Artificial Neural Networks and Deep Architectures, DD2437

Lab assignment 2

Radial basis functions, competitive learning and self-organisation

1 Introduction

In this lab the focus will be on unsupervised neural network approaches that involve competitive learning and the related concept of self-organisation. In this context, you will also experiment with Radial-Basis Function (RBF) networks, which incorporate both unsupervised and supervised learning to address classification and regression tasks. The second part of the lab is devoted to the most famous representative of self-organising NNs - Kohonen maps, commonly referred to self-organising maps (SOMs). These networks map points in the input space to points in the output space with the preservation of the topology, which means that points which are close in the input space (usually high-dimensional) should also be close in the output space (usually low-dimensional, i.e. 1D, 2D or 3D). These networks can be used to help visualise high-dimensional data by finding suitable low-dimensional manifolds or perform clustering in high-dimensional spaces. In both cases the objective is to organise and present complex data in an intuitive visual form understandable for humans.

1.1 Aims and objectives

When you successfully complete the assignment you should:

- know how to build the structure and perform training of an RBF network for either classification or regression purposes
- be able to comparatively analyse different methods for initialising the structure and learning the weights in an RBF network
- know the concept of vector quantisation and learn how to use it in NN context
- be able to recognise and implement different components in the SOM algorithm
- be able to discuss the role of the neighbourhood and analyse its effect on the self-organisation in SOMs
- know how SOM-networks can be used to fold high-dimensional spaces and cluster data

1.2 Scope

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.

2 Background

2.1 Radial-basis function networks

In this experiment you will use a set of RBFs to approximate some simple functions of one variable. The network is shown in fig 1.

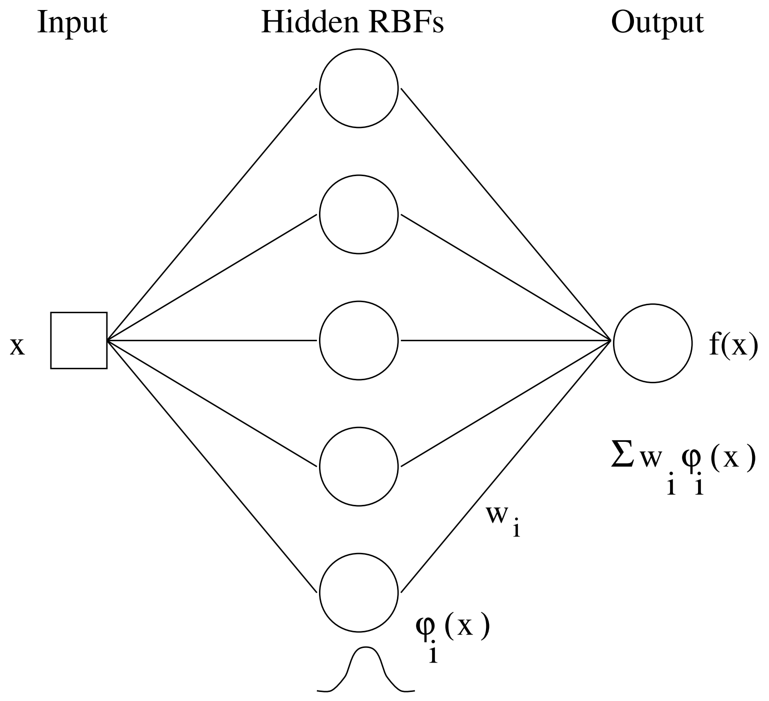

FIGURE 1: A simple RBF network which is used for one dimensional function approximation. The weights between the RBFs and the output layer may be trained in several ways. Two methods which are used in this exercise is the method of *least squares* and the *delta rule*.

We will use Gaussian RBFs, i.e. the units in the hidden layer implement the following transfer function:

$$\phi_i(x) = e^{\left(\frac{-(x-\mu_i)^2}{2\sigma_i^2}\right)} \tag{1}$$

where μ_i is the position for unit i and σ_i^2 is its variance. The output layer calculates the weighted sum of the n hidden layer units:

$$\hat{f}(x) = \sum_i^n w_i \phi_i(x) \tag{2}$$

where $\hat{f}(x)$ is an approximation of the desired function.

The units in the hidden layer are called "radial-basis functions" since they work as a *basis* in which the function $\hat{f}(x)$ can be expressed.[1] The units are often *radially* symmetric as is the case in (1).

Look again at figure 1. Note how the set of RBFs maps each pattern in the input space to an n-dimensional vector. n is usually higher than the dimension of the input space. Patterns belonging to different classes in a classification task are usually easier to separate in the higher-dimensional space of the hidden layer than in the input space. In fact, two sets of patterns which are not linearly separable in the input space can be made linearly separable in the space of the hidden layer.

