Lab 4 STRUCT Self-Organized Maps. Group 20: Sergio Serrano Hernández-Jaime 2 and Nicolas Schweizer Linsuain 3.

```
import numpy as np
import matplotlib
%matplotlib inline
from matplotlib import pyplot as plt
import time
import sys
from math import floor, ceil
```

```
def getNeighborhood(coordinate,radius,dim):
    neighborhood = np.zeros((2*radius+1, 2*radius+1), dtype=float)
    coordinate = np.array(coordinate)

    mins = (coordinate - (radius,)*2)*(-1)
    mins = (mins, (0,0))
    mins = np.amax(mins,
                   axis=0)
    maxs = (coordinate + (radius,)*2)
    maxs = (maxs, (dim-1,dim-1))
    maxs = np.amin(maxs,
                   axis=0)

    return neighborhood[mins[0]:,mins[1]:]
    [:maxs[0]+1,:maxs[1]+1]
    ,mins,maxs
    #print(getNeighborhood((0,0),1,3))
    #print(getNeighborhood((1,1),1,3))
    #print(getNeighborhood((2,2),1,3))
    #print(getNeighborhood((0,2),1,3))
    #print(getNeighborhood((2,0),1,3))
    #print(getNeighborhood((1,2),1,3))
    #print(getNeighborhood((2,1),1,3))
    #print(getNeighborhood((0,1),1,3))
    #print(getNeighborhood((1,0),1,3))
    #print(getNeighborhood((1,1),2,3))
    #print(getNeighborhood((2,2),2,3))
    #print(getNeighborhood((0,0),2,3))
    #print(getNeighborhood((0,2),2,3))
    #print(getNeighborhood((2,0),2,3))
    #print(getNeighborhood((1,2),2,3))
    #print(getNeighborhood((2,1),2,3))
    #print(getNeighborhood((0,1),2,3))
    #print(getNeighborhood((1,0),2,3))
```

```
n1 = np.ones((3,3),dtype=int)
n2 = np.ones((5,5),dtype=int)
n3 = np.ones((7,7),dtype=int)

print(n1,n2,n3)
```

```
[[1 1 1]
 [1 1 1]
 [1 1 1]] [[1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]] [[1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]]
```

```
n = np.ones((3,3),dtype=int)
n[1,1] = 0
print(n)
```

```
[[1 1 1]
 [1 0 1]
 [1 1 1]]
```

```
n = np.ones((5,5),dtype=int)
n[2,2] = 0
print(n)
```

```
[[1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 0 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]]
```

```
n = np.ones((7,7),dtype=int)
n[3,3] = 0
print(n)
```

```
[[1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]
 [1 1 1 0 1 1 1]
 [1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1]]
```

```
def printing_map(weights, train, train_labels,dimension):
    #Printing the function for plotting out of the training map
    for i in range(0,dimension):
        for j in range(0,dimension):
            if i % 2 == 0:
                print("%d,%d:",%(i,j),
                      np.linalg.norm(train-
                      weights[i,j],axis=1),
                      train_labels[np.argmin(
                      np.linalg.norm(train-
                      weights[i,j],axis=1)
                      ,axis=0)],sep=" ")
            else:
                print("%d,%d:",%(i,j),
                      np.linalg.norm(train-
                      weights[i,j],axis=1),
                      train_labels[np.argmin(
                      np.linalg.norm(train-
                      weights[i,j],axis=1)
                      ,axis=0)],sep="  ")
    print("\n")
```

```
def TSP(weights, trainingset,epochs,sigmaini,sigmafin,alpha,N,dim,sigmaprime,M,dimdim,M_train,
        M_train_labels,trainingset_labels):
    #Function for training the SOM
    time0 = time.
    time1 = time.
    time2 = time.
    time3 = time.
    for epoch in 0:
        #Initialization of the time variables
        if epoch == 0:
            print("Training the SOM for %d epochs",%(epochs),sep="")
            print("Initial weights:",weights,sep="\n")
            time0 = time.time()
        #Initialization of the learning rate and the neighborhood radius
        sigma = sigmaini*(sigmafin/sigmaini)**(epoch/epochs)
        alpha = alpha*(0.01/alpha)**(epoch/epochs)
        #Initialization of the distance matrix
        distance = np.zeros((dim, dim),dtype=float)
        #Initialization of the neighborhood matrix
        neighborhood = np.zeros((dim, dim),dtype=int)
        #Initialization of the variable that will contain the coordinates of the BMU
        BMU = (-1,-1,-1)
        #Initialization of the variable that will contain the quantization error
        qerror = 0
        #Loop for each sample in the training set
        for i in range(0,N):
            #Calculating the distance between the sample and the weights
            distance = np.linalg.norm(trainingset[i]-weights,axis=2)
            #Calculating the coordinates of the BMU
            BMU = (np.argmin(distance,axis=0),np.amin(distance,axis=0),i)
            #Adding the quantization error to the current one
            qerror = qerror + BMU[1]
            #Calculating the neighborhood matrix
            neighborhood = getNeighborhood(BMU[0],sigma*dim,dim)
            #Updating the weights
            weights = weights + np.transpose(np.transpose(trainingset[i]-weights,
                      axis=(1,0))*(alpha*neighborhood),
                      axis=(1,0))
        #Printing the training error and the learning parameters
        print("Training error for the epoch %d: %f",%(epoch,qerror/N),sep="")
        print("The learning parameters are: sigma=%f, alpha=%f",%(sigma,alpha),sep="")
        #Saving the time of the first epoch
        if epoch == 0:
            time1 = time.time()
        #Saving the time of the last epoch
        if epoch == epochs-1:
            time2 = time.time()
        #Printing the weights of the SOM
        print("Weights:",weights,sep="\n")
        #Printing the time of the epoch
        print("Time of the epoch: %f",%(time.time()-time0),sep="")
        time0 = time.time()
    #Printing the total time
    time3 = time.time()
    print("Total time: %f",%(time3-time1),sep="")
    #Printing the map
    printing_map(weights, train, train_labels,dimension)
    #Printing the test samples and their BMUs
    print("Test samples and their BMUs:",sep="")
    for i in range(0,M):
        distance = np.linalg.norm(M_train-weights,axis=2)
        BMU = (np.argmin(distance,axis=0),np.amin(distance,axis=0),i)
        print("%d: %d",%(BMU[2],M_train_labels[BMU[2]]),sep="")
    return trained_weights, labels, classes
```

```
trained_weights, labels, classes = TSP(weights,trainingset,epochs,sigmaini,sigmafin,alpha,N,dim,
                                       sigmaprime,M,dimdim,M_train,M_train_labels,trainingset_labels)
```

```
Training the SOM for 1000 epochs
Initial weights:
[[[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]]
Training error for the epoch 0: 0.000000
The learning parameters are: sigma=0.000000, alpha=0.000000
Weights:
[[[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]]
Time of the epoch: 0.000000
Total time: 0.000000
```

