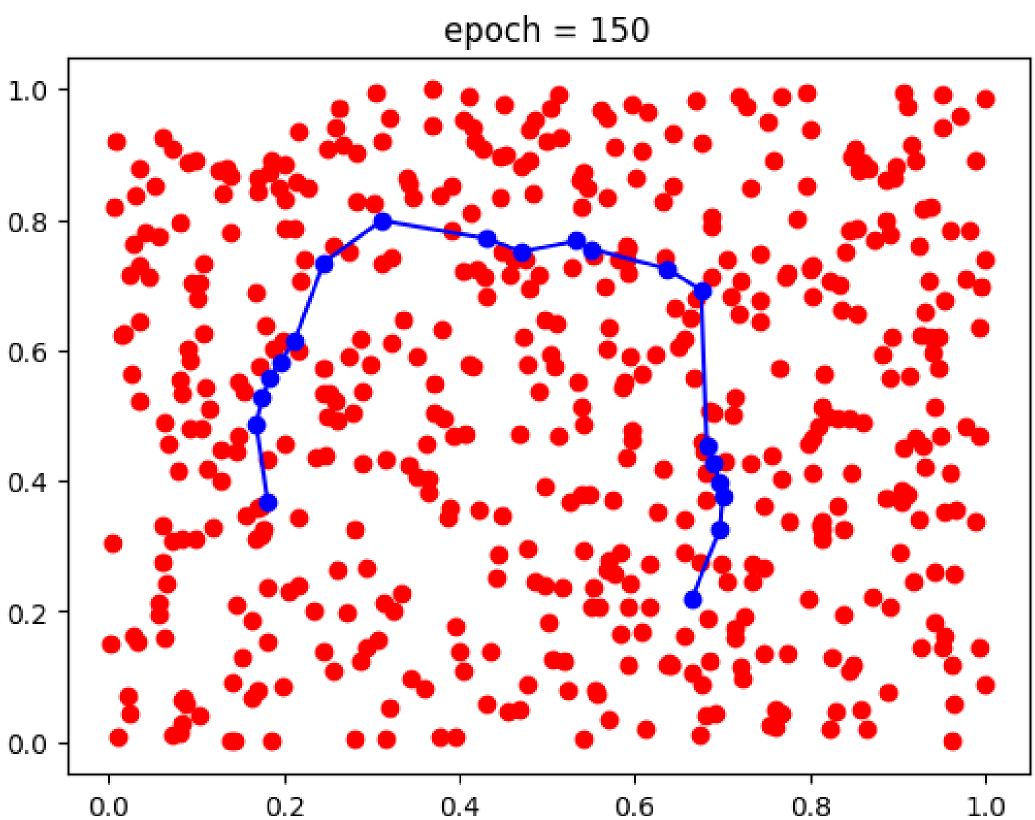
```
# Running Kohonan algorithm with 20 neurons on a line to fit the rectangle Di
stribution

num_neurons = 20
neurons = np.zeros((num_neurons,2))
neurons[:,0] = 0.4
neurons[:,1] = np.random.rand(num_neurons)
P = dict(enumerate(rect_data,0))
N = dict(enumerate(neurons,0))
# find best variables
epoches = 300
learning_rate = 0.2
neighborhood_size = 15
neighborhood_radius = len(N)/2
train(P,N , epoches, learning_rate, neighborhood_size, neighborhood_radius)
```


