Some exercises on Multi-layer perceptrons

Abstract

In this exercise we will study:

- Multilayer perceptrons (MLP) and for classification and time series problems (function approximation).

The problems we will look at are both artificial and real world problems. The environment you will work in is Matlab, specifically the Neural Network Toolbox.

The first section contains a description of the data sets used. Section 2 contains some information about the Matlab ANN toolbox. Section 3-6 contains the actual exercises.

Contents

1 Data sets

Various data sets will be used in this exercise, most of them coming from classification problems.

First a word about the notation. Following Matlab's own naming, `P` (or `Pt, Pv`) will be used for the matrix that is storing the inputs to the neural network. If there are 3 input variables and 200 data points the `P` will be a 3x200 matrix (the variables are stored row-wise). The targets are stored (row-wise) in `T` (or `Tt, Tv`). Next follows a description of the data sets.

1.1 2D AND and XOR

These problems are included in order to test the perceptron classifier. The XOR logic is interesting from a historical point of view since it was used to demonstrate the limitation of the simple perceptron. Table 1 shows the (familiar) definitions of AND and XOR.

The AND logic

x_1	x_2	target
0	0	0
1	0	0
0	1	0
1	1	1

The XOR logic

x_1	x_2	target
0	0	0
1	0	1
0	1	1
1	1	0

Table 1: The AND and XOR logic.

1.2 Synthetic data

3 different synthetic classification problems will be used. They are all 2-dimensional which allows for a visual inspection of the different classes. Most of them are drawn from Gaussian distributions with different widths and locations and the Matlab functions `loadsyn1, loadsyn2` and `loadsyn3` can be used to generate the data sets. For example

```
» [P,T] = loadsyn1(N);
```

will generate the first synthetic data set (N/2 of class 0 and N/2 of class 1). The data set can be visualized with the command

```
» plot2d2c(P,T);
```

1.3 Wood data

This is a real world data set which originates from a sawmill. The problem is to separate black and "ugly" twigs from the ones that you can't see. The inputs are different measurements on the twig. The file wood.dat (303 data) contains measurements from the twig according to: The first column contains the targets (1 if black and ugly otherwise 0), the second column describes its form and column 3-4 are reflections at different wavelengths. The last columns (5-6) are size measurements. You can load the data by `[P,T] = loadwood;`. The file `woodv.dat` contains additional 151 data that we will use for validation and can be loaded by `[Pv,Tv] = loadwoodval;`.

Since the wood data is 5-dimensional it's not easy to visualize it, but one can (perhaps) get a feeling for the data set by looking at 2 components only. For example, to look at inputs 1 and 2 you can type:

```
» [P,T] = loadwood; plot2d2c(P([1 2],:),T);
```

1.4 Sunspot data

In 1848 the Swiss astronomer Johann Rudolph Wolf introduced a daily measurement of sunspot number[1]. His method, which is still used today, counts the total number of spots visible on the face of the sun and the number of groups into which they cluster, because neither quantity alone satisfactorily measures sunspot activity. An observer computes a daily sunspot number by multiplying the number of groups he sees by ten and then adding this product to his total count of individual spots. However, the results vary strongly, since the measurement strongly depends on observer interpretation and experience and on the stability of the Earth's atmosphere above the observing site. Moreover, the use of the Earth as a platform from which to record these numbers also contributes to their variability, because the sun rotates and the evolving spot groups are distributed unevenly across solar longitudes. To compensate for these limitations, each daily international number is computed as a weighted average of measurements made from a network of cooperating observatories. Today, much more sophisticated measurements of solar activity are made routinely, but none has the link with the past that sunspot numbers have.

The data is available in the file `sunspot.dat`, which contains measurements from 1700-2012, and can be plotted by the command

```
» plotsunspot
```

1.5 Mackey-Glass data

A known example of a dynamic process is the so called Mackey-Glass process, which originates from a model of the amount of CO_2 in the blood. An often used variant of the Mackey-Glass time series is described by the differential equation,

$$\frac{dx}{dt} = -0.1x(t) + \frac{0.2x(t - t_{del})}{1 + x^{10}(t - t_{del})} \tag{1}$$

A common choice for t_{del} is 17. The Mackey Glass time series $x(t) \quad t = 1, ...1500$ is available in the file `mg17.dat`. We will however only use the first 400 data points. The data can be plotted by the command

```
» plotmg17
```

2 The Neural Network Toolbox in Matlab

It is very simple to define and train a neural network using the neural network toolbox[2]. In principle you only need to call three different functions:

- feedforwardnet: This defines and initializes a feed forward network
- train: This one trains the network given some input and target data.
- sim: Finally, "sim" simulates the network, i.e. calculates the output given some input data.

