Artificial Neural Networks and other Learning Systems, 2D1432

Exercise 4: Self-Organizing Maps

1 Objectives

When you are finished you should understand:

- the different components in the SOM algorithm
- the role of the neighborhood
- how SOM-networks can be used to fold high-dimensional spaces
- how SOM-networks can be used to cluster data

2 Introduction

Self-Organising Maps (SOM) are networks which map points in the input space to points in an output space while preserving the topology. Topology preservation means that points which are close in the input space should also be close in the output space. Normally, the input space is of high dimension while the output is one- or two-dimensional.

This kind of network is particularly useful in situations where complex high-dimensional data needs to be presented in a form understandable for humans. Visual information is essentially two-dimensional, and it is often desirable to organise data in planar graphs or tables.

In this exercise you will implement the core algorithm of SOM and use it for three different tasks. The first is to order objects (animals) in a sequential order according to their attributes. The second is to find a circular tour which passes ten prescribed points in the plane. The third is to make a two-dimensional map over voting behaviour of members of the swedish parliament. In all three cases the algorithm is supposed to find a low-dimensional representation of higher-dimensional data.

For RBF networks: given an input find the RBF unit that has the highest activation (based on the center of that unit and its spread)

2.1 The SOM algorithm

The basic algorithm is fairly simple. For each training example:

1. Calculate the similarity between the input pattern and the weights arriving at each output node.
2. Find the most similar node; often referred to as the *winner*.
3. Select a set of output nodes which are located close to the winner *in the output grid*. This is called the *neighborhood*.
4. Update the weights of all nodes in the neighborhood such that their weights are moved closer to the input pattern.

Neighborhood is only defined in the topological/output space, doesn't matter what happened in the input space

We will now go through these steps in somewhat more detail.

Measuring similarity

Similarity is normally measured by calculating the euclidian distance between the input pattern and the weight vector. Note that this means that the weights are not really used as proper weights, i.e. they are not multiplied with the input. If we have the input pattern $\bar{x}$ and the i'th output node has a weight vector $\bar{w}_i$, then the distance for that output node is

$$d_i = \sqrt{(\bar{x} - \bar{w}_i)^T \cdot (\bar{x} - \bar{w}_i)}$$

We only need to find out which output node has the minimal distance to the input pattern. The actual distance values are not interesting, therefore we can skip the square root operation to save some computer time. The same node will still be the winner.

Neighborhood

The neighborhood defines the set of output nodes close enough to the winner to get the privilige of having its weights updated. It is important not to confuse the distances in the previous step with the distances in the neighborhood calculation. Earler we talked about distances in the input space, while the neighborhood is defined in terms of the output space. The output nodes are arranged in a grid, see figures 1 and 2. When a winning node has been found, the neighborhood constitutes the surrounding nodes in this grid. In a one-dimensional grid, the neighbors are simply the nodes where the index differs less than a prescribed amount. In the two-dimensional case, is is normally sufficient to use a so called Manhattan distance, i.e. to add the absolute values of the index differences in row and column directions.

One important consideration is how large the neighborhood should be. The best strategy is normally to start off with a rather large neighborhood and gradually making it smaller. The large neighborhood at the beginning is necessary to get an overall organization while the small neighborhood at the end makes the detailed positioning correct. Often some trial-and-error experimenting is needed to find a good strategy for handling the neighborhood sizes.

In one of the tasks in this exercise you will need a circular one-dimensional neighborhood. This means that nodes in one end of the vector of output nodes

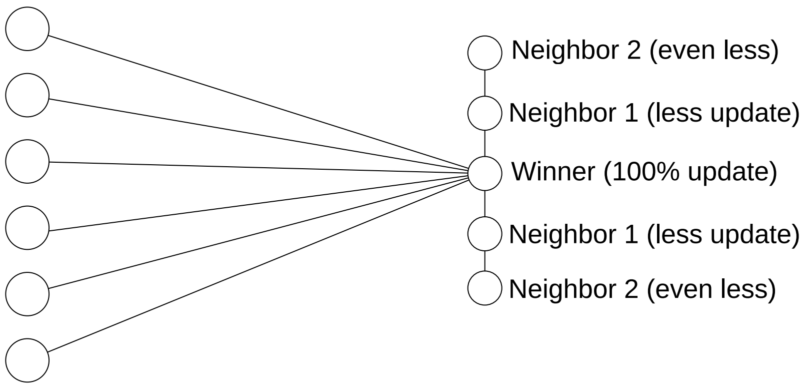

Figure 1: SOM network structure for mapping a 6-dimensional input (left) to a 1-dimensional grid (right). All input nodes are connected to all output nodes but in the figure, only connections to one particular output node are shown. The algorithm picks the output node which has the shortest distance between the input pattern and its weight vector. The weights are then updated for this winning node and for its neighbours in the output grid.

are close to those in the other end. This can be achieved by modifying how differences between indices are calculated.

