Artificial Neural Networks and other Learning Systems, DD2432

Exercise 2:

Radial-Basis Functions and Self Organization

1 Objectives

In this exercise you will experiment with Radial-Basis Functions (RBFs). RBF networks is a neural network type which can be used for classification as well as function approximation. During this exercise you will study two aspects of RBFs: supervised learning of the network weights and unsupervised learning (self-organized learning) of the positions of RBF units in the input space. In both cases, function approximation will be an example of the practical use of an RBF network.

When you are finished you should:

- understand the structure of an RBF network
- know methods for learning the weights in an RBF network
- know methods for learning the positions of units in the input space
- understand the terms *vector quantization* and *expectation maximization*
- know how to use an RBF network to approximate functions

2 Computer Environment

The tool you will use here is MatLab, in which it is easy to do the necessary manipulations of vectors and matrices.

If you are using the CSC Linux computers, copy the files needed for the exercise to your personal course directory:

```
> cd ~/ann13
> cp -pr /info/ann13/labbar/lab3 .
> cd lab2
```

(You must `cd` to this directory *before* starting MatLab.)

3 Introduction

3.1 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.

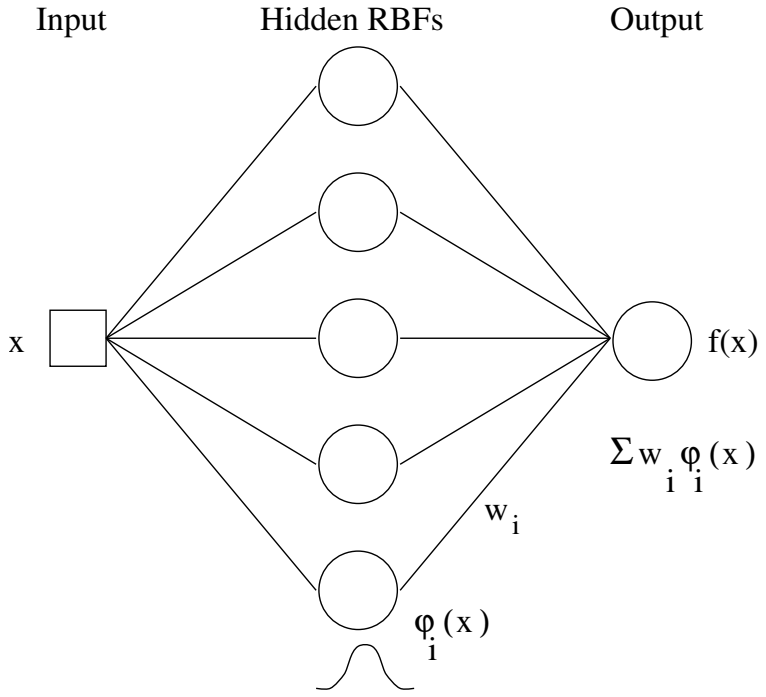

FIGURE 1: A simple RBF network which is used for one dimensional function approximation. The weights between the RBFs and the output layer may be trained in several ways. Two methods which are used in this exercise is the method of *least squares* and the *delta rule*.

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

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

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).

While a network with sigmoidal threshold functions needs multiple layers in order to be able to approximate any function, RBF networks can do this with only a single layer. The reason is the "local" nature of the nonlinearity: the gaussian is large only in a limited domain which makes it suitable as a basis function.

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.

3.2 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:

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

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:

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

If $N > n$, the system is *overdetermined* so we can't use Gaussian elimination directly to solve for $\mathbf{w}$. In fact, in practise there is no exact solution to (4).

- 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?
- 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 incremental learning using the *delta rule*, both of which are aimed at minimizing (3).

[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 $\bar{e}_y$.

The lower bound is n, since a smaller N will cause the matrix function group incompatible

If N = n and the phi matrix is full rank, then we could obtain a set of weights that could perfectly approximate the target function (fhat = f). Then the training error will be 0.

N = n and phi full rank

Of course we could get a better error performance, but the network may work badly with unseen data. This is because the structure of network is dependent of the training data. So, using training data to test it is kind of cheating behavior.

3.3 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).

3.4 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* which is an application of the *gradient descent method* to neural networks.