epoch = 0
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

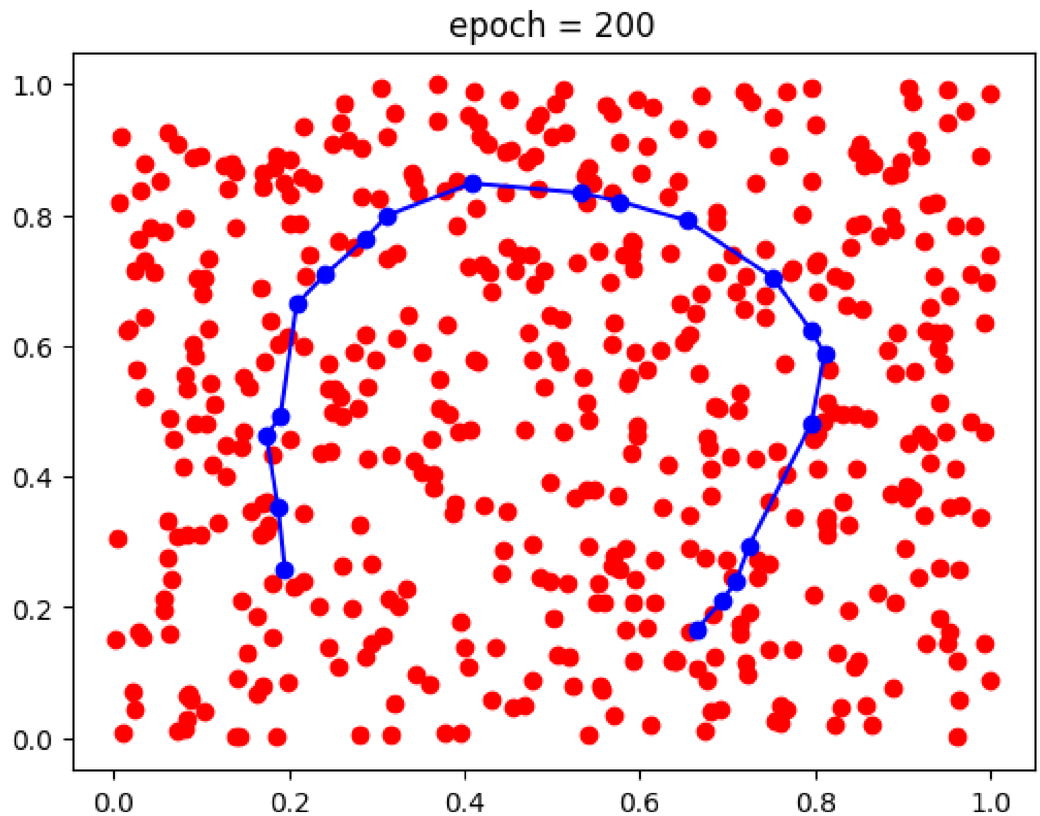

epoch = 50
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

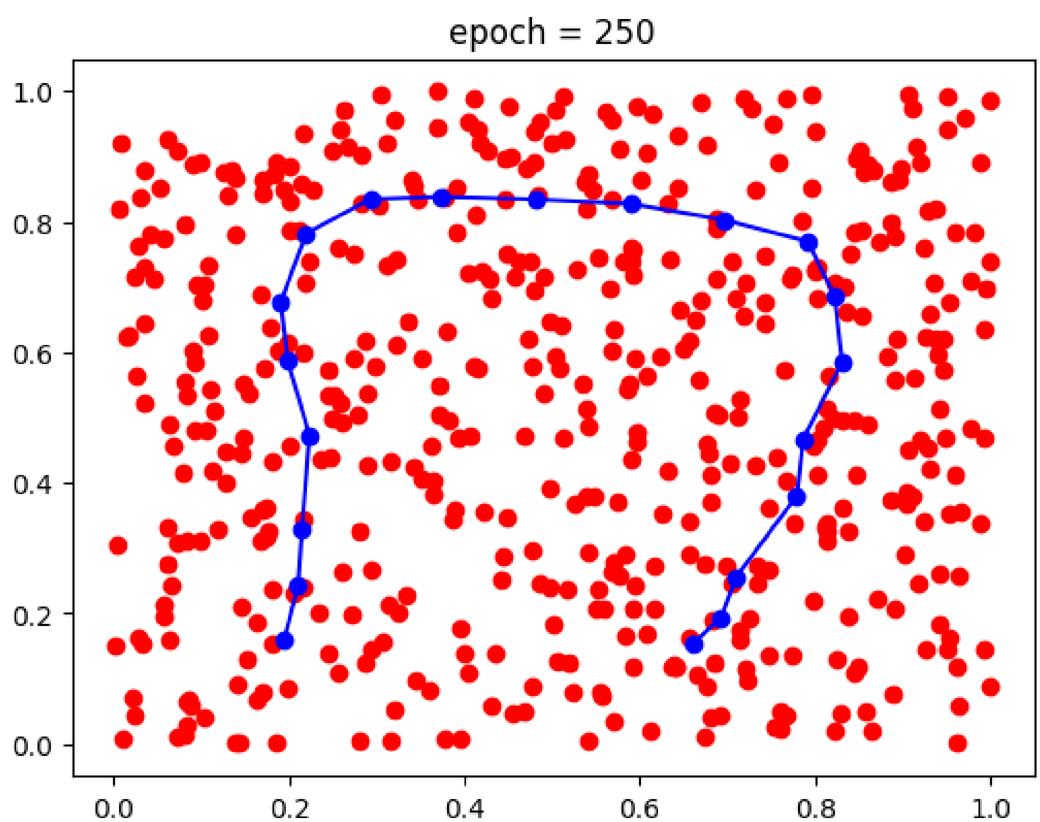


epoch = 100
0.0
0.2
0.4
0.6
0.8
1.0



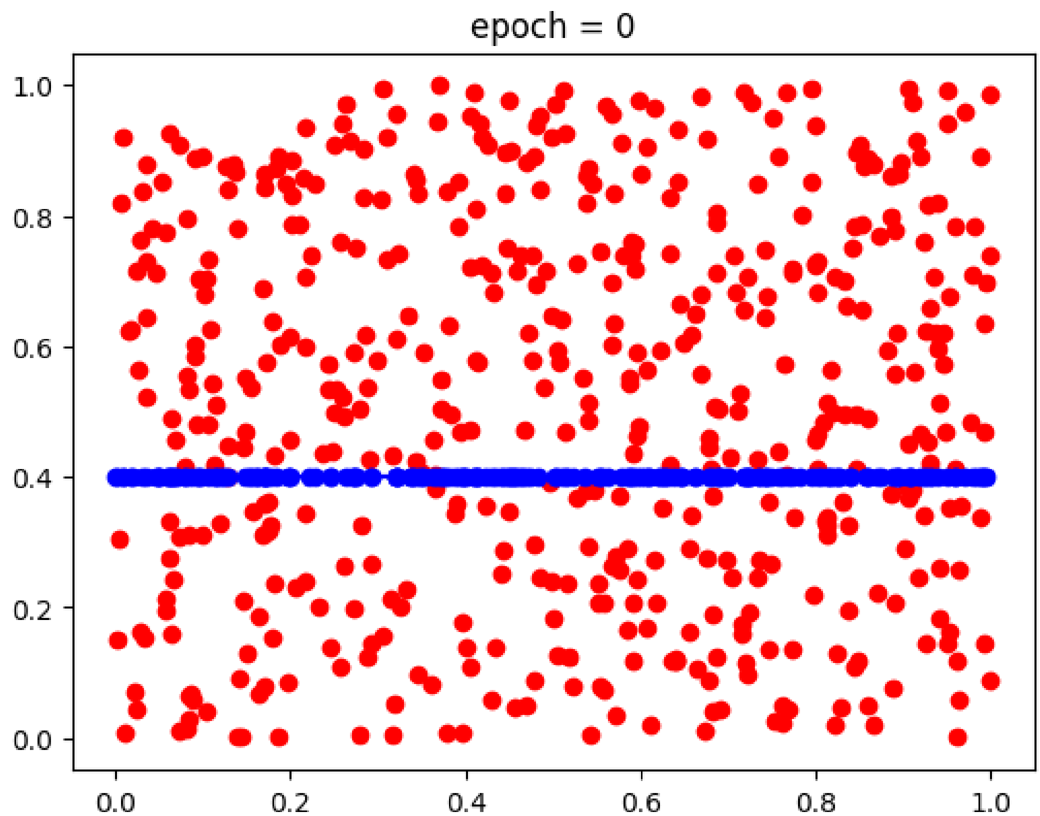
epoch = 150
0.0
0.2
0.4
0.6
0.8
1.0

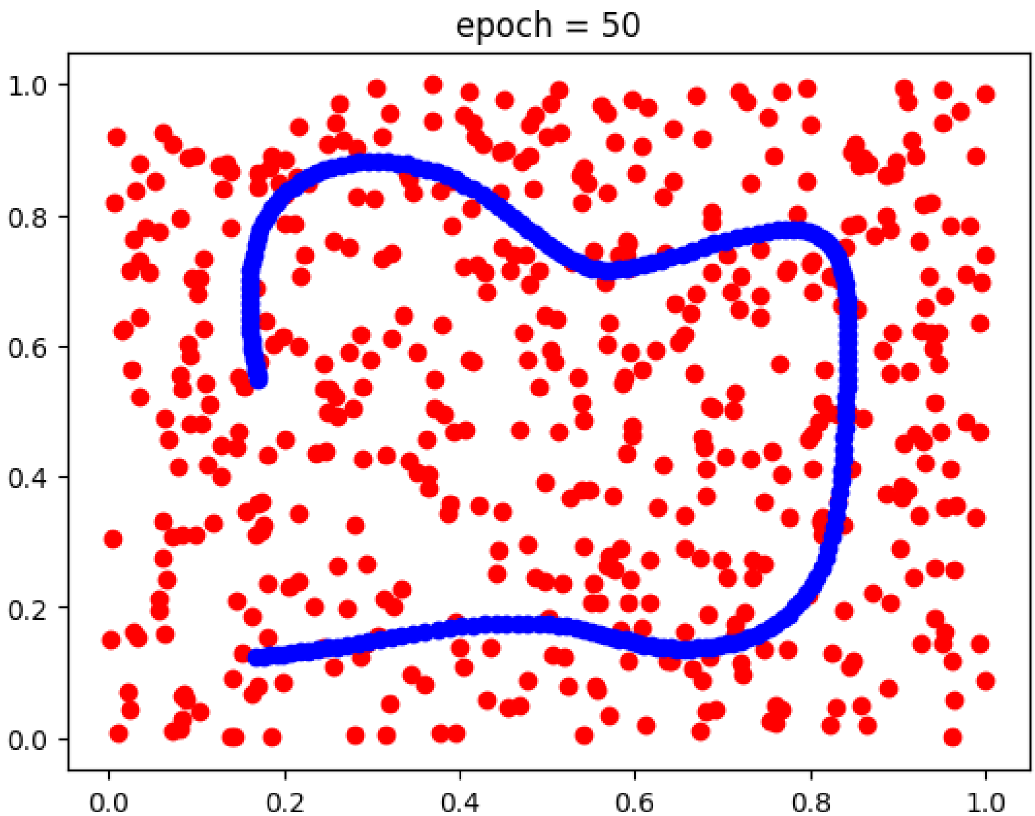

epoch = 200
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

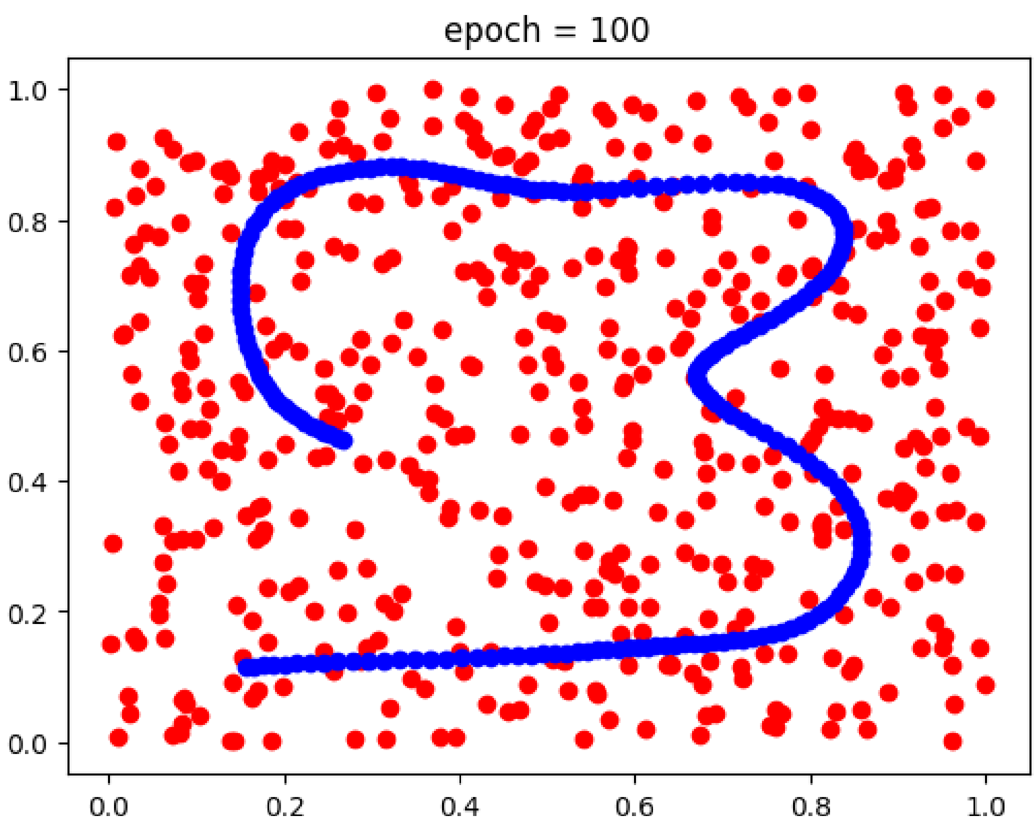

epoch = 250
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

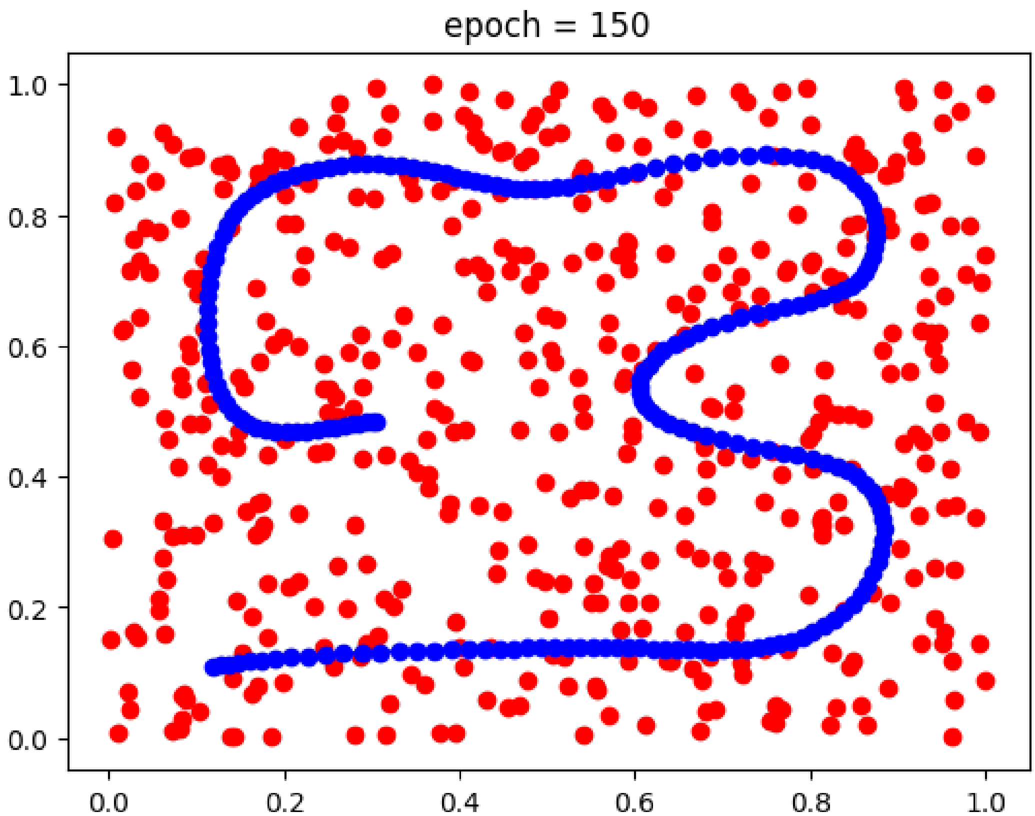
Running Kohonan algorithm with 200 neurons on the rectangle Distribution