Weight modification

Only the nodes sufficiently close to the winner, i.e. the neighbours, will have their weights updated. The update rule is very simple: the weight vector is simply moved a bit closer to the input pattern:

$$\bar{w}_i \leftarrow \bar{w}_i + \eta(\bar{x} - \bar{w}_i)$$

where η is the step size. A reasonable step size in these tasks is $\eta = 0.2$.

Now it is time to start with the three tasks. You will have to write the central parts of the algorithm yourself. Remember that Matlab is good at handling vectors and matrices so try to avoid dealing with individual elements. The main learning loop can be written in less than 20 lines.

Presentation of the result

After learning, the weight vectors will represent the positions in the input space where the output nodes give maximal response. However, this is normally not what we want to show. It is often much more interesting to see where different input patterns end up in the output grid. In these examples we will use the training patterns also for probing the resulting network. Thus, we loop through the input patterns once more, but this time we only calculate the winning node. Depending on the type of data, different techniques can then be used to present the mapping from pattern to grid index. In the first example you will sort the patterns in order of winner indices; in the last example you will use color to visualize where different input patterns end up.

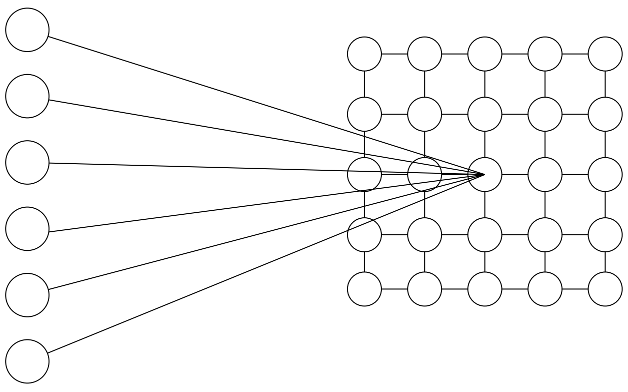

Figure 2: SOM network structure for mapping a 6-dimensional input (left) to a 2-dimensional grid (right). Like in figure 1, only the connections to one of the output nodes are shown.

3 Topological Ordering of Animal Species

The SOM algorithm can be used to assign a natural order to objects each characterized only by a large number of attributes. This is done by letting the SOM algorithm create a topological mapping from the high-dimensional attribute space to a one-dimensional output space.

As sample data, we will use a simple database of 32 animal species where each animal is characterized by 84 binary attributes. The SOM network will take these 84 values as input and the output will be 100 nodes arranged in a one-dimensional topology, i.e. in a linear sequence.

The SOM network will be trained by showing the attribute vector of one animal at a time. The SOM algorithm should now be able to create a mapping onto the 100 output nodes such that similar animals tend to be close while different animals tend to be further away along the sequence of nodes. In order to get this one-dimensional topology, the network has to be trained using a one-dimensional neighbourhood.

In the directory for this lab (`/info/ann06/labbar/lab4`) there is a Matlab file `animals.m` with all the animal data. This file defines the 32×84 matrix `props` where each row contains the attributes of one animal. There is also a vector `snames` with the names of the animals in the same order. This vector will only be used to print out the final ordering in a more readable format.

3.1 Your task

Your task is to write the core algorithm in Matlab. Use a weight matrix of size 100×84 initialized with random numbers between zero and one (hint: use the function `rand`). Use an outer loop to train the network for about 20 epochs, and an inner loop which loops through the 32 animals, one at a time.

For each animal you will have to pick out the corresponding row from the `props` matrix. This can be done using the Matlab statement `p = props(a,:)` where `a` is the row index. Then find the row of the weight matrix with the shortest distance to this attribute vector (`p`). Note that you can not use a scalar product since the attribute vectors are not normalized. Therefore you have to take the difference between the two vectors and calculate the length of this difference vector. (Hint: the Matlab function `min` returns two values where the second one is the index to the minimal value).

Once you have the index to the winning node, it is time to update the weights. Update the weights so that they come a bit closer to the input pattern. A suitable step size is 0.2. Note that only weights to the winning node and its neighbours should be updated. The neighbours are in this case the nodes with an index close to that of the winning one. You should start with a large neighbourhood and gradually make it smaller. Make the size of the neighbourhood depend on the epoch loop variable so that you start with a neighbourhood of about 50 and end up close to one or zero.