In this new context we have no fixed set of input patterns, so we should reformulate our error criterion (3):

$$\xi = \textit{expected error} = \mathbf{E}\left[\frac{1}{2}(f(x) - \hat{f}(x))^2\right] \tag{9}$$

Clearly, we can't find an exact expression for ξ. We now make the choice of using the *instantaneous error* as an estimate for ξ: Given the most recent pattern sample x_k, we write:

$$\xi \approx \hat{\xi} = \frac{1}{2}(f(x_k) - \hat{f}(x_k))^2 = \frac{1}{2}e^2 \tag{10}$$

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

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

error surface $\hat{\xi}(\mathbf{w})$ is steepest, i.e. in the negative gradient of the error surface:

$$\begin{aligned}\Delta\mathbf{w} &= -\eta\nabla_{\mathbf{w}}\hat{\xi} \\ &= -\eta\frac{1}{2}\nabla_{\mathbf{w}}(f(x_k) - \mathbf{\Phi}(x_k)^\top\mathbf{w})^2 \\ &= \eta(f(x_k) - \mathbf{\Phi}(x_k)^\top\mathbf{w})\mathbf{\Phi}(x_k) \\ &= \eta e\mathbf{\Phi}(x_k)\end{aligned} \tag{11}$$

where

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

Equation (11) 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.

4 Supervised Learning of Network Weights

4.1 Batch mode training using least squares

Now we'll use the RBF network described above to approximate two different functions, $\sin(2x)$ and $\text{square}(2x)$. 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.

Before you proceed, it is a good idea to study the questions at the end of this section.

1. Begin by creating a *column vector* containing the points (patterns) where you want to evaluate your function. Since we'll begin with $\sin(2x)$ it is suitable to choose points in the interval $[0, 2\pi]$.

   Hint: Use the MatLab colon operator with stepsize 0.1.

2. Now calculate the desired function values at the points which you chose in step 1, that is we compute $\mathbf{f}$ in equation (5).

3. Before we can solve for $\mathbf{w}$, we need to compute Φ. There are two MatLab scripts[3] available which can help you:

   `makerbf`

   initializes some variables and computes the positions and variances[4] of the RBF units. It requires the points in the input space in the column vector `x` and the number of RBF units in the network (an integer) in `units`

[3] You can use the MatLab command `type` <*name of script*> to show how a script is written.
[4] μ_i and σ_i^2 in (1)

and stores positions and variances in the column vectors `m` and `var`. It positions the RBF units equidistantly over the interval.

```
Phi = calcPhi(x,m,var)
```

is a function which computes Φ, given positions and variances of the units, and a set of input patterns.

Positions and variances may be fixed equidistantly positioned over the interval or they may be positioned according to the probability distribution of the data. In this part of the exercise we use fixed positions and fixed variances.

4. You should now be able to compute $\mathbf{w}$. Note that we are using *supervised learning*: Given a set of input patterns, $\mathbf{x}$, and a set of desired outputs, $\mathbf{f}$, we are "training" the network (computing the weights). Also note that we work on a batch of N patterns (the number of rows in $\mathbf{x}$) simultaneously. Therefore, this is an example of *batch mode* training.

Hint: In MatLab it isn't necessary to formulate the *normal equations* when solving a system $\mathbf{Ax} = \mathbf{b}$. If $\mathbf{A}$ is rectangular with more rows than columns, $\mathbf{A}\backslash\mathbf{b}$ will return the least squares solution.

To view the results you can plot the values using the script

```
rbfplot1(x,y,f,units)
```

where

`x`	*column vector*	is the points where the function is evaluated
`y`	*column vector*	is the approximated function values
`f`	*column vector*	is the actual function values
`units`	*integer*	is the number of RBF units in the network.

The graph contains the *residual value* which is defined

$$r = \|\mathbf{f} - \mathbf{\Phi w}\|_\infty \tag{12}$$

Now try to vary the number of units to get below 0.1, 0.01 and 0.001 in the residual value. Print the results with 5 and with 6 units and discuss the result. To print the graph on paper, use the command

```
print
```

- How many units did you require to get down to a maximum (absolute) residual value of 0.1, 0.01 and 0.001?
- Give a good reason for the big difference in residual between 5 and 6 units for $\sin(2x)$.