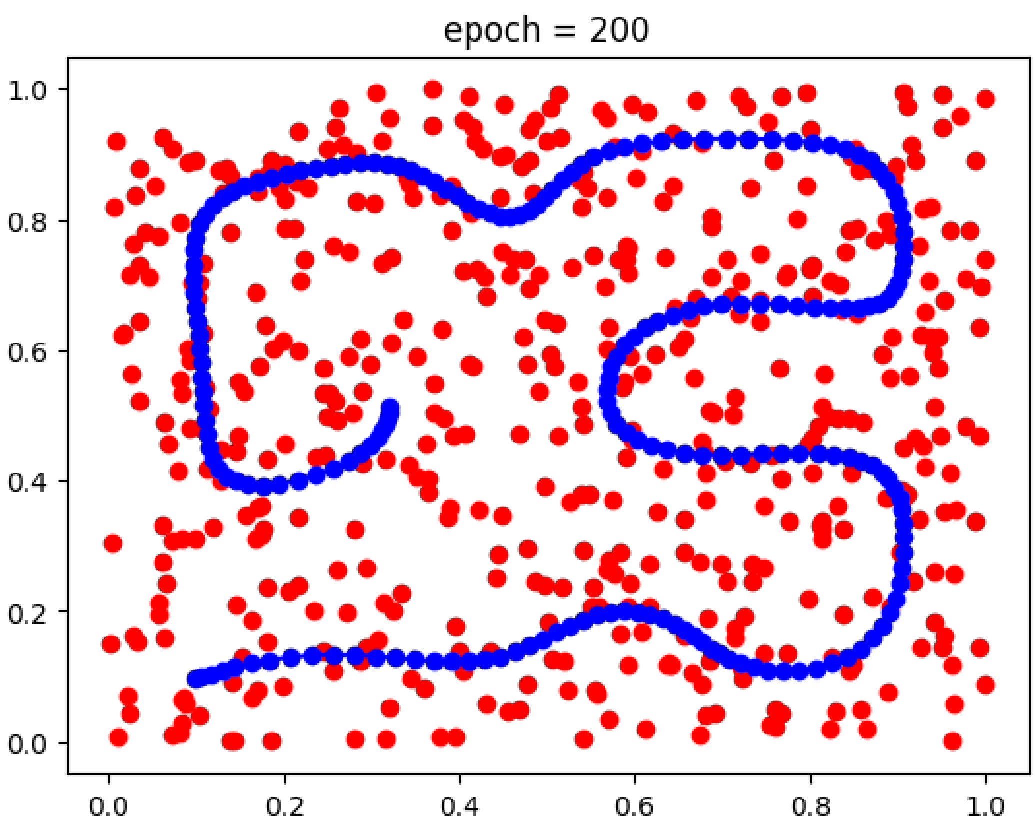
```
# Running Kohonan algorithm with 200 neurons on the rectangle Distribution
num_neurons = 200
neurons = np.zeros((num_neurons,2))
neurons[:,1] = 0.4
neurons[:,0] = np.random.rand(num_neurons)
P = dict(enumerate(rect_data,0))
N = dict(enumerate(neurons,0))
# find best variables
epoches = 300
learning_rate = 0.2
neighborhood_size = 15
neighborhood_radius = len(N)/2
train(P,N , epoches, learning_rate, neighborhood_size, neighborhood_radius)
```

