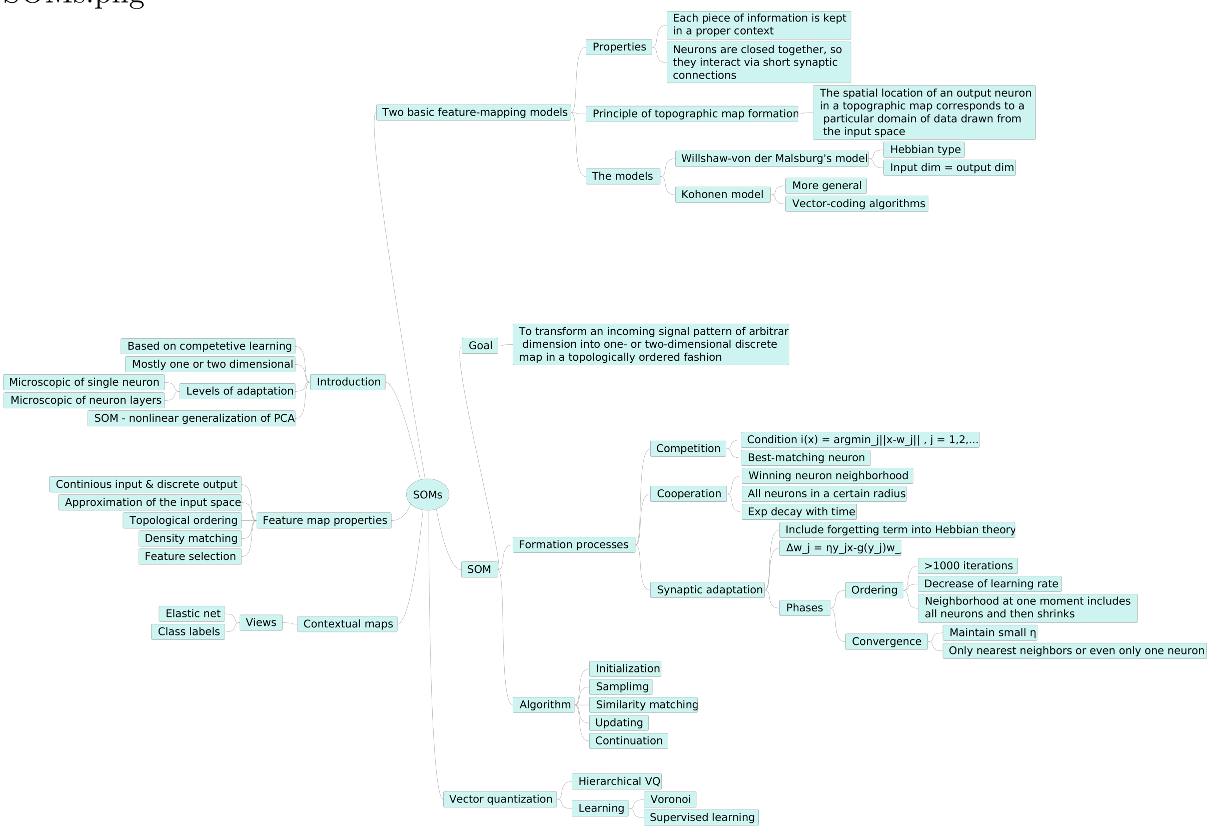
Neural Networks
- Homework 9 -

Petr Lukin, Evgeniya Ovchinnikova

Lecture date: 21 November 2016

1 Mind map

Figure 1: Mind map. Chapter 9 from Haykin's book. A zoomed version is attached as SOMs.png



2 Exercises

2.1 Exercise 2

Show that in the SOM algorithm the winner neuron for an input x is the neuron k whose weight vector w_k maximizes the inner product $< wk, x >$ of x and w k , with x and w_k normalized.