You can always find out how the different Matlab functions work and what kind of parameters that you can set by using Matlab's `help` and `doc` functions. Typing `doc feedforwardnet` will give information about how to create new feed-forward networks. Below follows a summary of the functions and parameters that you most likely will have to change during this exercise. We will only look at the batch learning algorithms (i.e. where you present the whole data set for the

[1] Annual sunspot data can be found here:
http://www.quandl.com/SIDC-Solar-Influences-Data-Analysis-Center/SUNSPOTS_A-Sunspot-Numbers-Annual

[2] The scripts in this exercise should work for neural network toolbox version 7.0 and 8.0.

network and then update the weights, which is different from the online learning where weights are updated after each data point), which are implemented using the `trainXX` functions. There is also a help to the functions provided for this exercise. To access the full Matlab documentation use the command `doc`.

2.1 The `feedforwardnet` function

The `feedforwardnet` function creates a feed-forward backpropagation network. The Matlab help on this function says

```
FEEDFORWARDNET Feed-forward neural network.

  Two (or more) layer feed-forward networks can implement any finite
  input-output function arbitrarily well given enough hidden neurons.

  feedforwardnet(hiddenSizes,trainFcn) takes a 1xN vector of N hidden
  layer sizes, and a backpropagation training function, and returns
  a feed-forward neural network with N+1 layers.

  Input, output and output layers sizes are set to 0.  These sizes will
  automatically be configured to match particular data by train. Or the
  user can manually configure inputs and outputs with configure.

  Defaults are used if feedforwardnet is called with fewer arguments.
  The default arguments are (10,'trainlm').

  Here a feed-forward network is used to solve a simple fitting problem:

    [x,t] = simplefit_dataset;
    net = feedforwardnet(10);
    net = train(net,x,t);
    view(net)
    y = net(x);
    perf = perform(net,t,y)

  See also fitnet, patternnet, cascadeforwardnet.

   Reference page in Help browser
      doc feedforwardnet
```

The first argument is the number of hidden nodes and the second argument gives the type of training algorithm you will use. Here are some listed:

- `traingd`: Gradient descent backpropagation.
- `traingda`: Gradient descent with adaptive learning-rate backpropagation.
- `traingdm`: Gradient descent with momentum backpropagation.
- `traingdx`: Gradient descent with momentum and adaptive learning-rate backpropagation.
- `traincgb`: Powell-Beale conjugate gradient backpropagation.
- `traincgf`: Fletcher-Powell conjugate gradient backpropagation.
- `traincgp`: Polak-Ribiére conjugate gradient backpropagation.
- `trainbfg`: BFGS quasi-Newton backpropagation.

- `trainlm`: Levenberg-Marquardt backpropagation.
- `trainrp`: RPROP - Resilient backpropagation.

By default the activation functions in the hidden layers are `tansig` ($\varphi(x) = \tanh(x)$) and the activation function for the output layer is `purelin` ($\varphi(x) = x$). See below how to change these.

2.2 The `train` function

After having created a feed-forward neural network we must now train it. This is accomplished using the `train` function. Suppose you have your training data stored in `P` (inputs) and `T` (targets), then we simply use,

```
» net = train(net,P,T);
```

in order to train the network. One *epoch* is one full presentation of all training data to the minimization method. The number of epochs is a parameter that can be set before you start the training.

```
net.trainParam.epochs = 50;
```

will train for 50 epochs. When you are running the `train` command you will get a small command window from which you can plot the performance during training and other things. By default the `train` function divides the supplied data into validation and test sets. To avoid or control this one can modify the `net.divideFcn` parameter (see below).

2.3 The `sim` function

After the training session is over we can simulate the network (i.e. calculate the output given some input data). If `Pn` contains new data the network response is given by

```
>> Y = sim(net,Pn);
```

2.4 Misc. functions and options

Here are some parameters/functions you may want to change/use. Note! Most of the parameters below has to be set **after** you have called the `feedforwardnet` function, in order to have any effect.

Changing the activation functions To change the activation function for the different layer one have to modify the `transferFcn` parameter in the different layers. The following will set hidden activation to tanh() and the output activation function to the logistic.

```
>> net.layers{1}.transferFcn = 'tansig';
>> net.layers{2}.transferFcn = 'logsig';
```

Input/Output normalization By default, the `feedforwardnet` function normalizes both input data and output data. For classification problems it is not a good thing to normalize the output data, therefore one have to turn the output normalization off. This is done with the following statement.

```
>> net.outputs{2}.processFcns = {};
```

Note! Here we assume a one-hidden layer network.