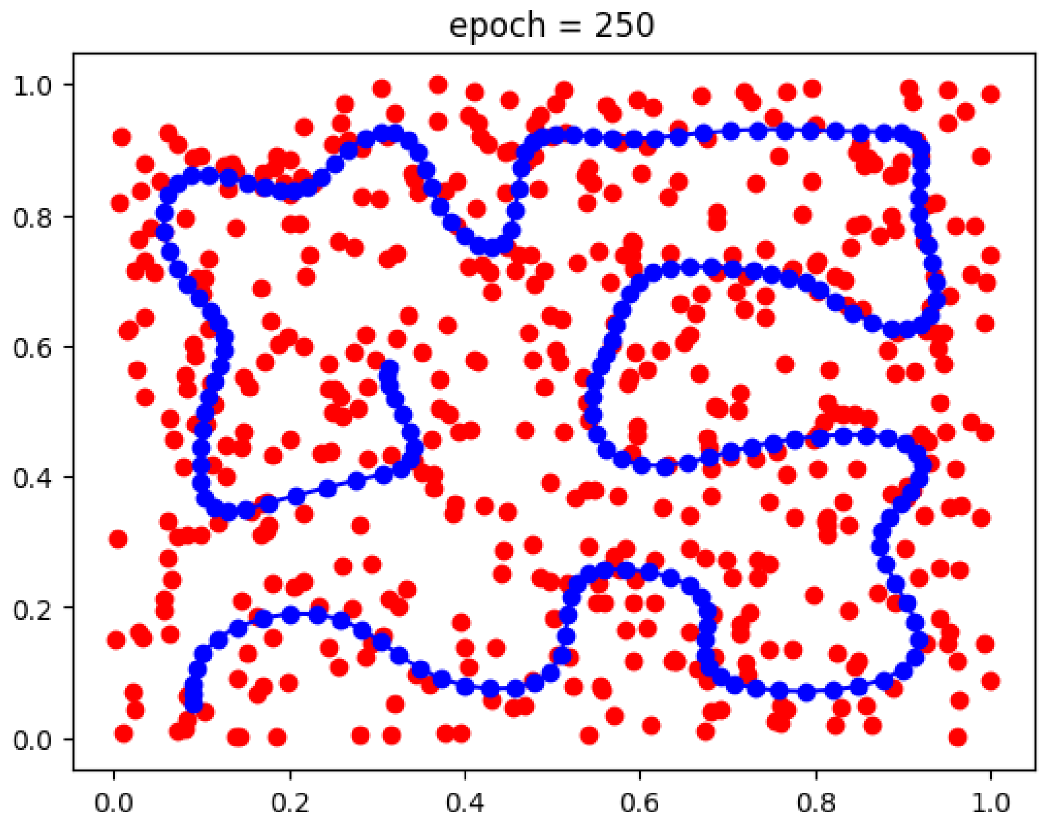



epoch = 50
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0


epoch = 100
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

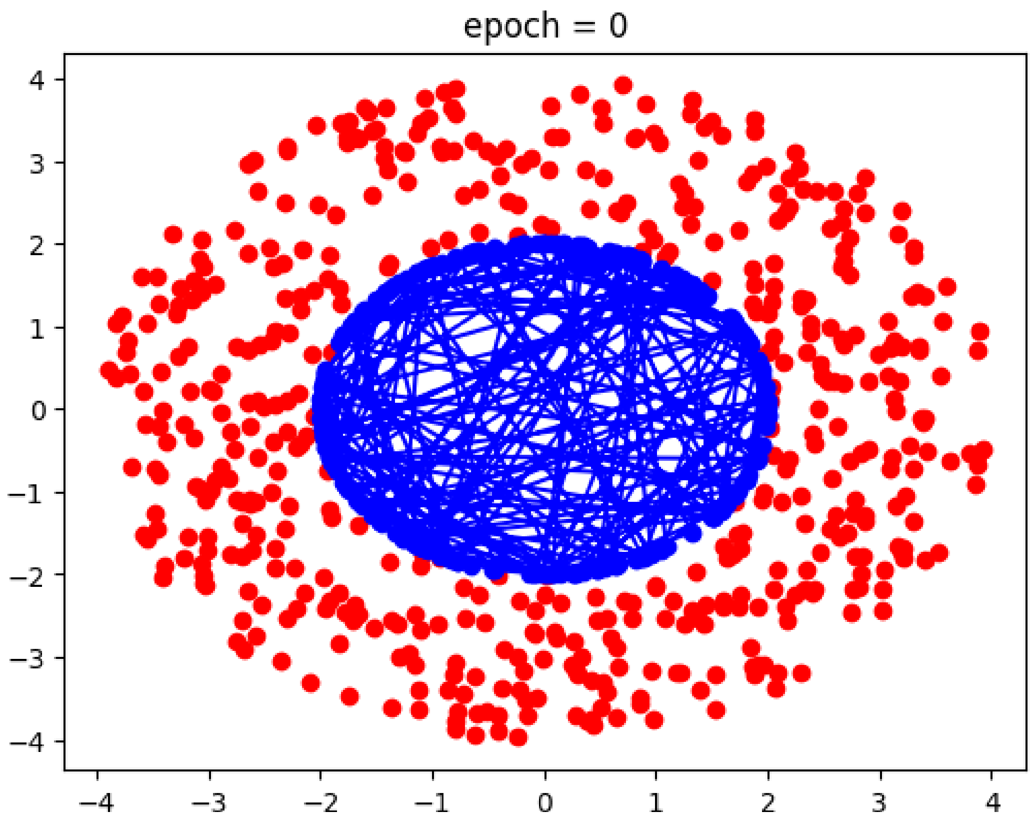

epoch = 150
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

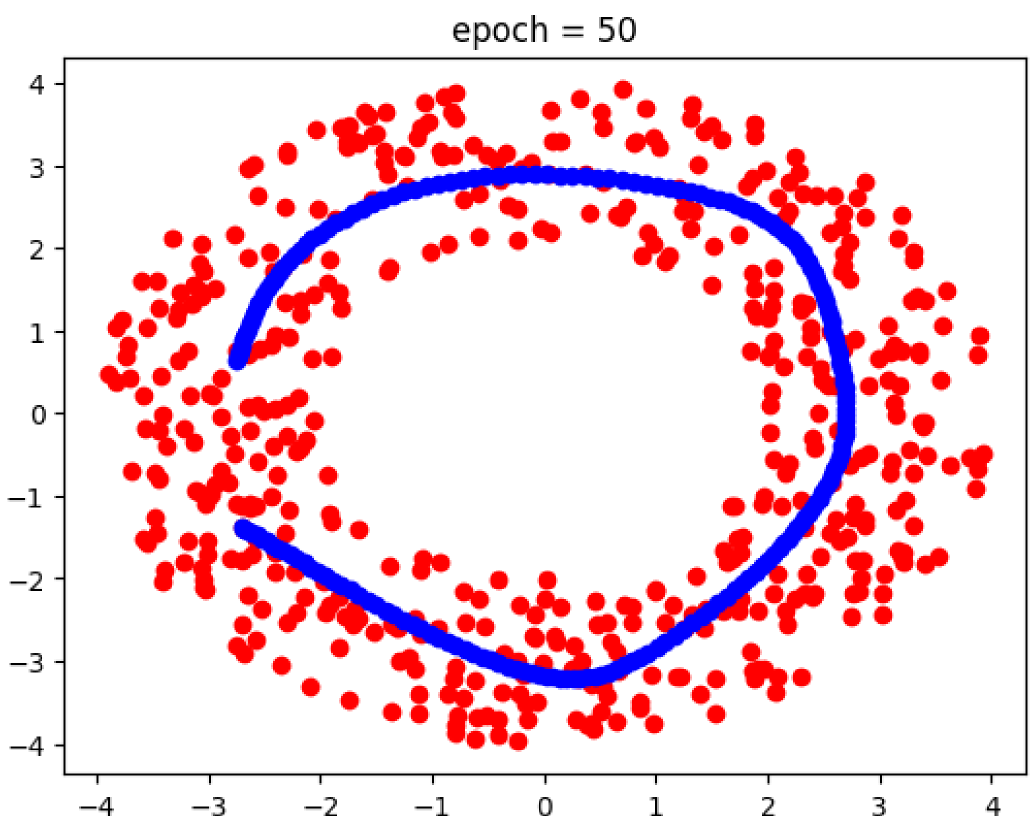

epoch = 200
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

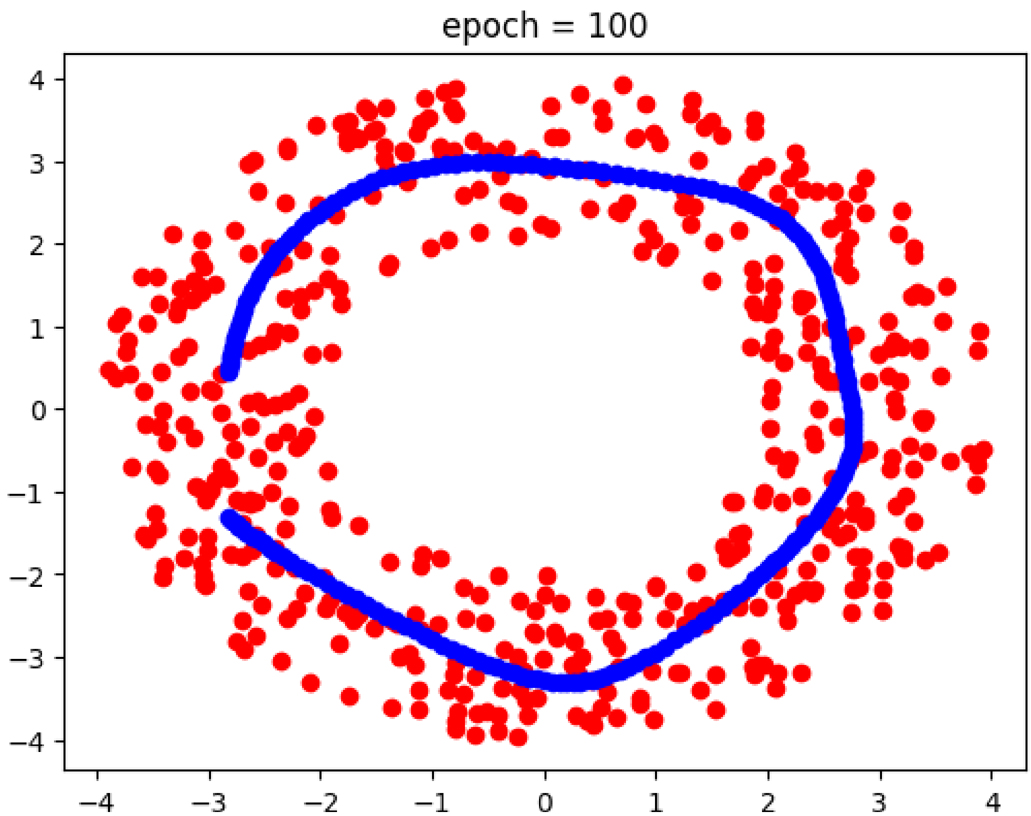


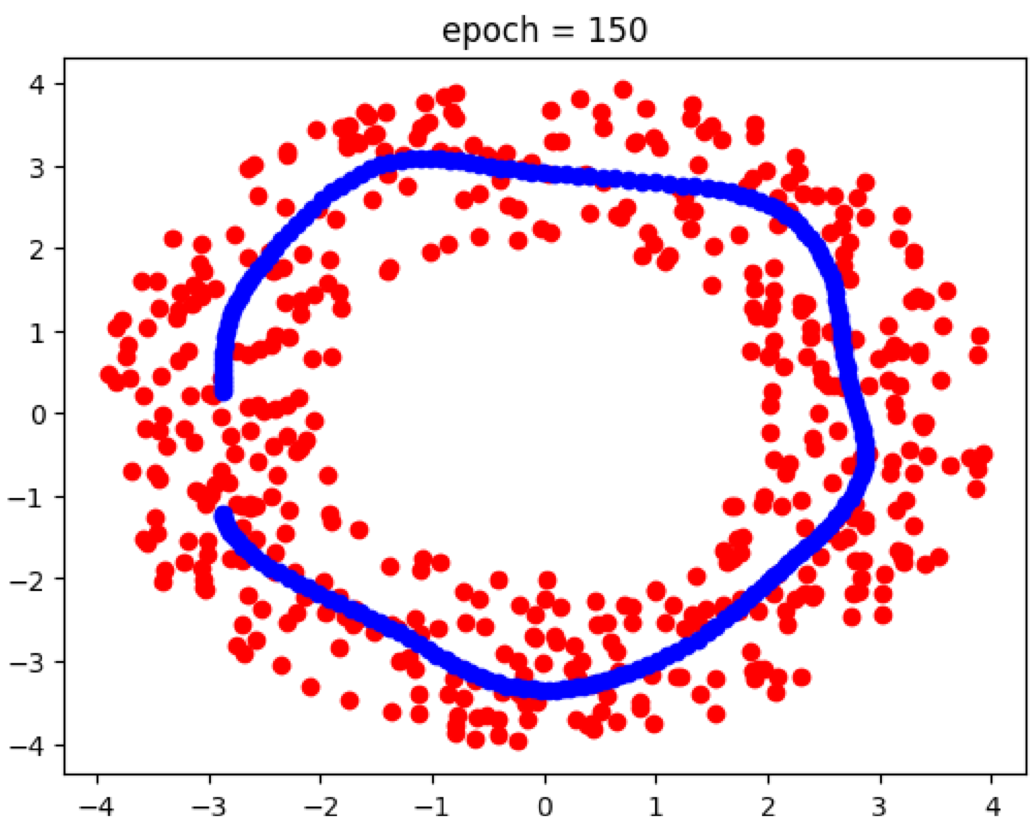
Part A.2

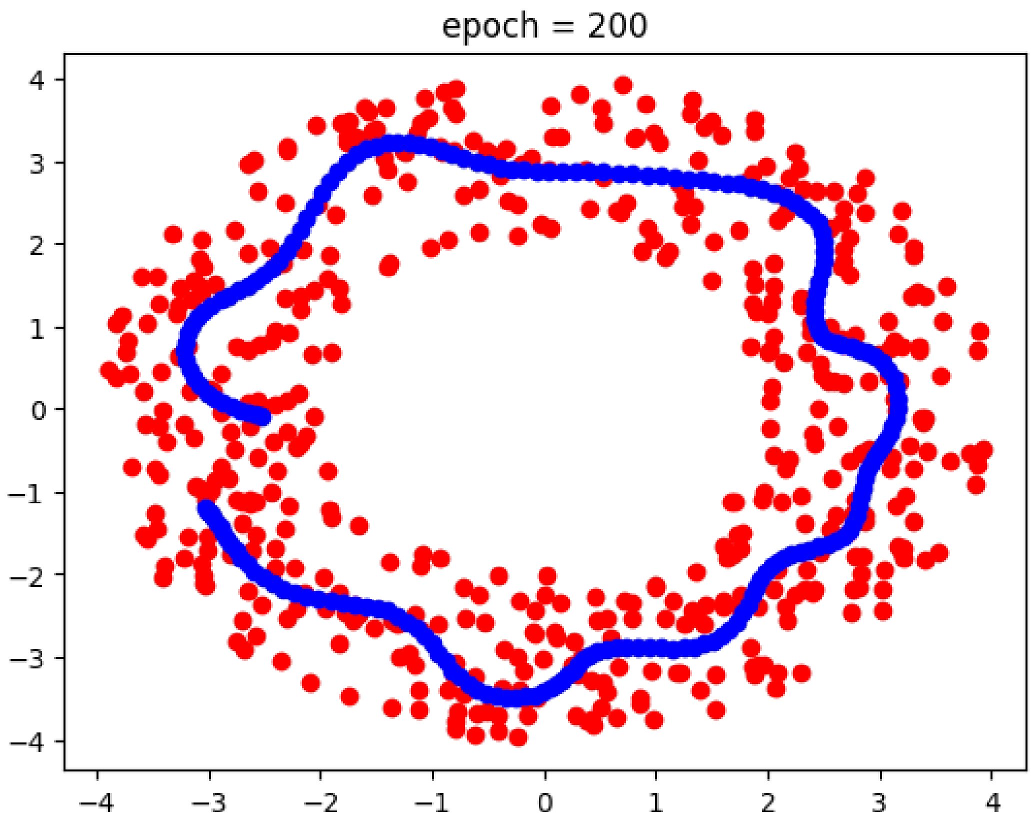
```
num_neurons = 200
neurons = makeNeuronsByDist('circle', num_neurons)
P = dict(enumerate(donut_data,0))
N = dict(enumerate(neurons,0))

# find best variables
epoches = 300
learning_rate = 0.2
neighborhood_size = 15
neighborhood_radius = len(N)/2
train(P,N , epoches, learning_rate, neighborhood_size, neighborhood_radius)
```