2.1.1 Computing the weight matrix

The learning algorithm should find a set of weights w_i so that $\hat{f}(x)$ is a good approximation of $f(x)$, i.e. we want to find weights which minimize the total approximation error summed over all N patterns used as training examples:

$$\textit{total error} = \sum_k^N (\hat{f}(x_k) - f(x_k))^2 \tag{3}$$

We begin by defining $f_k = f(x_k)$, where $f(\cdot)$ is the target function and x_k is the kth pattern, and write a linear equation system with one row per pattern and where each row states equation (2) for a particular pattern:

$$\begin{array}{rcl}
\phi_1(x_1)w_1 + \phi_2(x_1)w_2 + \cdots + \phi_n(x_1)w_n & = & f_1 \\
\phi_1(x_2)w_1 + \phi_2(x_2)w_2 + \cdots + \phi_n(x_2)w_n & = & f_2 \\
& \vdots & \\
\phi_1(x_k)w_1 + \phi_2(x_k)w_2 + \cdots + \phi_n(x_k)w_n & = & f_k \\
& \vdots & \\
\phi_1(x_N)w_1 + \phi_2(x_N)w_2 + \cdots + \phi_n(x_N)w_n & = & f_N
\end{array} \tag{4}$$

If $N > n$, the system is *overdetermined* so we cannot use Gaussian elimination directly to solve for $\mathbf{w}$. In fact, in practice there is no exact solution to (4). Please, reflect in this context over the following questions:

- What is the lower bound for the number of training examples, N?
- What happens with the error if $N = n$? Why?
- Under what conditions, if any, does (4) have a solution in this case?

[1]Note that $\hat{f}(x)$ in (2) is a linear combination of radial-basis functions, just as a vector in a two-dimensional Cartesian space is a linear combination of the basis vectors $\bar{e}_x$ and $\bar{e}_y$.

- During training we use an error measure defined over the training examples. Is it good to use this measure when evaluating the performance of the network? Explain!

Below we will look at two methods for determining the weights w_i, batch learning using *least squares* and sequential (incremental, on-line) learning using the *delta rule*. In both cases, the objective is to minimise (3).

Least squares We can write (4) as

$$\mathbf{\Phi w} = \mathbf{f} \tag{5}$$

where

$$\mathbf{\Phi} = \begin{pmatrix} \phi_1(x_1) & \phi_2(x_1) & \dots & \phi_n(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \dots & \phi_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_N) & \phi_2(x_N) & \dots & \phi_n(x_N) \end{pmatrix} \quad \text{and} \quad \mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \tag{6}$$

Our error function (3) becomes

$$\textit{total error} = \|\mathbf{\Phi w} - \mathbf{f}\|^2 \tag{7}$$

According to standard textbooks in linear algebra and numerical analysis, we obtain the $\mathbf{w}$ which minimizes (7) by solving the system

$$\mathbf{\Phi}^\top \mathbf{\Phi w} = \mathbf{\Phi}^\top \mathbf{f} \tag{8}$$

for w. (8) are the *normal equations.* The result is called the *least squares solution* of (5).

The delta rule It is not always that all sample patterns from the input space are accessible simultaneously. It is, on the contrary, a rather common situation that a neural network is operating on a continuous stream of data, to which it needs to adapt. Using linear algebra, it is possible to extend the technique described above to the case where the network needs to respond and adapt to newly incoming data at each time-step.[2]

We will instead derive the *delta rule* for RBF network, which is an application of the *gradient descent method* to neural networks, as you should know by now. The derivation follows the same line of reasoning as for the perceptron networks in Lab 1. Here we emphasise the inceremental nature of learning (unlike focus on batch in the discussion of delta and generalised delta rules in Lab 1). Therefore, in this new context we assume that we may not have a fixed set of input patterns, so we approximate the error criterion (3) using the *instantaneous error*, $\hat{\xi}$, as an estimate for the expected error, ξ, given the most recent pattern sample x_k:

$$\xi \approx \hat{\xi} = \frac{1}{2}(f(x_k) - \hat{f}(x_k))^2 = \frac{1}{2}e^2 \tag{9}$$

[2]C.f. *recursive least squares* and the *Kalman filter.*

We want $\hat{\xi} \longrightarrow 0$ as $t \longrightarrow \infty$, and we want it to go as fast as possible. Therefore, for each time step, we take a step $\mathbf{\Delta w}$ in the direction where the error surface $\hat{\xi}(\mathbf{w})$ is steepest, i.e. in the negative gradient of the error surface:

$$\begin{aligned}
\mathbf{\Delta w} &= -\eta \nabla_{\mathbf{w}} \hat{\xi} \\
&= -\eta \frac{1}{2} \nabla_{\mathbf{w}} (f(x_k) - \mathbf{\Phi}(x_k)^\top \mathbf{w})^2 \\
&= \eta (f(x_k) - \mathbf{\Phi}(x_k)^\top \mathbf{w}) \mathbf{\Phi}(x_k) \\
&= \eta e \mathbf{\Phi}(x_k)
\end{aligned} \tag{10}$$

where

$$\mathbf{\Phi}(x_k) = \begin{pmatrix} \phi_1(x_k) \\ \phi_2(x_k) \\ \vdots \\ \phi_n(x_k) \end{pmatrix}$$