2.2 Exercise 3

Consider the one dimensional input space S=0.1, 0.2, 0.4, 0.5. Cluster S using a one dimensional SOM network with:

- 2 nodes.
- learning rate equal to 0.1.
- Neighborhood function which is equal to 1 for the winner neuron and is 0 otherwise.
- Weight initialization:
  - w_1=0.15, w_2=0.45
  - w_1=0.3, w_2=0.9
- Stopping criterion:

$$\sum_{i=1}^{2} |w_i^{old} - w_i^{new}| < 0.01$$

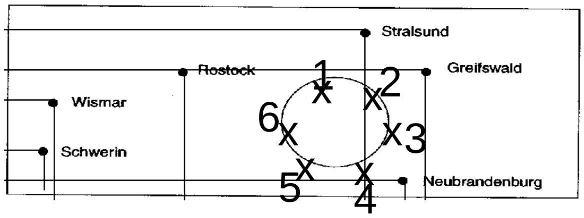
Comment on the two clusterings you obtained using the two different weight initializations.

2.3 Exercise 4

Program a SOM to solve the traveling salesman problem (TSP):

- Input layer contains just two neurons called "Xcoord" and "Ycoord"
- The Kohonen lattice is thought of as a circle containing at least as many neurons as we have cities (usage of more neurons is possible), the neurons are numbered in round about fashion.
- Each neuron n(i) (numbered by i) on the circle is connected to "Xcoord" and "Ycoord" and the weights on these edges are the initial x and y coordinates of the neuron in the plane (see picture)
- The input patterns are the (x,y) coordinates of all target cities.
- When applying one concrete city pattern the winner is of course the neuron with lies closest to the given city. Only the weights of the winning neuron will be adapted according to the learning rule, no other neighboring weights are changed. This "moves" the neuron closer to "its" city.
- After some cycles there is just ONE neuron closest to each city. These build pairs (n(i),City(closest to n(i))).
- If we sort this pairs according to the number of the neurons "i" this will give a journey for the respective cities, which solves the TSP approximately.
- Use tools like ICONNECT, TSPLIB or alike to do the programming, compare runtime, memory and achieved length of path.

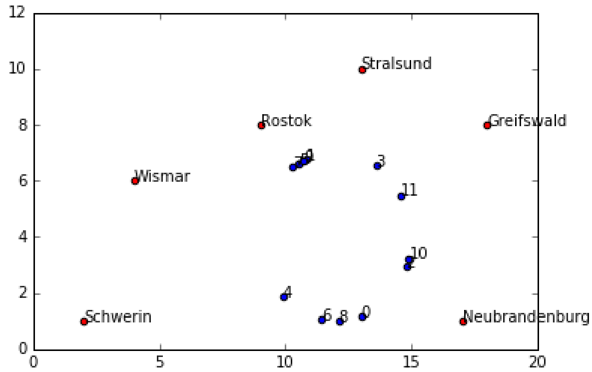
Figure 2: Cities for TSP.



Solution:

First we plot the cities and initial neuron weights (Fig. 3). One can see that here we have 12 -more than 6 hidden neurons. The reason is that with 6 and even 10 neurons we ended up in a local minimum - with one neuron stuck between two cities.

Figure 3: Cities on coordinate plane.



To initialize the weight and create those points we used the following code:

```
import numpy as np
import matplotlib.pyplot as plt
import math
from numpy import linalg as LA
%matplotlib inline

cities_names = np.array(["Schwerin", "Wismar", "Rostok", "
    Stralsund", "Greifswald", "Neubrandenburg"])
cities_coordinates = np.array
    ([[2,1],[4,6],[9,8],[13,10],[18,8],[17,1]])

leng = 12
neuron_numbers = np.arange(leng)
circle_x = np.random.randint(90, 150, leng)
circle_x = circle_x/10.0
circle_y = np.array([])
for i in range(leng):
    sign = 1
    if (i%2 == 0):
        sign = -1
```

```
    y = sign*math.sqrt(9 - (circle_x[i]-12)**2) + 4
    circle_y = np.append(circle_y, y)

initial_weights = np.column_stack((circle_x, circle_y))
weights = initial_weights

def plot(cities_coordinates, weights):
    fig, ax = plt.subplots()
    ax.scatter(cities_coordinates[:,0], cities_coordinates[:,1],
        c = 'r')
    ax.scatter(weights[:,0], weights[:,1], c = 'b')

    #ax.annotate(cities_names[0], xy=cities_coordinates[],
        xytext=(3, 1.5))

    for i in range(len(cities_names)):
        ax.annotate(cities_names[i], (cities_coordinates[i,0],
            cities_coordinates[i,1]))
    for i in range(len(neuron_numbers)):
        ax.annotate(neuron_numbers[i], (weights[i,0],weights[i
            ,1]))

    plt.show()
plot(cities_coordinates, weights)
```