Avoiding validation/test splits By default the `train` function divides the supplied data into a validation and test set. This can be avoided by setting,

```
>> net.divideFcn = '';
```

Now all data will be used for training. Look in the file `syn_mlp.m` to learn how to define your own validation data set using the setting

```
net.divideFcn = 'divideind';
```

Learning rate You can change the learning rate for the gradient descent method by,

```
» net.trainParam.lr = 'new value';
```

Momentum parameter The momentum parameter in gradient descent learning can be changed,

```
net.trainParam.mc = 'new value';
```

Epochs The number of epochs can be changed by,

```
net.trainParam.epochs = 'new value';
```

Performance goal The minimization procedure can be terminated when the error has reached a certain "goal". You can change that by,

```
» net.trainParam.goal = 'new value';
```

Default is 0.

Minimum gradient The minimization procedure will terminate if the calculated gradient falls below some level. The level is by default $1e-07$. To change is modify this parameter,

```
» net.trainParam.min_grad = 'new value';
```

If this one is set to zero the training will never stop because of a small gradient.

Regularization Matlab does not support that many regularization methods. Early stopping and simple weight decay is supported. In order to use weight decay you should modify the parameter,

```
net.performParam.regularization
```

By default it is set to 0, meaning no weight decay. Denote the regularization parameter with γ, then the error function is defined as follows:

$$E = (1-\gamma)mse + \gamma \sum_i \omega_i^2$$

"mse" is the usual mean squared error function.[3]. The closer to one the more regularization you get!

By default if you have defined a validation set the "early stopping" procedure has been turned on. You can turn it off by setting the parameter

```
net.trainParam.max_fail
```

to a large number (e.g. larger than the number of epochs run).

[3]Matlab does not implement the cross-entropy error that is the more natural choice for classification problems. One can of course still use the mean squared error function!

3 The McCulloch-Pitts Perceptron

In this section we will do some small (AND and XOR) experiments on the simple perceptron that is using a "sharp" activation function and the perceptron learning rule. Make sure that the necessary Matlab files are in your working directory.

3.1 AND and the XOR logic

The function `andxor_perc` trains a perceptron to perform either the AND logic or the XOR logic. Run,

```
» andxor_perc
```

Try both the AND and the XOR logic.

- *Exercise 1:* ***Why can't we learn the XOR logic?***

4 Multilayer Perceptrons for Classification

In this section we will use MLPs for classification. We will start by looking at the XOR logic and then concentrate on synthetic and Wood data sets.

4.1 XOR logic

The AND gate should be no problem for the MLP, since it can be solved by the simple perceptron. The XOR logic, on the other hand, needs a hidden layer since it's not linearly separable. With 2 hidden neurons we know that it is theoretically possible to solve the XOR logic, but it turns out to be a challenging problem for some learning algorithms. You can use the function `xor_mlp` in this exercise.

In the following exercises (2-4) use 2 hidden neurons for the XOR problem.

- *Exercise 2: Use ordinary gradient descent learning.* ***How often do you manage to learn the XOR logic?***
- *Exercise 3: Use gradient descent with momentum and variable learning rate.* ***How often do you manage to learn the XOR logic?***
- *Exercise 4: Use a second order method, like the Levenberg-Marquardt algorithm.****How often do you manage to learn the XOR logic?***
- *Exercise 5: Increase the number of hidden neurons.* ***Does this change the success rate of the methods you tried above?***

4.2 Synthetic Data

Here we will use the MLP on the synthetic data sets. You should use the `syn_mlp` function here. The size of the validation set is fixed to the relatively large number of 1000 in order to have an accurate estimation of the performance.

One useful and instructive function you can use to get a feeling for the network behavior is the `boundary` function. It plots the decision boundary that the network is implementing. For example, type

```
» boundary(net,Pt,Tt,500,0.5,0.03);
```

to find all points that has a network output $y \in [0.47, 0.53]$ on a 500 x 500 grid covering the input data. If you get lots of disconnected points, try increasing the grid resolution and/or the cut (0.03).