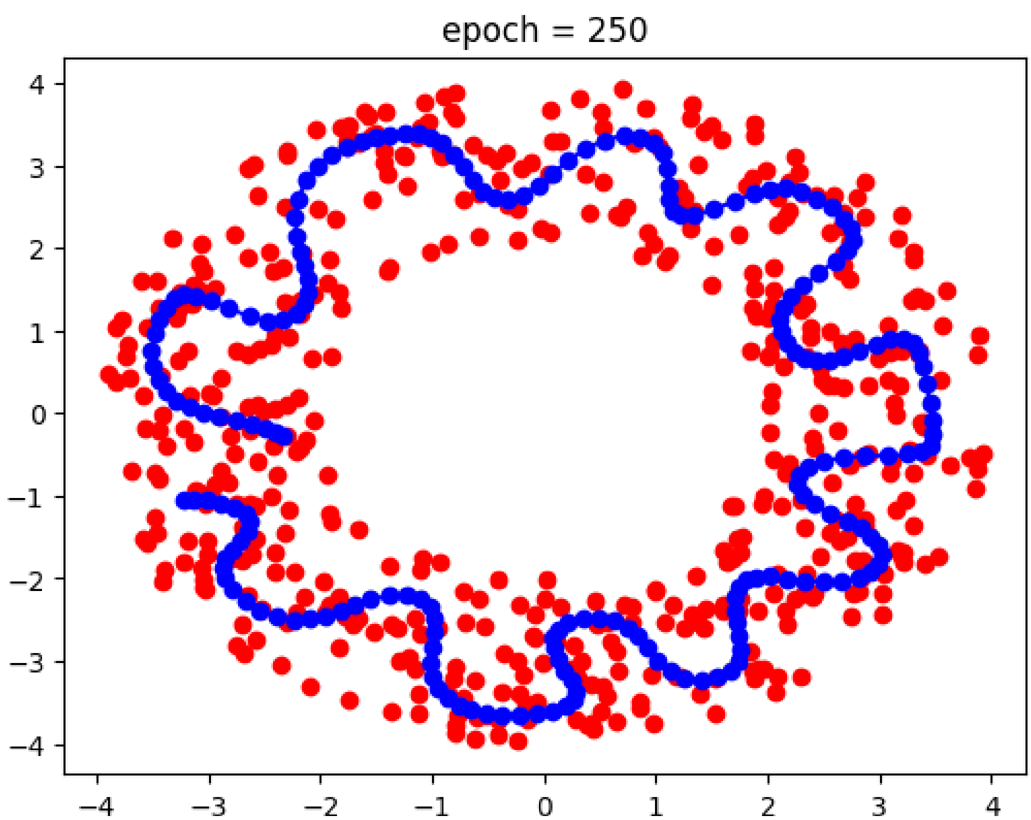

epoch = 0
4
3
2
1
0
−1
−2
−3
−4
−4 −3 −2 −1 0 1 2 3 4


epoch = 50
4
3
2
1
0
−1
−2
−3
−4
−4 −3 −2 −1 0 1 2 3 4


epoch = 100
4
3
2
1
0
−1
−2
−3
−4
−4 −3 −2 −1 0 1 2 3 4

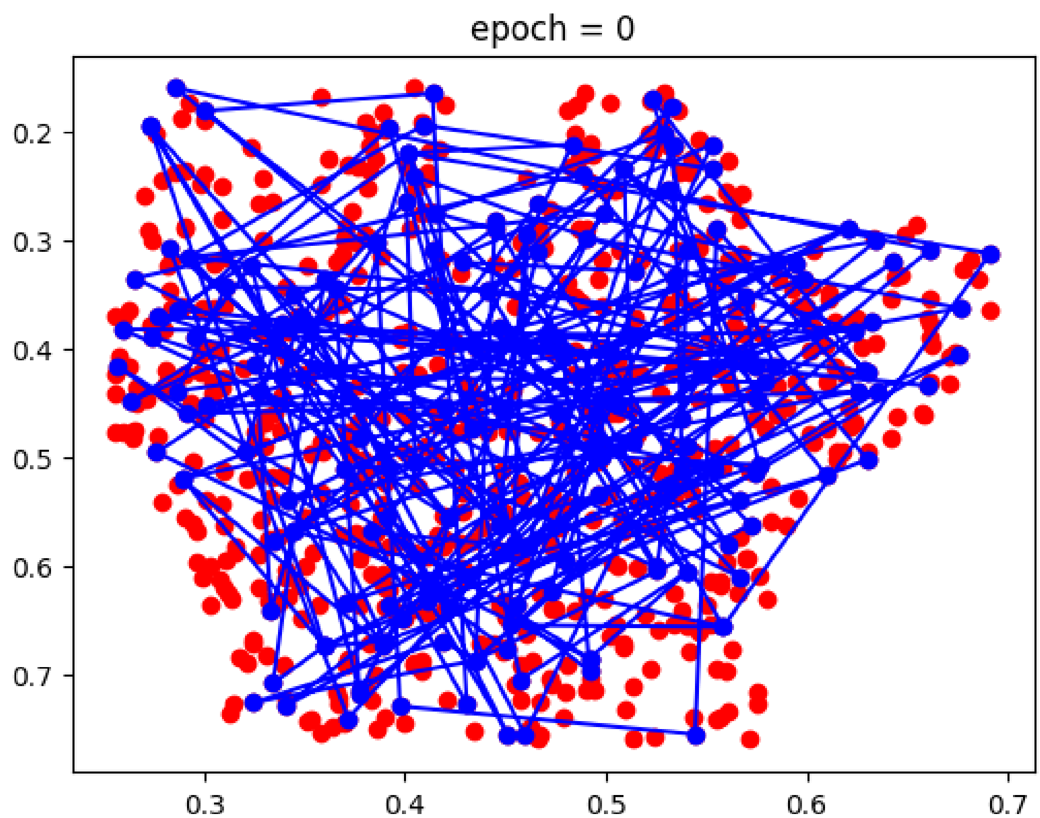

epoch = 150
4
3
2
1
0
−1
−2
−3
−4
−4 −3 −2 −1 0 1 2 3 4

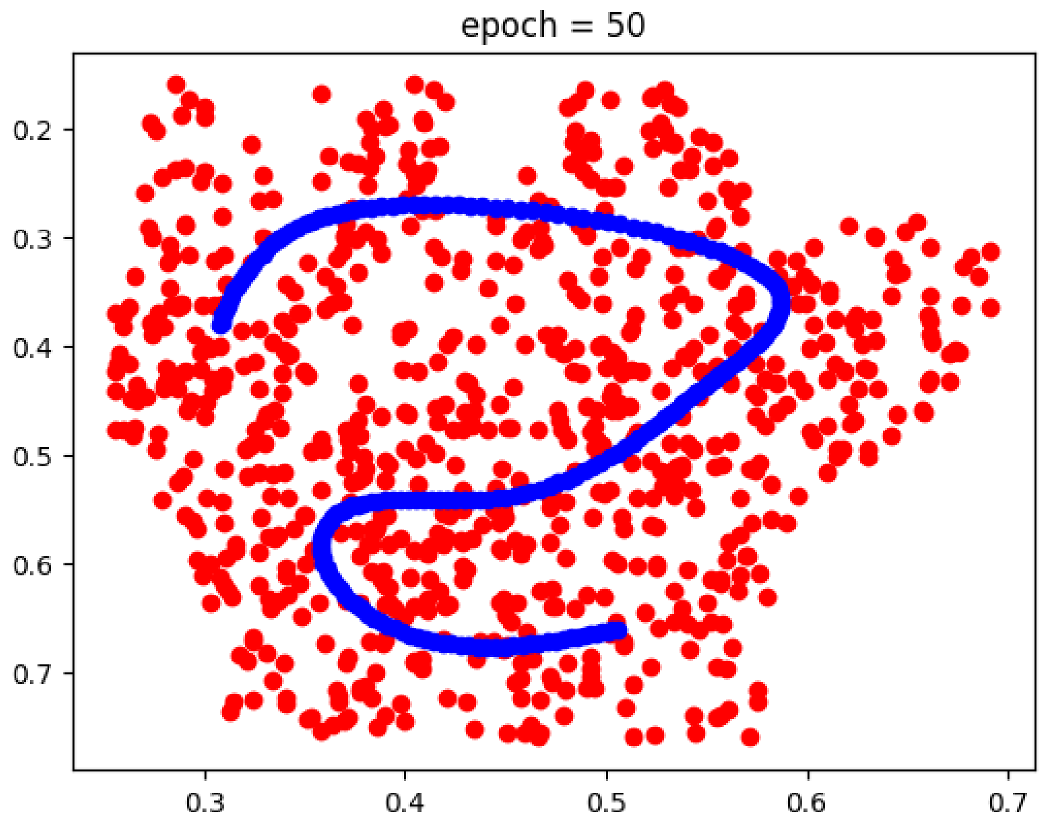

epoch = 200

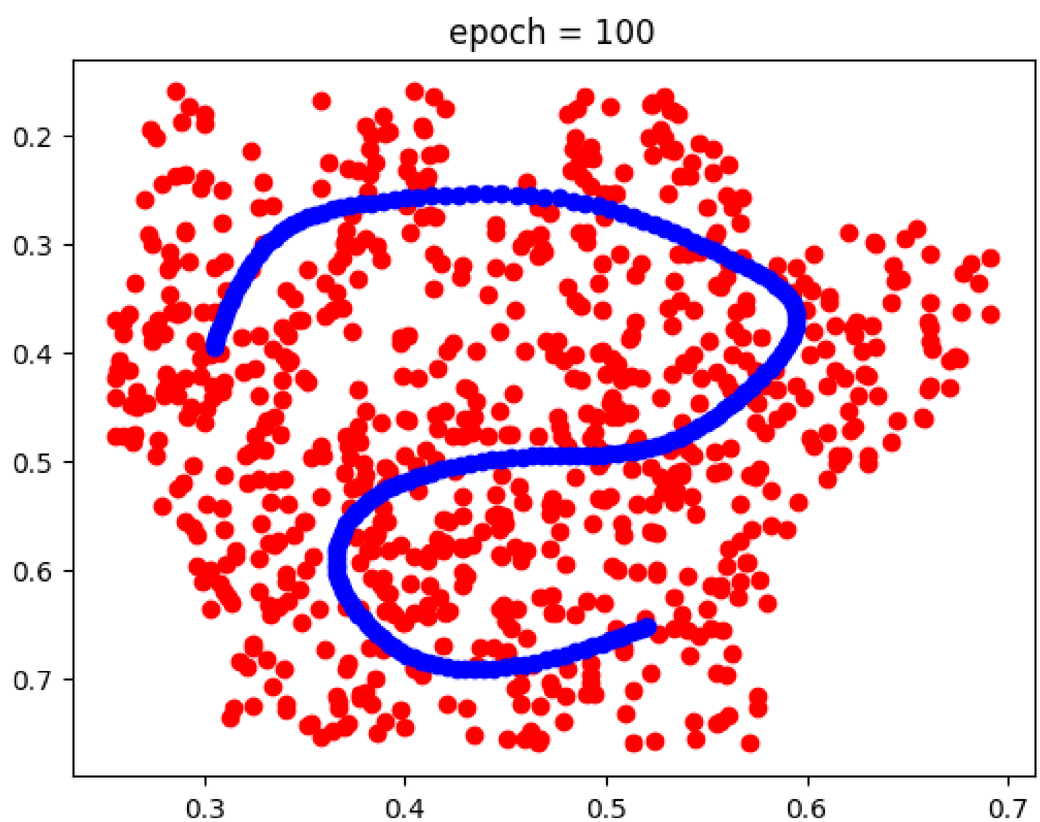

epoch = 250