Now perform an approximatition of square$(2x)$ in the same way.[5]

[5] By the function `square` we mean a square wave. The `square` function is already available in MatLab.

- How many units did you require, when approximating square$(2 * x)$, to come down to residual values of 0.1, 0.01 and 0.001?

- Approximating square$(2 * x)$ is a somewhat special case of function approximation since it is similar to another area of use for artificial neural networks. Which?

  Hint: ANNs can be used for pattern completion, noise reduction, ...

- Can you, with a suitable action (e.g. transforming network output), easy get down (for training values) to a residual value=0? What action? How many units did you require?

- Can an RBF network solve the XOR problem? If not, explain why not. If yes, explain how.

It is similar to noise reduction since function approximation using RBF network works in a similar manner that ignores those high frequency components

Yes. RBF network could nonlinearly map the unseparable problem from a lower dimension to a higher one, which makes it separable.

We could add one more unit in the hidden layer which works as a residual compensator. It subtracts the residual component from the output and makes the training error zero. When test data is inputed into network, the last unit will look up table of residual to find the corresponding value

4.2 On-line training using the delta rule

Least squares yields an optimal (in the sense of the square error measure (3)) solution to the approximation problem. It is, however, not so "neural network like". It requires a lot of memory as we need to store the Φ-matrix, which is of size $n \times N$, i.e. number of units times number of samples, and it requires that we can access all samples at the same instant of time.

Assume that we intend our RBF network to model how some device transforms a signal. The "device" could, for example, be an electrical circuit or an industrial process. At each time step k we measure the input signal into the device, x_k, and the output from the device, $f(x_k)$. Just as in section 4.1, the task for the network is to learn the mapping $f(\cdot)$, however, since data arrive one-by-one, it is suitable to update the weights every time an input x_k arrives. This is called *on-line training*.

The script

`diter`

runs `itermax` iterations, generating one random sample pattern x_k and using the delta rule (11) with that sample once per iteration. In order not to spend too much time plotting, it only plots data every `itersub`th iteration. `itermax` must be an even multiple of `itersub`. The variables `iter`, `itersub`, and, `itermax` are initiated by the script `makerbf`, so, if you want to use your own values, you should set them after calling `makerbf`. While random sample pattern are used in the delta rule during training, the variable `x` which you created in section 4.1 is used in the error measure.

`diter` also depends on two other variables which you have to set yourself:

`fun`	*string*	is the name of the mapping
`eta`	*float*	is the learning rate constant η.

Assume that our "device" implements the mapping $\sin(2x)$. Run `makerbf`, select values for `fun` and `eta`, and, start using `diter`! (You may get "log of zero" warnings. This is nothing to worry about.)

Hint: There is a function `sin2x(x) = sin(2*x)`, so a suitable name is `sin2x`.

Compare the rate of convergence for different values of η.[6]

- How many *units* and *iterations* did you require to come down to a maximum residual value of 0.01? What value(s) of η did you use?
- Now try approximating some function of your own choice. Use least squares or the delta rule as you wish.

5 RBF Placement by Self Organization

Now we will take a look at the problem of placing the RBFs in input space. We will use two different algorithms for this. One is a version of *Competitive Learning* (CL) for Vector Quantization, and the other is an *Expectation Maximization* algorithm (EM). The simple *Competitive Learning* algorithm we use here can only adjust the positions of the RBF units while the *Expectation Maximization* algorithm can also adjust the width of the units.

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.

The EM algorithm instead works by estimating where the clusters are. Each iteration step consists of weighing all the data points by the value of each basis function and then moving the units towards their respective centroid, adjusting the widths according to their variances.

The data you will use in these self organization experiments are two dimensional but the algorithms works for arbitrary number of dimensions. In the case you will try these on data with more dimensions the variables `p1` and `p2` select which dimensions you project on the display.

Start by initializing some plot parameters. (You may need to perform this command again later if, for some reason, the plot window created below has disappeared.)

```
>> plotinit
```

Now load the data. These experiments will use data situated in the variable `data`. The file `cluster.dat` contains some data points which can be loaded, using the following command:

```
>> data=read('cluster');
```

Define the number of RBF units you want to use, e.g. 5.

```
>> units=5;
```

Then call the script `vqinit`, which opens a plot window, positions the RBF units randomly in the space defined by `data` and plots data points and RBF units. The RBF units are plotted as circles representing the RBFs gaussian standard deviation σ.