As you know, equation (10) is the *delta rule* and η is the *learning rate constant.* Ideally, η is large enough so that we get to the optimum (the least squares solution) in one step. However, since $\hat{\xi}$ only is an approximation and since the data contains noise, a too large η could cause us to miss the optimum and instead add error to $\mathbf{w}$. This can give rise to oscillations.

2.2 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 *neighbourhood.*
4. Update the weights of all nodes in the neighbourhood 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 Euclidean distance between the input pattern and the weight vector. Note that this implies that the weights describe the centers of nodes in the input space, i.e. they are not multiplied with the input as in perceptron. If we have the input pattern $\bar{x}$ and the i'th node has a weight vector $\bar{w}_i$, then the distance for that node is

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

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 compute time. The same node will still be the winner.

Neighbourhood The neighbourhood defines the set of 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 neighbourhood calculation. Earlier we talked about distances or rather measuring similarity in the input space, while the neighbourhood is defined in terms of the output space. The nodes are arranged in a grid. When a winning node has been found, the neighbourhood 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 integer number. 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. Sometimes it may be easier to describe it in terms of a Gaussian function (in the discrete space of indices).

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

In one of the tasks in this exercise you will need a circular one-dimensional neighbourhood. This means that nodes in one end of the vector of 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 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 $\eta = 0.2$.

The algorithm is so imple that you are asked here to write all the implementation by yourselves from scratch without relying on any dedicated SOM libraries.

Presentation of the result After learning, the weight vectors 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.

3 Assignment - Part I

Important notes:

- Please execute all the tasks below and report them in your short reports where you focus on key points. For presentation of your results in lab sessions please choose the key findings so that you could tell a short but coherent and insightful story.
- Most tasks are formulated with a great level of detail whereas the more advanced ones allow certain room for your own assumptions, decisions. Please be clear about these assumptions even if they are arbitrary (you have full freedom to make these simplifying decisions unless it is clearly stated that you should not.)
- Most developments in this part of the lab assignment should be implemented from scratch wthout using any dedicated libraries for neural networks (except for a comparative analysis with a multi-layer perceptron). The computations are relatively easy to code and light to run. If you get stuck I see a possibility of relying on the existing functions for competitive learning (CL).

3.1 Batch mode training using least squares - supervised learning of network weights

In this simple assignment, you should focus on supervised learning of weights of the RBF network built to address a simple regression problem. Please implement both batch and incremental learning algorithms (you will need delta rule for incremental learning in the next task when noise is introduced) from scratch without using dedicated NN toolboxes. The two function to approximare are $\sin(2x)$ and $\text{square}(2x)$ (square is a rectangular curve serving as a "box" envelope for the sine wave). Note that the input space is $\mathbb{R}$, so each pattern $x_1, x_2, \ldots, x_N$ in (6) is in fact a scalar. Begin by creating a *column vector* containing the points (patterns) where you want to evaluate your function. Let's limit the regression to the interval $[0, 2\pi]$. Sample this interval starting from 0 with the step size of 0.1 and calculate the values of the two functions at the these points to obtain the corresponding training sets. The testing sets could be generated analogously with sampling starting from 0.05 and the same step size. For a varying number of RBF nodes, please place them by hand in the input space according to your judgement, and set the same variance to each node. Next, apply your batch learning algorihtm on your training set to adjust the output weights and test accordingly on the hold-out set. For both functions (studied indpenedently), please consider and discuss the following issues (which involve running the suggested experiments):

- Try to vary the number of units to get the absolute residual error below 0.1, 0.01 and 0.001 in the residual value (absolute residual error is understood as the average absolute difference between the network outputs and the desirable target values). Please discuss the results, how many units are needed for the aforementioned error thresholds?

- How can you simply transform the output of your RBF network to reduce the residual error to 0 for the square($2x$) problem? Still, how many units do you need? In what type of applications could this transform be particularly useful?