Finally, you have to print out the result, i.e. the animals in a natural order. Do this by looping through all animals once more, again calculating the index of the winning output node. Save these indices in a 32 element vector `pos`. By sorting this vector we will get the animals in the desired order. If we save the permutation vector from the sorting we can print the names of the animals using these statements:

```
[dummy, order] = sort(pos);
snames(order)'
```

Check the resulting order. Does it make sense? If everything works, animals next to each other in the listing should always have some similarity between them. Insects should typically be grouped together, separate from the different cats, for example.

4 Cyclic Tour

In the previous example, the SOM algorithm in effect positioned a one-dimensional curve in the 84-dimensional input space so that it passed close to the places where the training examples were located. We will now use the same technique to layout a curve in a two-dimensional plane so that it passes a set of points. In fact, we can interprete this as a variant of the TSP (travelling sales-person) problem. The training points correspond to the cities and the curve corresponds to the tour. With some luck, the SOM algorithm will be able to find a fairly short route which passes all cities.

The actual algorithm is very similar to what you implemented in the previons task. In fact, you might be able to reuse much of the code. The main differences are:

- The input space has two dimensions instead of 84. The output grid should have 10 nodes, corresponding to the ten cities used in this example.
- The neighbourhood should be circular since we are looking for a circular tour. When calculating the neighbours you have to make sure that the first and the last output node are treated as next neighbours.
- The size of the neighbourhood must be smaller, corresponding to the smaller number of output nodes. It is reasonable to start with a neighbourhood size of 2 and then change it to 1 and finally zero.

- When presenting the result, it is better to plot the suggested tour graphically than to sort the cities.

The location of the ten cities is defined in the file `cities.m` which defines the 10×2 matrix `city`. Each row contains the coordinates of one city (value between zero and one).

If you use a 10×2 matrix `w` to store the weights, then the following code can be used to plot both the tour and the training points:

```
tour = [w;w(1,:)];
plot(tour(:,1),tour(:,2),'b-*',city(:,1),city(:,2),'+')
```

5 Data Clustering: Votes of MPs

The file `votes.m` contains data about how all 349 members of the swedish parliament did vote in the 31 first votes during 2004–2005. There are also three additional files `mpparty.m`, `mpsex.m` and `mpdistrict.m` with information about the party, gender and district of each member of parliament (MP). Finally, there is a file `mpnames.txt` with the names of the MPs. Your task is to use the SOM algorithm to position all MPs on a 10×10 grid according to their votes.

By looking at where the different parties end up in the map you should be able to see if the MP's votes actually reflect the traditional left–right scale, and if there is a second dimension as well. You should be able to see which parties are far apart and which are close.

By looking at the distribution of female and male MPs you could get some insight into whether MPs tend to vote differently depending on their gender. You can also see if there is a tendency for MPs from different districts to vote systematically different.

The file `votes.m` defines a 349×31 matrix `votes`. Each row corresponds to a specific MP and each column to a specific vote. The elements are zero for a **no**-vote and one for a **yes**-vote. Missing votes (abstrained or non-present) are represented as 0.5.

You should use the SOM algorithm to find a topological mapping from the 31-dimensional input space to a 10×10 output grid. The network should be trained with each MPs votes as training data. If all works well, voting patterns that are similar will end up close to each other in the 10×10 map.

5.1 Useful Matlab tricks

Indexing output units

The output nodes are conceptually arranged in a two-dimensional grid. Still, it may be easier to write the algorithm itself so that all the nodes are represented in a vector, using a single index. The Matlab function `reshape` can be used to transform between the two representations. For example, you can calculate the indices in the two-dimensional grid and store them i two vectors like this:

```
[x,y]= meshgrid([1:10],[1:10]);
xpos = reshape(x, 1, 100);
ypos = reshape(y, 1, 100);
```

Now, you can write, e.g. `xpos(winner)` to get the x-position in the grid of the winner node.

Displaying the results

By locating the winning output node for each training example, i.e. for each member of parliament (MP), you can build a 349-element vector, `pos`, where each element is a number between 1 and 100. Using these two statements:

```
a = ones(1,100)*350;
a(pos) = 1:349;
```

you will get a vector `a` where each element corresponds to one output node and contains the index to an MP who belongs to that position. In fact, several MPs may end up at the same output node and this code will simply keep the last one. Nodes which never win will have the index number 350 (remember: there are only 349 MPs).

Load the file `mpparty.m` which defines a vector `mpparty` with the party for each MP. The parties are coded with numbers from 1 to 7. The following code will add a 350'th element and then draw a colored map of where the parties have ended up.

```
p=[mpparty;0];
image(p(reshape(a,10,10))+1);
```

The `+1` is needed because the `image` function plots both index zero and one as black. The colors codes for the different parties are defined in the `mpparty.m` file.

By replacing `mpparty` with `mpsex` or `mpdistrict`, you can look at the distribution of women versus men or the distribution of the different voting districts, respectively.

Good luck!