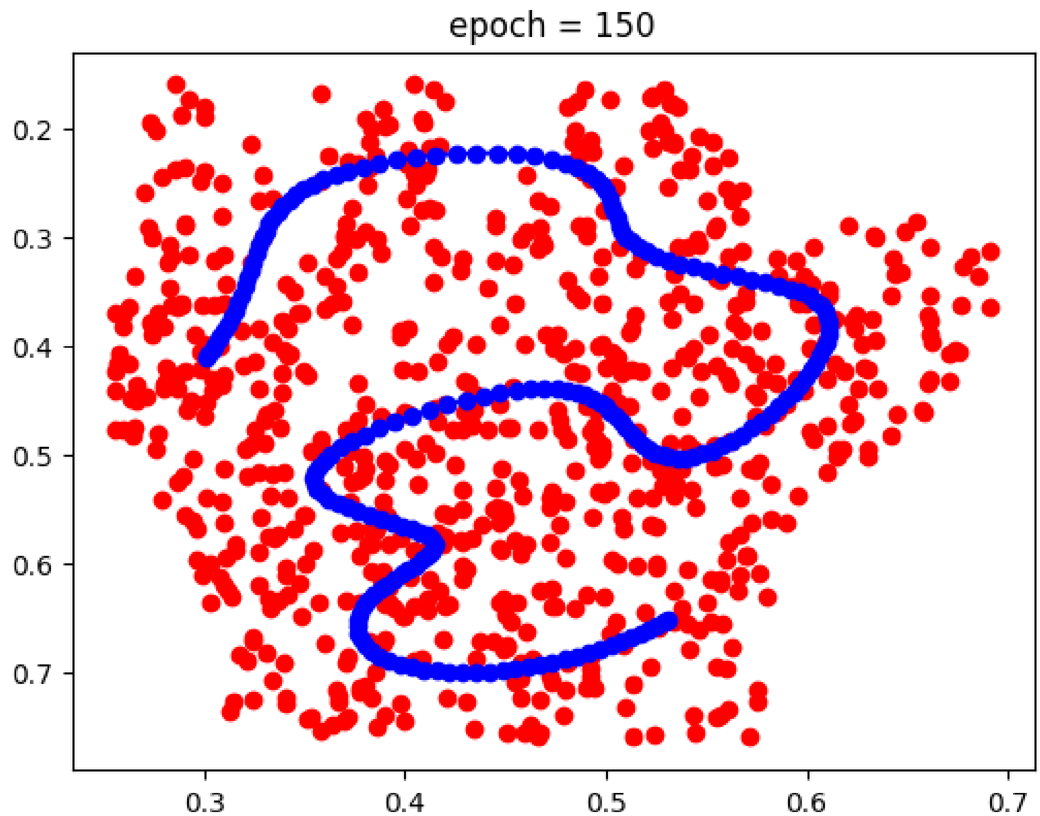
Part B

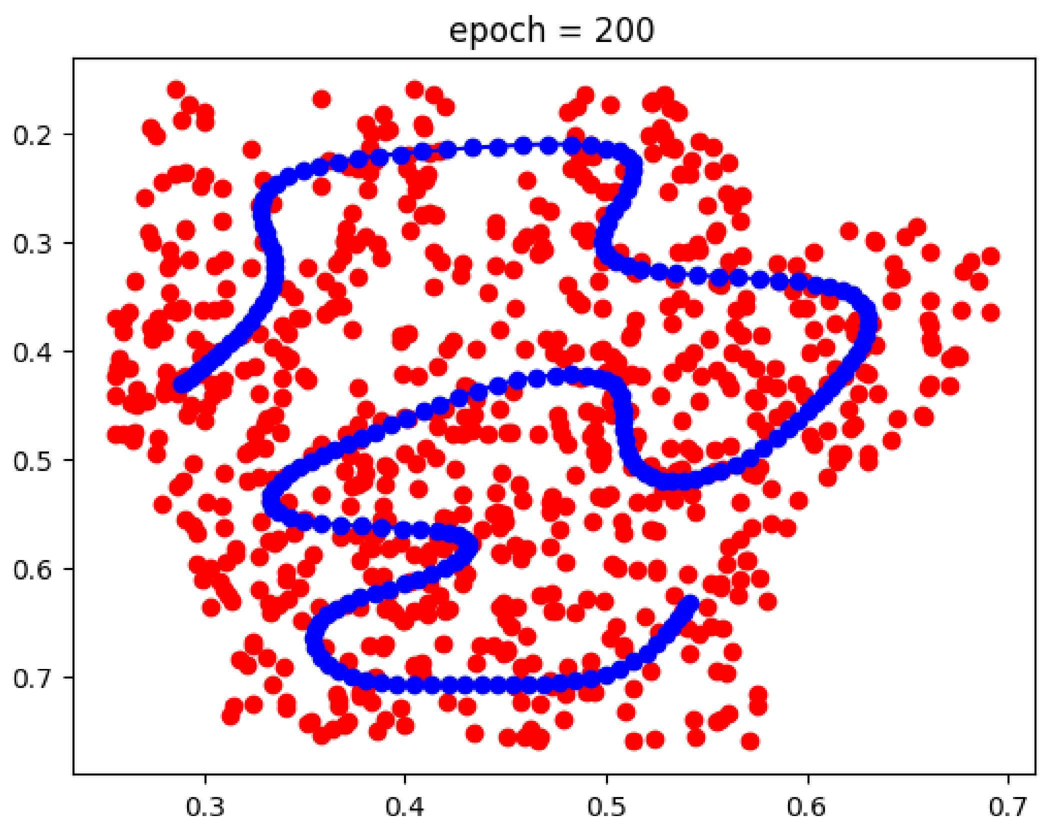
```
num_neurons = 200
neurons = makeNeuronsByDist('hand', num_neurons)
P = dict(enumerate(hand,0))
N = dict(enumerate(neurons,0))

# find best variables
epoches = 300
learning_rate = 0.2
neighborhood_size = 15
neighborhood_radius = len(N)/2
train(P,N , epoches, learning_rate, neighborhood_size, neighborhood_radius)
```