Next part of the program implements the learning. In a cycle over all cities we check the distances between the city and neurons and move the closest neuron to the city according to the following formula:
$w_j(n+1) = w_j(n)_\eta(n)h_{j,i(x)}(n)(x(n) - w_j(n))$, where η is a learning rate, $h_{j,i(x)}$ is a neighboring function, which is not really important in our case since we are required to use only one neuron, and x - coordinates of a city of our concern. We repeat it until all cities have neurons in a distance less than 0.0001 from them.

```
change = 100
step = 0
summ = 100
indicies = np.array([])

while(abs(summ) > 0.0001):
    summ = 0
    indicies = np.array([])
    for city in cities_coordinates:
        #Determine the winning neuron
        dists = np.array([])
        for weight in weights:
            dist = math.hypot(city[0]-weight[0], city[1]-weight
                [1])
            dists = np.append(dists, dist)
        winner_index = np.argmin(dists)
```

```
        indicies = np.append(indicies, winner_index)
        #Adapt the weights of the winner neuron:
        weights[winner_index] = weights[winner_index] + eta(step
            )*(city - weights[winner_index])
        change = LA.norm(eta(step)*(city - weights[winner_index
            ]))
        summ += change
    step += 1
print step
print indicies

>> 231
>> [  2.    5.   11.    1.    7.    0.]
```

The algorithm converged in 231 steps and returned us the indices of neurons those are the closest to the cities. So, now we can plot the neurons with the cities and a path (Fig.5). The path length = 58.71443512.

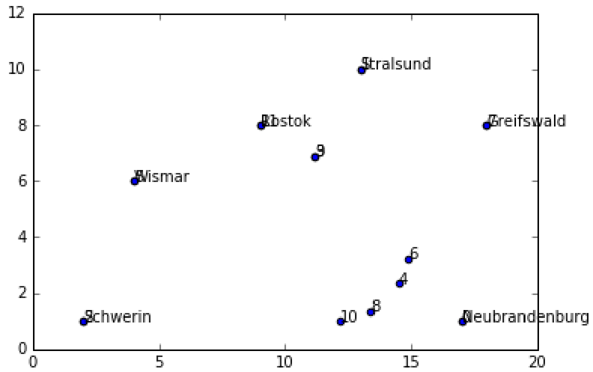


Figure 4: Paired neurons.

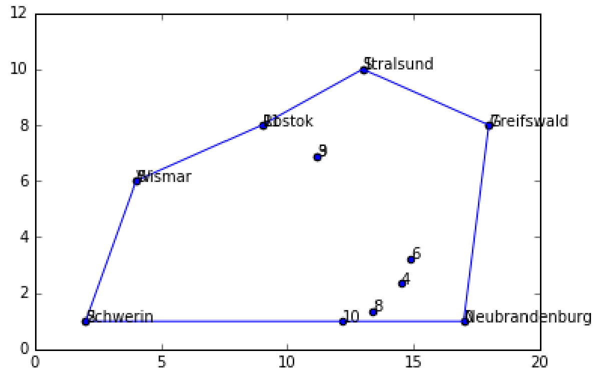


Figure 5: Path.

In the end we need to compare our solution with the one that will be returned by Concorde TSPLIB (Fig. 6). The paths are identical. It is hard to estimate time and memory, because in both cases they are too small. Concorde rounded time to 0.

Figure 6: TSPLIB path.