Already provided example by Evgeny

```
import numpy as np
#Initialization of the variables, header lines:
N = 1
input_dim = len(train_inputs[0, header:header])
dimdim = 3
classes = 3*dim*dim*dim*N
sigmaini = 0.5*dimdim/2
sigmafin = 0.5*dimdim/10
sigmaprime = 0.5*dimdim/2
alpha = 0.01
epochs = 1000
M = len(test_inputs)
header = 1
train_inputs, labels, classes
```

```
(array([[0.0, 0.0, 0.0, ..., 0.0, 0.0, 0.0],
        [0.0, 0.0, 0.0, ..., 0.0, 0.0, 0.0],
        [0.0, 0.0, 0.0, ..., 0.0, 0.0, 0.0],
        ...,
        [0.0, 0.0, 0.0, ..., 0.0, 0.0, 0.0],
        [0.0, 0.0, 0.0, ..., 0.0, 0.0, 0.0],
        [0.0, 0.0, 0.0, ..., 0.0, 0.0, 0.0]],
       dtype=float64),
array([1, 1, 1, ..., 1, 1, 1], dtype=int64),
27000)
```

```
weights = np.random.rand(dimdim,dimdim,input_dim)
trained_weights, labels, classes = TSP(weights,train_inputs,epochs,sigmaini,sigmafin,alpha,N,dim,
                                       sigmaprime,M,dimdim,test_inputs,test_labels,train_inputs,labels)
```

```
Training the SOM for 1000 epochs
Initial weights:
[[[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]]
Training error for the epoch 0: 0.000000
The learning parameters are: sigma=0.000000, alpha=0.000000
Weights:
[[[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]]
Time of the epoch: 0.000000
Total time: 0.000000
```



```
printing_map(trained_weights, train, train_labels,dimdim)
```



```
#Printing the test samples and their BMUs
print("Test samples and their BMUs:",sep="")
for i in range(0,M):
    distance = np.linalg.norm(trained_weights-test_inputs[i],axis=2)
    BMU = (np.argmin(distance,axis=0),np.amin(distance,axis=0),i)
    print("%d: %d",%(BMU[2],test_labels[BMU[2]]),sep="")
```

```
Test samples and their BMUs:
0: 0
1: 0
2: 0
3: 0
4: 0
5: 0
6: 0
7: 0
8: 0
9: 0
10: 0
11: 0
12: 0
13: 0
14: 0
15: 0
16: 0
17: 0
18: 0
19: 0
20: 0
21: 0
22: 0
23: 0
24: 0
25: 0
26: 0
27: 0
28: 0
29: 0
30: 0
31: 0
32: 0
33: 0
34: 0
35: 0
36: 0
37: 0
38: 0
39: 0
40: 0
41: 0
42: 0
43: 0
44: 0
45: 0
46: 0
47: 0
48: 0
49: 0
50: 0
51: 0
52: 0
53: 0
54: 0
55: 0
56: 0
57: 0
58: 0
59: 0
60: 0
61: 0
62: 0
63: 0
64: 0
65: 0
66: 0
67: 0
68: 0
69: 0
70: 0
71: 0
72: 0
73: 0
74: 0
75: 0
76: 0
77: 0
78: 0
79: 0
80: 0
81: 0
82: 0
83: 0
84: 0
85: 0
86: 0
87: 0
88: 0
89: 0
90: 0
91: 0
92: 0
93: 0
94: 0
95: 0
96: 0
97: 0
98: 0
99: 0
```