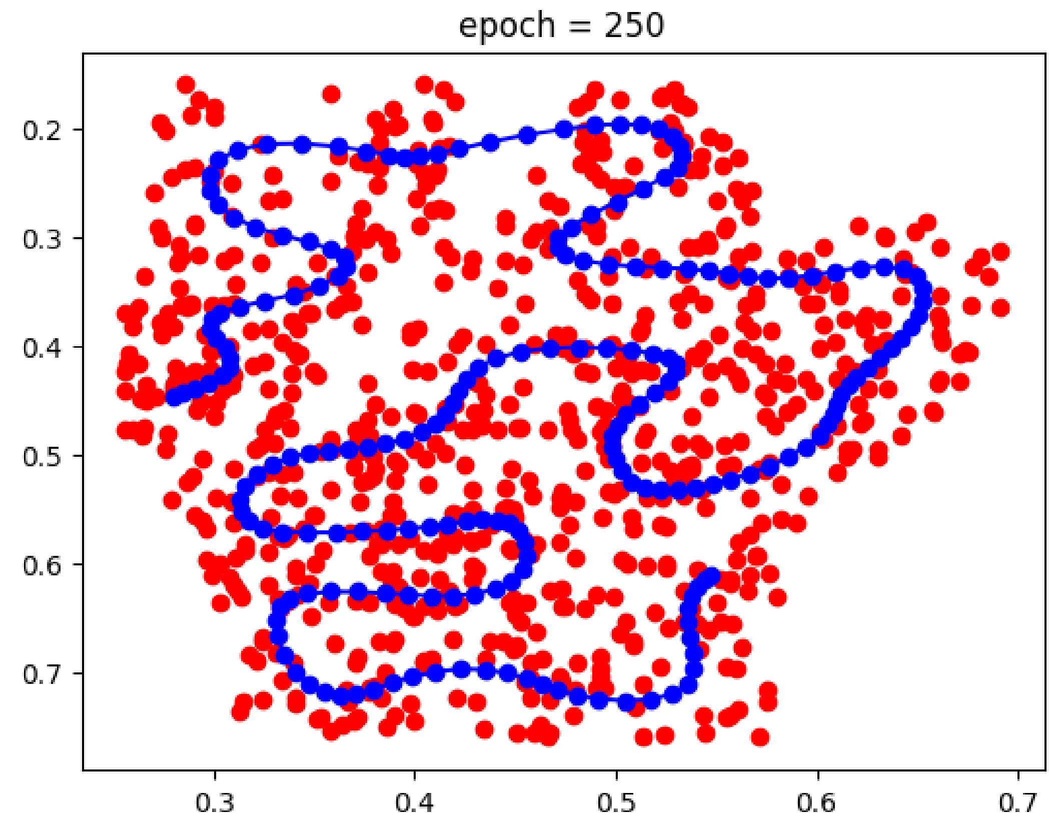



epoch = 50
0.2
0.3
0.4
0.5
0.6
0.7
0.3
0.4
0.5
0.6
0.7


epoch = 100
0.2
0.3
0.4
0.5
0.6
0.7
0.3
0.4
0.5
0.6
0.7

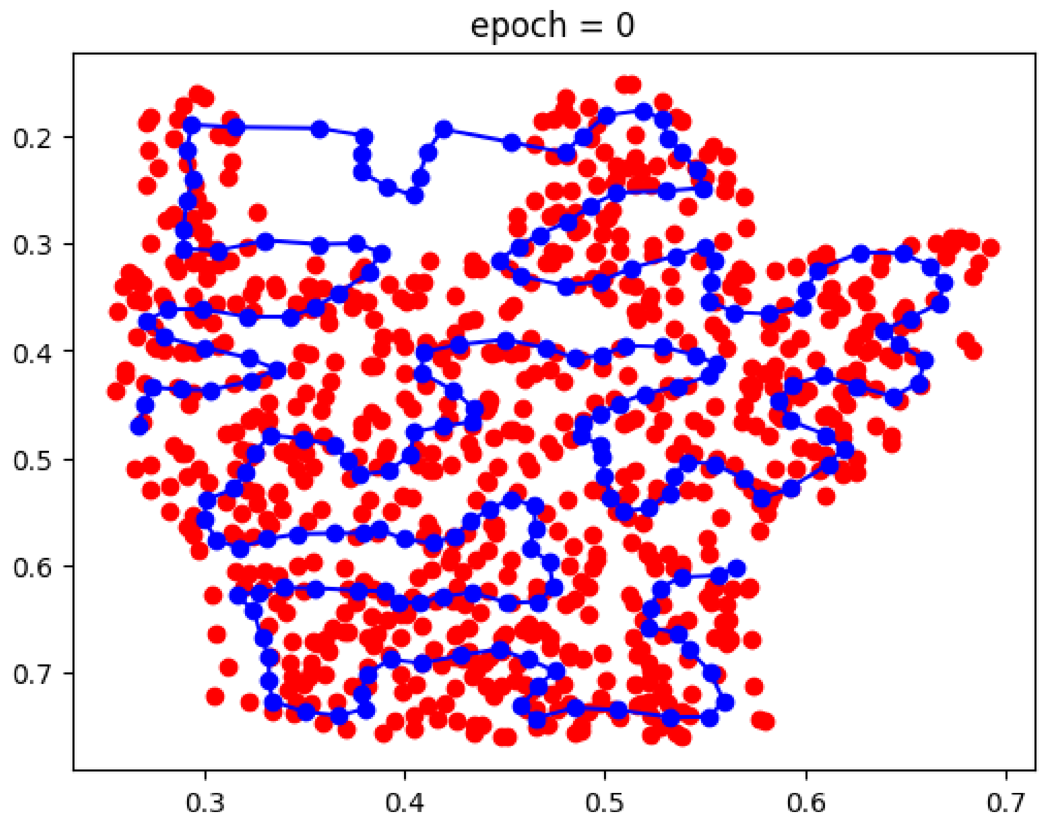

epoch = 150
0.2
0.3
0.4
0.5
0.6
0.7
0.3
0.4
0.5
0.6
0.7


epoch = 200
0.2
0.3
0.4
0.5
0.6
0.7
0.3
0.4
0.5
0.6
0.7

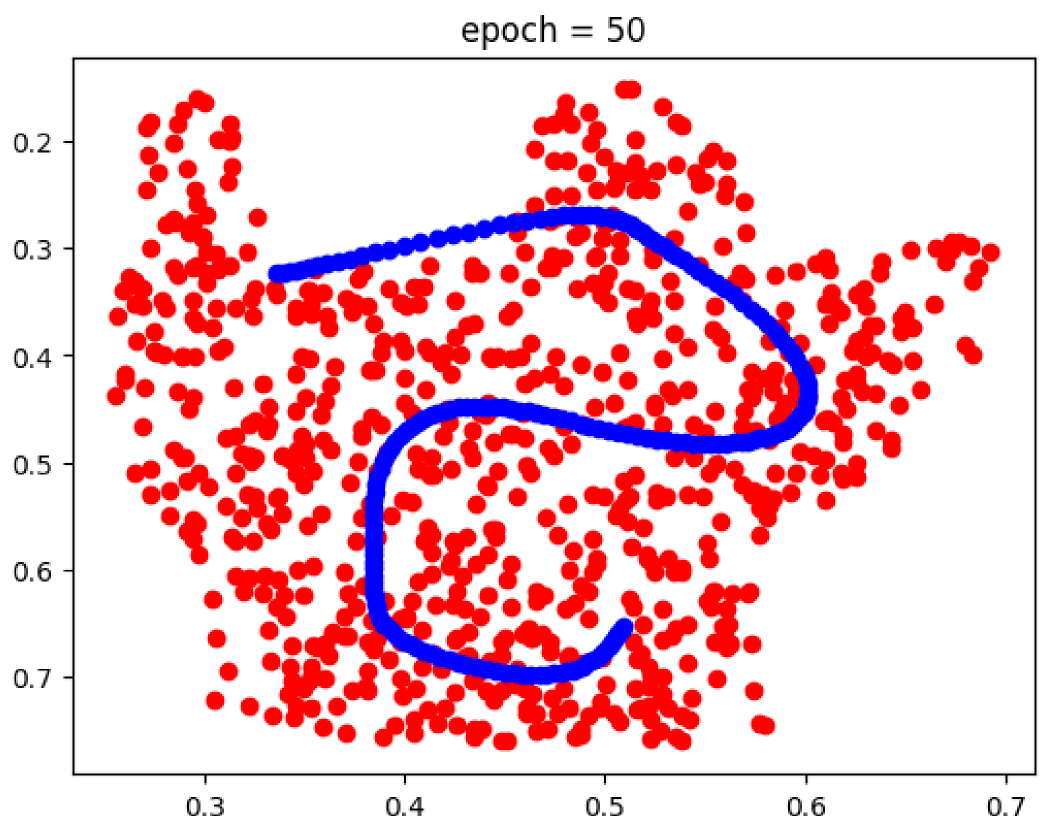


Part B.2

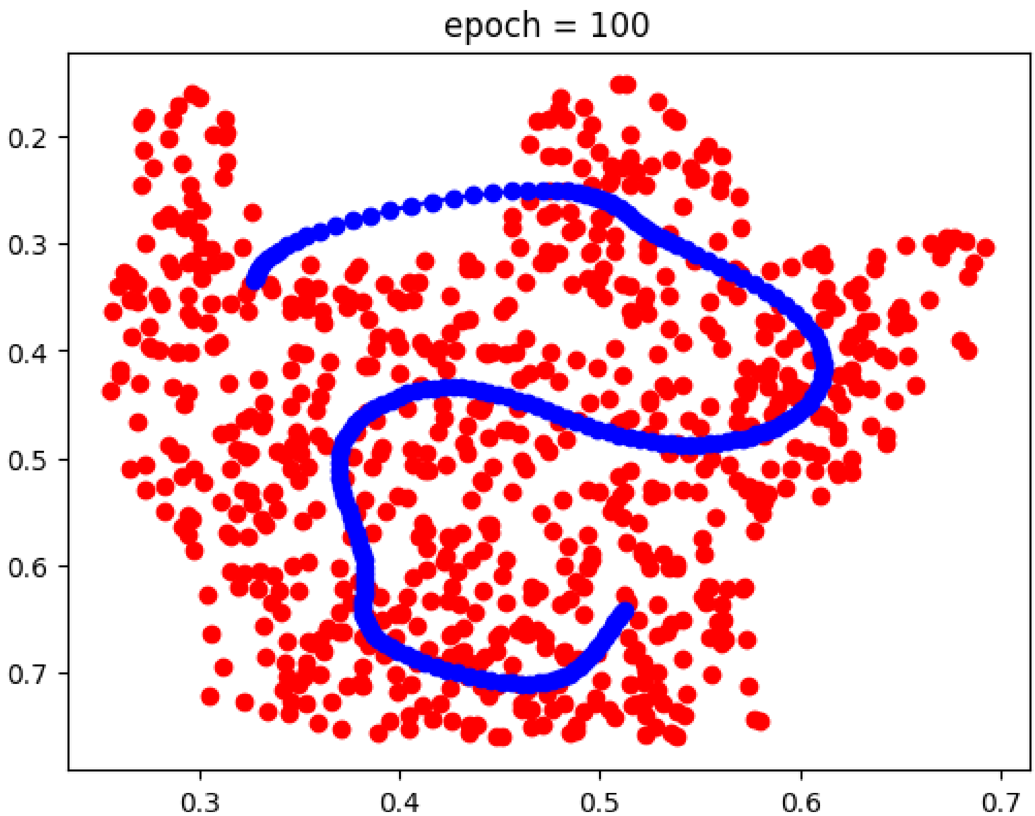
```
P = dict(enumerate(no_finger_hand,0))
N = dict(enumerate(neurons,0))

# find best variables
epoches = 300
learning_rate = 0.2
neighborhood_size = 15
neighborhood_radius = len(N)/2
train(P,N , epoches, learning_rate, neighborhood_size, neighborhood_radius)
```