[6] Different values of η can be optimal for different phases of the learning.

```
>> vqinit;
```

The simplest way to implement this algorithm is to move only the closest unit (the winner) each step. To specify that you want to use this "winner-take-all" strategy here you have to set the variable `singlewinner` to 1 (=yes), or otherwise to 0 (=no) if you allow all units to move (dependent on the distance).

Begin by setting `singlewinner` to 1.

```
>> singlewinner=1;
```

Then try to run the algorithm a few steps with:

```
>> vqstep;
```

As you see the new positions of the units will be plotted together with a line showing direction and length of the last move of the units. With the command:

```
>> vqiter;
```

you will see an animation plot showing each iterative step until stability.

Try a few times to run `vqinit` followed by `vqiter`. Then set `singlewinner` to 0, which allows all units to move each time, not only the closest one.

```
>> singlewinner=0;
```

and run a few steps again by using `vqinit` and `vqiter` a couple of times.

- What problem could be seen when using the single winner strategy (singlewinner=1)?
- What is the advantage of using this strategy?

Now you should insted use the *Expectation maximization* algorithm. In its batchwise version each iteration consists of two passes. In the first pass we estimate which data points "belongs to" (are "closest to") which RBF units. In the next pass new positions and variances are estimated from the corresponding data points.

As before we start with:

```
>> vqinit;
```

Also switch back to the single winner strategy:

```
>> singlewinner=1;
```

then run the batchwise EM algorithm either single steps with:

```
>> emstepb;
```

or iterate until convergence with:

```
>> emiterb;
```

The occasional units with large variance which don't "belong" to a specific cluster correspond to the units that never moved in the vector quantization case.

Now enable move of all units, and run the batchwise algorithm again:

```
>> vqinit;
>> singlewinner=0;
>> emiterb;
```

Experiment a little with these, vary the number of units and run with new initializations a few times.

- Describe differences between using the single winner strategy (singlewinner=1) and allowing all units to move (singlewinner=0) for the batchwise algorithm.

6 Function Approximation for Noisy Data

Finally, we will use an RBF network for approximating a function from $\mathcal{R}^2$ to $\mathcal{R}^2$. As training examples we will 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 x (input values=<angle,velocity>) and y (output values=<distance,height>) training and test vectors. Then we will train the RBF network to find a mapping between the input and output values.

```
>> [xtrain ytrain]=readxy('ballist',2,2);
>> [xtest ytest]=readxy('balltest',2,2);
```

Use e.g. 20 units and train the unsupervised part.

```
>> units=20;
>> data=xtrain;
>> vqinit;
>> singlewinner=1;
>> emiterb
```

In this case we have a two dimensional input space and a two dimensional output space. The two output values are coded with one unit each in the output layer. Here we will train the weight vectors to these units separately. Start as before by calculating the matrix ϕ of RBF responses:

```
>> Phi=calcPhi(xtrain,m,var);
```

Extract the two desired y vectors for train and test

```
>> d1=ytrain(:,1);
>> d2=ytrain(:,2);
>> dtest1=ytest(:,1);
>> dtest2=ytest(:,2);
```

and calculate the weight vectors by the pseudo inverse method

```
>> w1=Phi\d1;
>> w2=Phi\d2;
```

Now we can calculate approximations of training data

```
>> y1=Phi*w1;
>> y2=Phi*w2;
```

as well as approximations of test data

```
>> Phitest=calcPhi(xtest,m,var);
>> ytest1=Phitest*w1;
>> ytest2=Phitest*w2;
```

Finally we plot these

```
>> xyplot(d1,y1,'train1');
>> xyplot(d2,y2,'train2');
>> xyplot(dtest1,ytest1,'test1');
>> xyplot(dtest2,ytest2,'test2');
```

When the network har a large learning capacity, there is always the risk of overlearning. If you have time, try reducing the number of units to see if it gets better or worse. You can also test what happens if you low-pass filter the input to get rid of some noise during training.

● Did you try these improvements? Did it help?

Good luck!