- *Exercise 6: Use synthetic data 2 (100 data points) and train a linear MLP to separate the two classes, i.e. use a single hidden node.* ***Why can you handle this problem with a single hidden node?***
- *Exercise 7: Use synthetic data 1 (100 data points) and train a linear MLP to separate the two classes, i.e. use a single hidden node.* ***What is the performance you get on the validation dataset?*** *Note: Use a fixed random seed for this exercise since you will compare with other runs later.*
- *Exercise 8: Use synthetic data 1 (100 data points) and train an MLP to separate the two classes. Use 2, 4, 8 and 10 hidden neurons on the same data set (i.e. use the same random seed as for exercise 7) and record the results on both the training and validation data sets.* ***How do the results compare to those obtained with the linear MLP?*** *The result on the validation set*[4] *can be seen during the training by pressing the "Performance" button on the command window.* ***Is there any sign of over-training when you increase the number of hidden neurons?***
- *Exercise 9: Use synthetic data 3 (100 data points) and train an MLP to separate the two classes. By looking at the data, one would expect to need at least 3 hidden neurons to separate the two classes.* ***Why?*** *Use the same data set (i.e. the same seed) in a few different runs to be able to compare the results, and examine* ***how many hidden neurons give the optimal classification result on the validation set?*** *You may add loops to the* `syn_mlp` *instead of manually checking each hidden layer size.*

4.3 Wood Data

From now on there aren't any Matlab functions available, you are supposed to make your own "Matlab networks". You can of course use the previous ones as templates, like `syn_mlp` in this next exercise. NOTE: In all of the remaining classification exercises you should avoid the rescaling of the output data that is the default behavior of the toolbox. Remember that you can do that by the following statement

```
» net.outputs2.processFcns = {};
```

The functions `loadwood` and `loadwoodval` returns the wood training and validation data respectively. They are used as,

```
» [Pt,Tt] = loadwood;
» [Pv,Tv] = loadwoodval;
```

- *Exercise 10: Train an MLP on the wood data and optimize your network with respect to the result on the validation data set.* ***(i) What is the best result you can get? (ii) What is the architecture for that network?***

5 Multilayer Perceptrons and Time Series

In this section we are going to use the MLP to predict time series. The problems you are going to look at are the Sunspot data and the Mackey-Glass data. Figure 1 and 2 shows the two time series and the parts that you are going to use for training and testing (you can also use `plotsunspot` and `plotmg17`).

[4] It's shown on the Matlab figure as the test set!

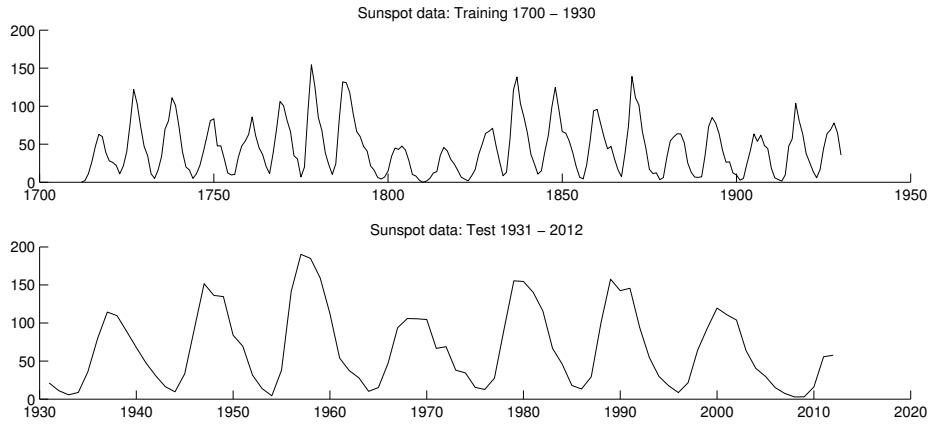

Figure 1: The sunspot time series.

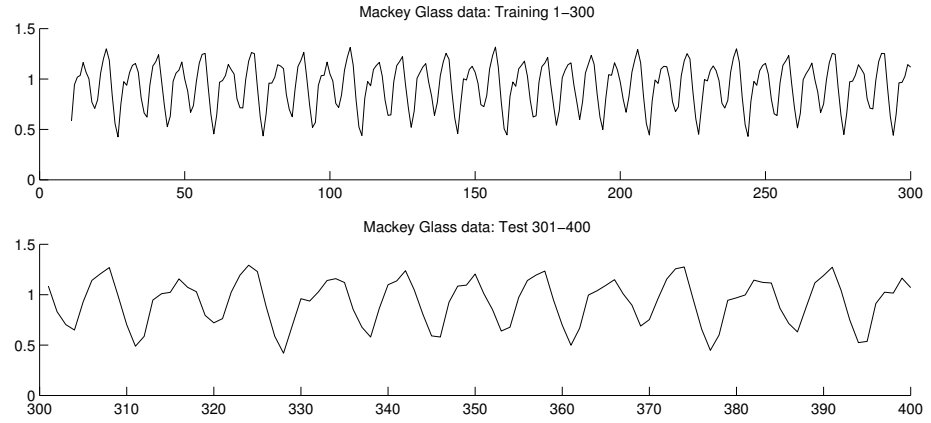

Figure 2: The Mackey-Glass time series.