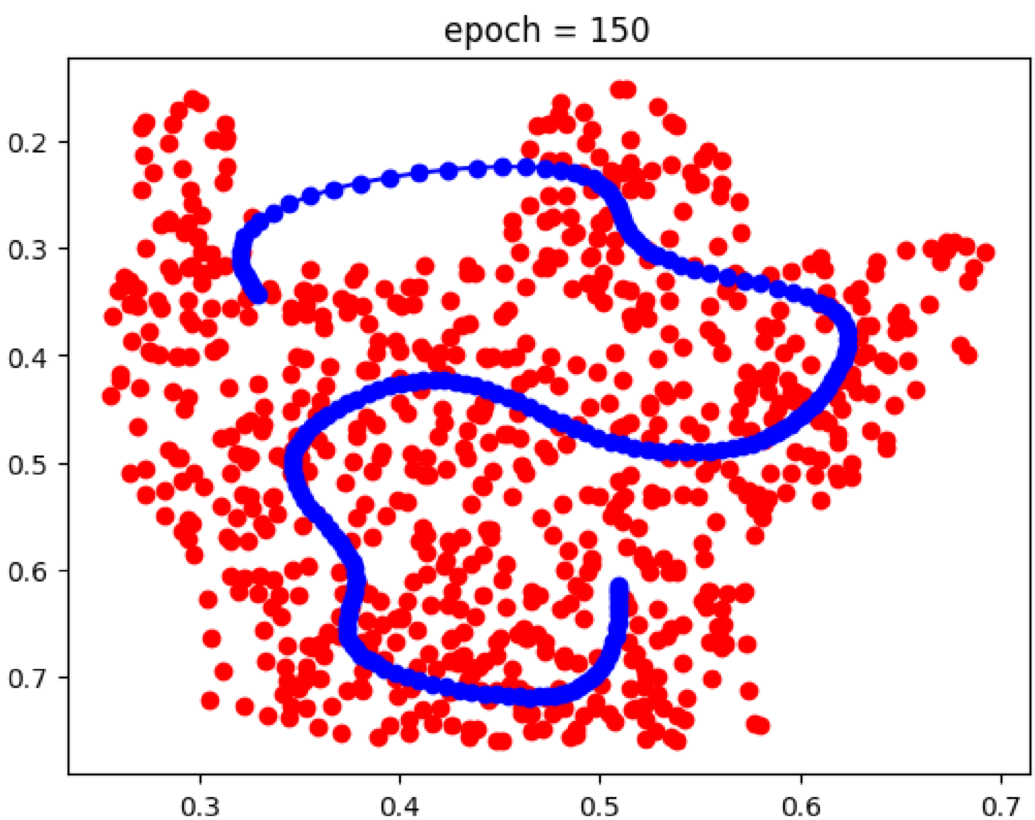


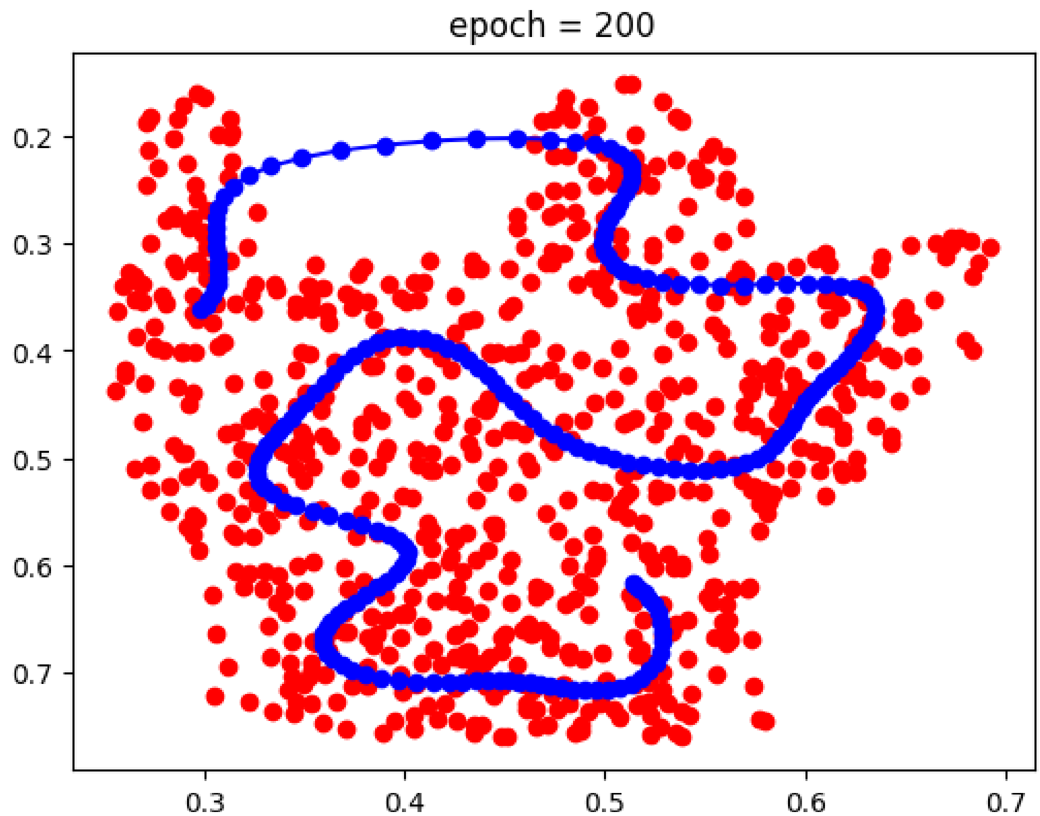
epoch = 0
0.2
0.3
0.4
0.5
0.6
0.7
0.3
0.4
0.5
0.6
0.7

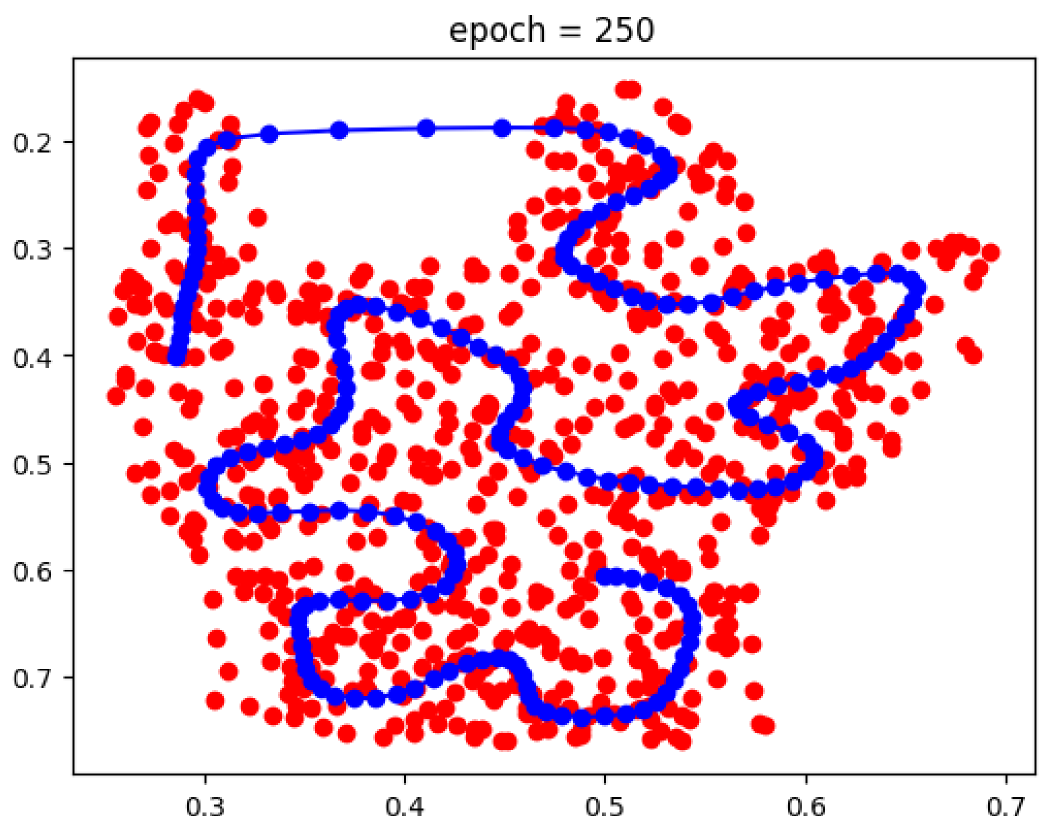


epoch = 50
0.2
0.3
0.4
0.5
0.6
0.7
0.3
0.4
0.5
0.6
0.7


epoch = 100
0.2
0.3
0.4
0.5
0.6
0.7
0.3
0.4
0.5
0.6
0.7


epoch = 150
0.2
0.3
0.4
0.5
0.6
0.7
0.3
0.4
0.5
0.6
0.7


epoch = 200
0.2
0.3
0.4
0.5
0.6
0.7
0.3
0.4
0.5
0.6
0.7


epoch = 250
0.2
0.3
0.4
0.5
0.6
0.7
0.3
0.4
0.5
0.6
0.7