Machine Learning CMPU 395

Assignment 4: Self-Organizing Maps

1 Task

The task is to run the self-organizing map (SOM) algorithm on different datasets and study its behavior. A skeleton code for mapping into one dimension is given in the instructions. You will have to find which parameters work best for each particular dataset.

For highest grade you will have to modify the code to map into a two-dimensional representation. You can either use one of the given datasets, or use any other data you may find interesting.

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 to humans. Visual information is essentially two-dimensional, and the SOM algorithm may be useful in organizing complex data in the form of planar graphs or tables.

In this assignment you will use the SOM algorithm for three different tasks, where the third is optional, but required to reach the highest grades. In all three cases the algorithm is supposed to find a low-dimensional representation of higher-dimensional data. 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 some data of your own choice. You may use data from the supplied files containing voting behaviour of the House of Representatives, or, alternatively, the members of the Swedish parliament.

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 of 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.

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 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 neighbors, will have their weights updated. The update rule is very simple: the weight vector is simply

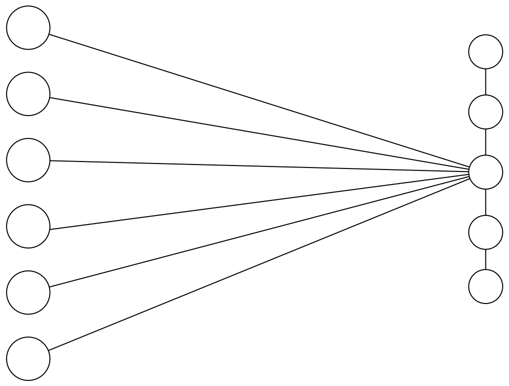

Figure 1: SOM network structure for mapping a 6-dimensional input (left) to a 1-dimensional grid (right). All input nodes are connected to all output nodes but in the figure, only connections to one particular output node are shown. The algorithm picks the output node which has the shortest distance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

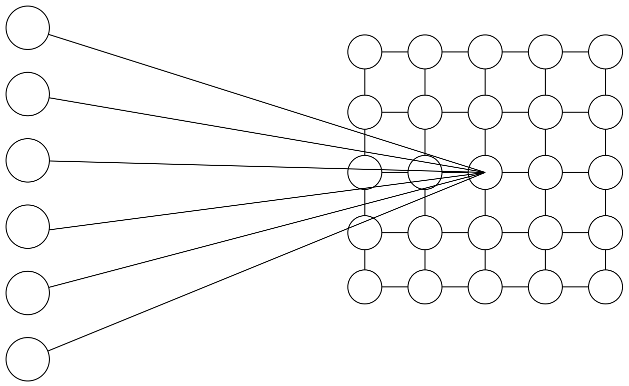

Figure 2: SOM network structure for mapping a 6-dimensional input (left) to a 2-dimensional grid (right). Like in figure 1, only the connections to one of the output nodes are shown.

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 $\eta = 0.2$.

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 may want to use color to visualize where different input patterns end up.

2.2 Sample Code

The core of the SOM algorithm can be efficiently computed making use of `numpy` arrays and operations. Here is some skeletal code:

```
import numpy
import animals as a

w = appropriatelySizedMatrixOfRandomNumbers

for nbh in listOfNeighborhoodSizes:
    # Loop through the dataset
    for x in a.animals:
        # Difference between x and all prototype vectors
        diff = a.props[x] - w

        # Calculate sum of squared differences
        # for all prototypes simultaneously
        dist = numpy.sum(diff*diff, axis=1)

        # Locate the winner (index to smallest distance)
        winner = numpy.argmin(dist)

        # Fill in code to find the neighbordhood here

        # Update weights of all nodes in the neighborhood
        for i in theNeighborhood:
            w[i] += diff[i] * 0.2
```

We store the prototype vectors as rows in the matrix w. The data in the `animals.py` file is stored in a dictionary called `props` which should be indexed by the names of each animal. These names are found as elements in the list `animals`.

Note that the distance calculation needed to identify the winner is here done using matrix operations. First, the difference between all rows (i.e., prototype vectors) and the datapoint is calculated, yielding a new matrix (called `diff`). Then, all elements in this matix are squared, followed by a row-wise summation, resulting in the vector `dist`. Finally, `numpy.argmin` is used to find the index in that vector containing the smallest number, i.e. the index of the winner.