Tasks 1 and 2

Working with the MNIST Dataset

```
train_inputs, train_labels,test_inputs,test_labels=
np.load("/content/drive/MyDrive/mnist.npz",allow_pickle=True)
print(train_inputs,train_labels,test_inputs,test_labels,sep="\n")
N,input_dim=train_inputs.shape,784
```

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

Tasks 3 and 4

Training the SOM with dimensions 30, 40 and 60. We not only bring some light with the heavily reduced dataset which I find interesting, but also we bring some light with the time needed for training the SOM with different dimensions. We can see that the time needed for training grows the magnitude of a single order but the training averages the neurons in different ways that will lead to a better training error. Lower is better and significant help.

```
dimdim = 30
input_dim = 784
N = 10000
sigmaini = 0.5*dimdim/2
sigmafin = 0.5*dimdim/10
sigmaprime = 0.5*dimdim/2
alpha = 0.01
epochs = 1000
M = 10000
header = 0
train_inputs, labels, classes
```

```
(array([[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]], dtype=uint8),
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8),
27000)
```

```
weights = np.random.rand(dimdim,dimdim,input_dim)
trained_weights, labels, classes = TSP(weights,train_inputs,epochs,sigmaini,sigmafin,alpha,N,dim,
                                       sigmaprime,M,dimdim,test_inputs,test_labels,train_inputs,labels)
```

```
Training the SOM for 1000 epochs
Initial weights:
[[[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]]
Training error for the epoch 0: 0.000000
The learning parameters are: sigma=0.000000, alpha=0.000000
Weights:
[[[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]
 [[0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]
  [0.0 0.0 0.0]]]
Time of the epoch: 0.000000
Total time: 0.000000
```

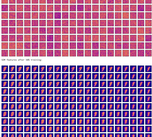
```
printing_map(trained_weights, train, train_labels,dimdim)
```



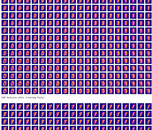
```
#Printing the test samples and their BMUs
print("Test samples and their BMUs:",sep="")
for i in range(0,M):
    distance = np.linalg.norm(trained_weights-test_inputs[i],axis=2)
    BMU = (np.argmin(distance,axis=0),np.amin(distance,axis=0),i)
    print("%d: %d",%(BMU[2],test_labels[BMU[2]]),sep="")
```