3.2 Regression with noise

Now please add zero-mean Gaussian noise (with the variance of 0.1) to both training and testing datasets for the function $\sin(2x)$. Apply batch learning, as in the previous part of the assignment,and compare with on-line learning with delta rule (given that the data points are randomly shuffled in each epoch). As before, please place the RBF units by hand wherever you consider suitable and fix the same width for them. Analogously, iterate over a reasonable number of RBF nodes and find the overall network configurations (independently for the two training modes) that result in the absolute residual error below 0.1, 0.01 and 0.001 on the testing datasets. Please consider the following points and make the requsted analyses:

- Compare the two learning approaches in terms of the number of epochs and the number of nodes needed to obtain the requested performance levels. How does it compare, particularly with respect to the number of RBF nodes, to the batch mode training of $\sin(2x)$ without noise, as in the previous assignment task? Compare the rate of convergence for different learning rates,
- How important is the positioning of the RBF nodes in the input space? What strategy did you choose? Is it better than random positioning of the RBF nodes? Please support your conclusions with quantitative evidence (e.g., error comparison).
- What are the main effets of changing the width of RBFs?
- How does the rate of convergence change with different values of `eta`?
- Please compare your optimal RBF network trained in batch mode with a two-layer perceptron trained with backprop (also in batch mode). To find a suitable candidate two-layer perceptron architecture try to use the same overall number of hidden units as in the RBF network and distribute them in different ways over the two hidden layers (use the existing libraries of your choice for these simulations). Please remember that generalisation performance and training time are of greatest interest.

3.3 Competitive learning for the initialisation of RBF units

Now we will take a look at the problem of placing the RBFs in input space. We will use a version of *Competitive Learning* (CL) for Vector Quantization. The simple *Competitive Learning* algorithm we use here can only adjust the positions of the RBF units without adjusting the width of the units. Therefore you will have to make these adjustment yourselves based on the distribution of data around the cluster centers found with the simple CL algorithm. At each iteration of CL a training vector is randomly selected from the data. The closest RBF unit (usually called the *winning* unit) is computed, and this unit is updated, in such a way that it gets closer to the training vector. The other units may or may not (depending on the version of CL used) be moved towards it too, depending on distance. This way the units will tend to aggregate in the clusters in the data space. Please, couple the CL-based approach to RBF network initilisation with the aforementioned delta learning for the output weights.

- Compare the CL-based approach with your earlier RBF network where you manually positioned RBF nodes in the input space. Use the same number of units (it could be the number of unicts that allowed you to lower the absolute residual error below 0.01). Make this comparison for both noise-free and noisy approximation of $\sin(2x)$. Pay attention to convergence, generalisation performance and the resulting position of nodes.

- Introduce a strategy to avoid dead units, e.g. by having more than a single winner. Choose an example to demonstrate this effect in comparison with the vanilla version of our simple CL algorithm.

- Configure an RBF network with the use of CL for positioning the RBF units to approximate a two-dimensional function, i.e. from $\mathbb{R}^2$ to $\mathbb{R}^2$. As training examples please use noisy data from ballistical experiments where inputs are pairs: $<$angle, velocity$>$ and the outputs are pairs: $<$distance, height$>$. There are two datasets available: `ballist` for training and `balltest` for testing. First thing to do is to load the data and then train the RBF network to find a mapping between the input and output values. Please be careful with the selection of a suitable number of nodes and their initialisation to avoid dead-unit and overfitting problems. Report your results and observations, ideally with the support of illustrations, and document your analyses (e.g., inspect the position of units in the input space).

4 Assignment - Part 2

4.1 Topological Ordering of Animal Species

The SOM algorithm can be used to assign a natural order to objects 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, please use a simple database of 32 animal species where each animal is characterized by 84 binary attributes. The data is in the file `animals.dat`. This file defines the 32×84 matrix `props` where each row contains the attributes of one animal. The data are organised in row-by-row manner. There is also a file `animalnames.dat` with the names of the animals in the same order. This vector should only be used to print out the final ordering in a more readable format. These 84 values serve as input and 100 nodes arranged in a one-dimensional topology, i.e. in a linear sequence, constitute the output.

Train the SOM network 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 particular, your task is to write the core algorithm. Use a weight matrix of size 100×84 initialized with random numbers between zero and one. 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. Then find the row of the weight matrix with the shortest distance to this attribute vector (`p`). Note that you cannot 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. 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. 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.

4.2 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. Now the same technique can be used to layout a curve in a two-dimensional plane so that it passes a set of points. In fact, this can be interpreted as a variant of the travelling salesman problem. The training points correspond to the cities and the curve corresponds to the tour. SOM algorithm should 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.dat` which defines the 10×2 matrix `city`. Each row contains the coordinates of one city (value between zero and one).

Please plot both the tour and the training points. Give your interpretation.

4.3 Data Clustering: Votes of MPs

The file `votes.dat` 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.dat`, `mpsex.dat` and `mpdistrict.dat` 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 votes of the MPs 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.dat` defines a 349×31 matrix `votes`. Data are organised in row-by-row manner. Each one of 349 rows corresponds to a specific MP and each one of 31 columns 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.

Please display the results with respect to different attributes (i.e. party, gender, district) and describe the results, provide your interpretation.

Good luck!