How do we use an MLP to to time series prediction? The idea is to convert the time series prediction task into a function approximation task. Let's use the sunspot data as our example. The task is to predict the sunspot number $S(t)$ at year t, given as much as 12 previous sunspot numbers. Suppose you want to use 4 *time lags* $S(t-1), S(t-2), S(t-6), S(t-12)$ in order to predict $S(t)$; then you can construct the following data set from the time series.

Inputs				Targets
$S(1)$	$S(7)$	$S(11)$	$S(12)$	$S(13)$
$S(2)$	$S(8)$	$S(12)$	$S(13)$	$S(14)$
$S(3)$	$S(9)$	$S(13)$	$S(14)$	$S(15)$
.	.	.	.	.
$S(N-12)$	$S(N-6)$	$S(N-2)$	$S(N-1)$	$S(N)$

Now this can be viewed as a function approximation task, for which we can use an MLP.

For the last exercises there are 4 handy functions available, `loadsunspot`/`loadmg17` and `evalsunspot`/`evalmg17`. They are used in the following way:

```
[P,T] = loadsunspot(iset);
```

and `help loadsunspot` says

```
>> help loadsunspot
  function [P,T] = loadsunspot(iset)
  function [P,T] = loadsunspot(iset,sel)

  Returns the annual sunspot number.

  If iset=1, then the years 1700-1930 are returned (training set). If
  iset=2, then the years 1921-2012 are returned (test set).

  The optional sel parameter determines the time lags to use. Example, if sel
  = [1 2 6 12] then the returned P matrix will have 4 rows, corresponding to
  the sunspot number in year t-1, t-2, t-6 and t-12. If you don't provide a
  sel parameter P will contain 12 rows, corresponding to t-1,...,t-12.  The
  returned T vector has two rows. The first is the sunspot number at year t
  and the second row is the year t.

  Feb 2012, Mattias Ohlsson
  Email: mattias@thep.lu.se
```

`err = evalsunspot(net,scale,sel);`

and `help evalsunspot` says

```
>> help evalsunspot
  NMSE = evalsunspot(net,scale,sel);
  NMSE = evalsunspot(net,scale);

  This function computes Normalized Mean Squared Errors for both single step
  predictions and iterated predictions for the Sunspot data, using the
  supplied network "net" and a dummy model. The dummy model just says
  that the next value is the same as the last value, i.e. S(t+1)=S(t).
  The results are plotted in a figure and the errors are returned
  in NMSE as follows:

  NMSE(1) = NMSE for single step pred. on the training set
  NMSE(2) = NMSE for single step pred. on the test set
  NMSE(3) = NMSE for iterated pred. on the test set
  NMSE(4) = NMSE for the dummy single step pred. on the test set

  The scale parameter is used to scale the sunspot numbers, i.e. sn(t) ->
  sn(t)/scale. Use this if you did some rescaling before you trained your
  network.

  The sel variable are the selections of time lags you used
  during the training. E.g. if you trained you net using t-1, t-6, t-12
  then sel = [1 6 12]. If not supplied, 12 time lags are used.

  Feb 2012, Mattias Ohlsson
  Email: mattias@thep.lu.se
```

The corresponding functions `loadmg17` and `evalmg17` are used in similar ways. To make your own program using these functions, you will need to know how to access specific parts of matrices in Matlab. This is how it's done: given a matrix `A`,
`A(1,2)` returns the element in row 1, column 2;
`A(1,:)` returns all of row 1, whereas

`A(:,[2 3])` returns columns 2 and 3.
`max(max(A))` returns the greatest value of matrix `A` (useful when rescaling).

- *Exercise 11: Train an MLP on the sunspot data between 1700 and 1930 and make both single step and iterated predictions for the years 1931-2012 (using* `evalsunspot`*).*[5] *Try out different number of inputs, different architectures and perhaps weight decay in order to make as good predictions as possible.* ***Make a plot of the best single step prediction, and one plot of the best iterated prediction, and give details of the network used.***
- *Exercise 12: Repeat the above exercise for the Mackey-Glass data (only one plot required).* ***Why is this task much easier?***

[5] You may have to rescale the inputs to the network in order to get the learning to work properly.