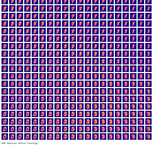
```
Test samples and their BMUs:
0: 0
1: 0
2: 0
3: 0
4: 0
5: 0
6: 0
7: 0
8: 0
9: 0
10: 0
11: 0
12: 0
13: 0
14: 0
15: 0
16: 0
17: 0
18: 0
19: 0
20: 0
21: 0
22: 0
23: 0
24: 0
25: 0
26: 0
27: 0
28: 0
29: 0
30: 0
31: 0
32: 0
33: 0
34: 0
35: 0
36: 0
37: 0
38: 0
39: 0
40: 0
41: 0
42: 0
43: 0
44: 0
45: 0
46: 0
47: 0
48: 0
49: 0
50: 0
51: 0
52: 0
53: 0
54: 0
55: 0
56: 0
57: 0
58: 0
59: 0
60: 0
61: 0
62: 0
63: 0
64: 0
65: 0
66: 0
67: 0
68: 0
69: 0
70: 0
71: 0
72: 0
73: 0
74: 0
75: 0
76: 0
77: 0
78: 0
79: 0
80: 0
81: 0
82: 0
83: 0
84: 0
85: 0
86: 0
87: 0
88: 0
89: 0
90: 0
91: 0
92: 0
93: 0
94: 0
95: 0
96: 0
97: 0
98: 0
99: 0
```

MR Features before training:

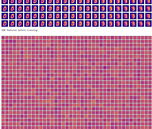


MR Features after training:

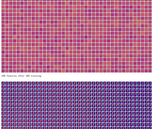


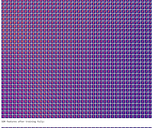


MR Features before training Test:

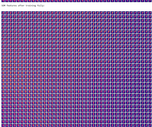


MR Features after training Test:



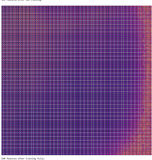


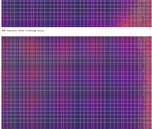
MR Features before training Test:



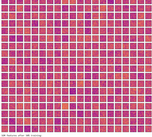
MR Features after training Test:

MR Features after MR training:

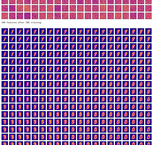


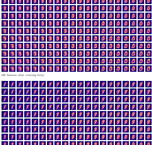


MR Features after training Test:



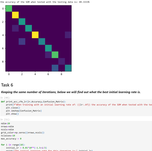
MR Features after MR training:





MR Features after training Test:

MR Features after MR training:



MR Features after training Test:

Task 5

Printing the confusion matrices for both the training dataset and the testing dataset

Evaluating on the training dataset

```
TRAIN:
Accuracy:
0.9999999999999999
precision recall f1-score support
accuracy 1.000 1.000 1.000 1.000
macro avg 1.000 1.000 1.000 1.000
weighted avg 1.000 1.000 1.000 1.000
Confusion Matrix:
[[1000 0 0 0 0 0 0 0 0 0]
[ 0 1000 0 0 0 0 0 0 0 0]
[ 0 0 1000 0 0 0 0 0 0 0]
[ 0 0 0 1000 0 0 0 0 0 0]
[ 0 0 0 0 1000 0 0 0 0 0]
[ 0 0 0 0 0 1000 0 0 0 0]
[ 0 0 0 0 0 0 1000 0 0 0]
[ 0 0 0 0 0 0 0 1000 0 0]
[ 0 0 0 0 0 0 0 0 1000 0]
[ 0 0 0 0 0 0 0 0 0 1000]]
```

Evaluating on the testing dataset

```
TEST:
Accuracy:
0.9999999999999999
precision recall f1-score support
accuracy 1.000 1.000 1.000 1.000
macro avg 1.000 1.000 1.000 1.000
weighted avg 1.000 1.000 1.000 1.000
Confusion Matrix:
[[1000 0 0 0 0 0 0 0 0 0]
[ 0 1000 0 0 0 0 0 0 0 0]
[ 0 0 1000 0 0 0 0 0 0 0]
[ 0 0 0 1000 0 0 0 0 0 0]
[ 0 0 0 0 1000 0 0 0 0 0]
[ 0 0 0 0 0 1000 0 0 0 0]
[ 0 0 0 0 0 0 1000 0 0 0]
[ 0 0 0 0 0 0 0 1000 0 0]
[ 0 0 0 0 0 0 0 0 1000 0]
[ 0 0 0 0 0 0 0 0 0 1000]]
```

```
Confusion Matrix for the training dataset:
[[1000 0 0 0 0 0 0 0 0 0]
[ 0 1000 0 0 0 0 0 0 0 0]
[ 0 0 1000 0 0 0 0 0 0 0]
[ 0 0 0 1000 0 0 0 0 0 0]
[ 0 0 0 0 1000 0 0 0 0 0]
[ 0 0 0 0 0 1000 0 0 0 0]
[ 0 0 0 0 0 0 1000 0 0 0]
[ 0 0 0 0 0 0 0 1000 0 0]
[ 0 0 0 0 0 0 0 0 1000 0]
[ 0 0 0 0 0 0 0 0 0 1000]]
```