You have to fill in the code to make the last statement loop over the nodes in the neighborhood of the winner. Note that you should make use of the `nbh`-variable which is supposed to go through a series of smaller and smaller values to get proper global ordering followed by good convergence. You can simply replance the variable `listOfNeighborhoodSizes` in the sample code with a list of suitable numbers.

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 neighborhood.

The file `animals.py` defines three variables: `animals`, `properties`, and `props`. `animals` is a list of strings; the names if all the animals. `properties` is another list of strings; the names of all the properties. You will probably not need to use this. Finally, `props` is a dictionary, indexed by an animal name and containing a `numpy`-vector of length 84 with values 0 or 1 depending on if the animal has the corresponding property or not.

3.1 Your task

Use a weight matrix of size 100×84 initialized with small random numbers. Complete the code from the example above to train the weight matrix.

Experiment to find a suitable list of neighborhood sizes. You should start with a large neighborhood and gradually make it smaller.

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 and print the animals in the order of these indices. You can do this by saving the indices in a separate dictionary, `order`, and then use the Python function `sorted` to get a sorted list of the animals.

```
order = {}
for x in a.animals:
    ...find the winner...
    order[x] = winner

print sorted(a.animals, cmp=compare)
```

To make `sorted` use the order stored in the `order` dictionary wee need to supply a function which does the comparition the way *we* want it, i.e. by comparing the indices stored in the `order` dictionary:

```
def compare(x, y):
    if order[x] < order[y]:
        return -1
    if order[x] > order[y]:
        return 1
    return 0
```

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 should be able to reuse most of the code. The main differences are:

- The input space has two dimensions instead of 84. The output grid should have at least 10 nodes, corresponding to the ten cities used in this example.

- The neighborhood should be circular since we are looking for a circular tour. When calculating the neighbors you have to make sure that the first and the last output node are treated as next neighbors. Tip: you may find the `%`-operator in Python useful here; it returns the remainder after an integer division, which is what is needed to make vector indices wrap around at the ends.
- The size of the neighborhood must be smaller, corresponding to the smaller number of output nodes.
- 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.py` which defines the 10×2 matrix `city`. Each row contains the coordinates of one city (value between zero and one).

When learning is complete, use `pylab.plot` to plot a line starting at one prototype vector (i.e. rown in your w matrix) to successive vectors. Remember to close the loop by ending the line where you started. You should also plot all the training datapoints, preferrably as points or crosses.

Note that you will normally not get a perfect solution. You will have to accept that the algorithm may miss a few points along the tour. You may want to experiment with having more than 10 prototype vectors to see if this helps.

5 Two-Dimensional Neighborhood

The final, optional, task is to modify the algorithm to get a two-dimensional neighborhood. You can use different strategies to get this effect. One is to replace the w matrix with a tensor having three indices. Another, possibly simpler strategy, is to use these functions to convert between indices and positions in the two-dimansional grid:

```
def toGrid(index):
  return (index % gridSide, index // gridSide)

def fromGrid(i, j):
  return i + j * gridSide
```

You may use any suitable data for this task. The files `house.py` and `votes.py` contain data on how membes of the House of Representatives, and the members of the Swedish Parliament, respectively, voted.

The file `house.py` contains data downloaded and extracted from the official webpage in 2009. It contains data on the first 100 votes by each member, stored as a dictionary. There is also information about which party, state, and district they represent.

The file `votes.py` contains data about how all 349 members of the swedish parliament did vote in the 31 first votes in 2004. There is also additional information about the party, gender and district of each member of parliament.

By looking at where the different parties end up in the map you may be able to see if the 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 may get some insight into whether MPs tend to vote differently depending on their gender.

Another option may be to use the animal data again, this time producing a two-dimensional layout of the animals.

5.0.1 Plotting

One way of visualizing the result of a two-dimensional mapping is to create a matrix corresponding to the output grid and then store numbers encoding factors of interest about the data on the winning location. For example, you could store the number one for all winners when the datapoints belong to a certain party. You can then use the `pylab` function `matshow` to see a colorcoded display of the contents of the matrix.

Good luck!