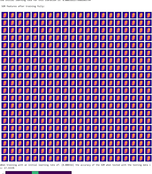
```
Confusion Matrix for the testing dataset:
[[1000 0 0 0 0 0 0 0 0 0]
[ 0 1000 0 0 0 0 0 0 0 0]
[ 0 0 1000 0 0 0 0 0 0 0]
[ 0 0 0 1000 0 0 0 0 0 0]
[ 0 0 0 0 1000 0 0 0 0 0]
[ 0 0 0 0 0 1000 0 0 0 0]
[ 0 0 0 0 0 0 1000 0 0 0]
[ 0 0 0 0 0 0 0 1000 0 0]
[ 0 0 0 0 0 0 0 0 1000 0]
[ 0 0 0 0 0 0 0 0 0 1000]]
```


Task 6

Keeping the same number of iterations, below we will find out what the best initial learning rate is.

```
MR_LEARN_RATE_LIST=[0.01,0.0001,0.00001,0.000001]
for MR_LEARN_RATE in MR_LEARN_RATE_LIST:
    print("Training with initial learning rate = {}".format(MR_LEARN_RATE))
    MR_LEARNED_CLASSIFIER_MATRIX=
    MR_LEARN_RATE=MR_LEARN_RATE
    MR_LEARN_ITERATIONS=100
    MR_LEARNED_CLASSIFIER_MATRIX=
    MR_LEARN_RATE=MR_LEARN_RATE
    MR_LEARN_ITERATIONS=100
    MR_LEARNED_CLASSIFIER_MATRIX=
    MR_LEARNED_CLASSIFIER_MATRIX=
    for i in range(10):
        print("Training digit {}".format(i))
        print("Training with initial learning rate = {}".format(MR_LEARN_RATE))
        MR_LEARNED_CLASSIFIER_MATRIX[i,:]=GradientDescent(MR_TRAIN_MATRIX,MR_TRAIN_LABELS,MR_LEARN_RATE,MR_LEARN_ITERATIONS,
        print("Training digit {}".format(i))
        print("Training with initial learning rate = {}".format(MR_LEARN_RATE))
        MR_LEARNED_CLASSIFIER_MATRIX[i,:]=GradientDescent(MR_TRAIN_MATRIX,MR_TRAIN_LABELS,MR_LEARN_RATE,MR_LEARN_ITERATIONS,
        print("Training digit {}".format(i))
        print("Training with initial learning rate = {}".format(MR_LEARN_RATE))
        MR_LEARNED_CLASSIFIER_MATRIX[i,:]=GradientDescent(MR_TRAIN_MATRIX,MR_TRAIN_LABELS,MR_LEARN_RATE,MR_LEARN_ITERATIONS,
        print("Training digit {}".format(i))
        print("Training with initial learning rate = {}".format(MR_LEARN_RATE))
        MR_LEARNED_CLASSIFIER_MATRIX[i,:]=GradientDescent(MR_TRAIN_MATRIX,MR_TRAIN_LABELS,MR_LEARN_RATE,MR_LEARN_ITERATIONS,
        print("Training digit {}".format(i))
        print("Training with initial learning rate = {}".format(MR_LEARN_RATE))
        MR_LEARNED_CLASSIFIER_MATRIX[i,:]=GradientDescent(MR_TRAIN_MATRIX,MR_TRAIN_LABELS,MR_LEARN_RATE,MR_LEARN_ITERATIONS,
        print("Training digit {}".format(i))
        print("Training with initial learning rate = {}".format(MR_LEARN_RATE))
        MR_LEARNED_CLASSIFIER_MATRIX[i,:]=GradientDescent(MR_TRAIN_MATRIX,MR_TRAIN_LABELS,MR_LEARN_RATE,MR_LEARN_ITERATIONS,
        print("Training digit {}".format(i))
        print("Training with initial learning rate = {}".format(MR_LEARN_RATE))
        MR_LEARNED_CLASSIFIER_MATRIX[i,:]=GradientDescent(MR_TRAIN_MATRIX,MR_TRAIN_LABELS,MR_LEARN_RATE,MR_LEARN_ITERATIONS,
        print("Training digit {}".format(i))
        print("Training with initial learning rate = {}".format(MR_LEARN_RATE))
        MR_LEARNED_CLASSIFIER_MATRIX[i,:]=GradientDescent(MR_TRAIN_MATRIX,MR_TRAIN_LABELS,MR_LEARN_RATE,MR_LEARN_ITERATIONS,
        print("Training digit {}".format(i))
        print("Training with initial learning rate = {}".format(MR_LEARN_RATE))
        MR_LEARNED_CLASSIFIER_MATRIX[i,:]=GradientDescent(MR_TRAIN_MATRIX,MR_TRAIN_LABELS,MR_LEARN_RATE,MR_LEARN_ITERATIONS,
```

The initial learning rate for this iteration is: 0.0001111111111111111



The initial learning rate for this iteration is: 0.00011

```
[1,0,0]
[0,1,0]
[0,0,1]
[0,0,0,
```

Table 3 and 5

What are the effect of the previous changes?

- Increasing the batch size (larger training data for SOM) leads to lower quantization error, probably due to the law of number of samples we are using. I believe if we would be using more samples for the training it would also lower quick-well with smaller learning rates. As for the weight initialization, random initialization leads to lower quantization error than PCA initialization, which is unexpected. I believe this is because now we are not starting with a very complex network the algorithm does not need to explore the neighborhood of the best unit that is why it is okay for the SOM to have a big neighborhood at the beginning and quickly reduce the radius of the neighborhood. Regarding the learning rate and the radius, as we know from the theory, the lower the values of these two hyper-parameters (which controls the hyper-bigger neighborhoods) there is a great influence of all the neurons in each other and in the neighborhood, which leads to lower quantization error. Regarding the number of epochs, as we know from the theory, the higher the number, is with any parameter in the, there is a overtrain and training too large of a weight decay also makes the SOM not generalize well and increases the quantization error. Regarding the topology and the neighborhood function, I believe that what happens here underlying chaos to the input vectors will develop very quickly and they will be the best matching unit radius after. Although without the weight influence of any other neurons which will lowered the radius later neurons as well (updating up and all the neurons in the neighborhood will be updated with the same weight vector) the neighborhood function will be very small and the learning rate is lowered and will not substantially enough for a new instance to be properly classified. You can find an unsupervised example in the cell below where I have changed the topology and the neighborhood function.

What is a biological neuron? How does it relate to the concept of neurons in SOM?

Biological neurons are specialized cells that process and transmit information through electrical and chemical signals. The structure and function of biological neurons have inspired the development of artificial neural networks and SOMs in particular. Both biological neurons and artificial neurons in SOMs share the fundamental concept of processing information through interconnected nodes and adjusting connection weights, and transmitting signals. SOMs may be much simpler than more biological neural networks but they capture the essential idea of how neurons can learn and adapt to process information in a structured way.

```
weight_decay_neighborhood = 30
model.fit(X_train, epochs=10, batch_size=32, verbose=1)# weight_decay_neighborhood=
map_training_MNIST = SOM(20,20,784,sigma=3, learning_rate=0.5, neighborhood_function='gaussian', random_seed=42, weight_decay_neighborhood=
map_training_MNIST.random_weights_init(X_train)
map_training_MNIST.train_random(X_train, 10000, verbose=True)# train_pca_idx, lr=0.5, sigma=3.0)
plot_u_matrix(map_training_MNIST, X_train, X_train[:5000], y_train[:5000])
```

```
Epoch 1/10
313/313 [==============================] - 1s 2ms/step - loss: 0.2304 - accuracy: 0.9293
Epoch 2/10
313/313 [==============================] - 1s 2ms/step - loss: 0.1000 - accuracy: 0.9700
Epoch 3/10
313/313 [==============================] - 1s 2ms/step - loss: 0.0700 - accuracy: 0.9787
Epoch 4/10
313/313 [==============================] - 1s 2ms/step - loss: 0.0543 - accuracy: 0.9833
Epoch 5/10
313/313 [==============================] - 1s 2ms/step - loss: 0.0444 - accuracy: 0.9863
Epoch 6/10
313/313 [==============================] - 1s 2ms/step - loss: 0.0374 - accuracy: 0.9883
Epoch 7/10
313/313 [==============================] - 1s 2ms/step - loss: 0.0321 - accuracy: 0.9900
Epoch 8/10
313/313 [==============================] - 1s 2ms/step - loss: 0.0280 - accuracy: 0.9912
Epoch 9/10
313/313 [==============================] - 1s 2ms/step - loss: 0.0246 - accuracy: 0.9923
Epoch 10/10
313/313 [==============================] - 1s 2ms/step - loss: 0.0217 - accuracy: 0.9932
[ 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000]
```