Artificial neural networks - Exercise session 1

Supervised learning and generalization

2015-2016

1 The perceptron

The perceptron is the simplest one-layer network. It consists of R inputs connected to N neurons arranged in a single layer via weighted connections. These neurons have the `hardlim` transfer function, thus the output values are only 0 and 1, where the inputs can take on any value. The perceptron is used as a simple classification tool.

In order to create such a network, we can use the command

```
net = newp(PR,N)
```

where `PR` is a $R \times 2$ matrix of minimum and maximum values for the R inputs, and `N` is the number of neurons (or outputs). For instance

```
net = newp([-2 2; -2 2],1)
```

creates a perceptron with two inputs between -2 and 2, and one neuron (output).
To create a perceptron we can also use the command

```
net = newp(P,T,TF,LF)
```

where `P` and `T` are input and target vectors e.g. `P=[2 1 -2 -1; 2 -2 2 1]`, `T=[0 1 0 1]`. `TF` is the transfer function (typically 'hardlim'), `LF` represents the perceptron learning rule (for instance 'learnp'). In this case the number of neurons is set automatically.
The weights of the connections and the bias of the neurons are initially set to zero, which can be checked and changed with the commands

`net.IW{1,1}`	Returns the weights of the neuron(s) in the first layer
`net.b{1,1}`	Returns the bias of the neuron(s) in the first layer
`net.IW{1,1} = rand(1,2);`	Assigns random weights in [0,1]
`net.b{1,1} = rands(1);`	Assigns random bias in [-1,1]

What are these with curly brackets for? Are these some kind of special arrays? (Formal nomenclature please)

One can initialize the network with a single command by issuing:

```
net = init(net);
```

all weights and biases of the perceptron will be initialized to zero. See `help network/init` for more information on this function.

We can teach a perceptron to perform certain tasks which are defined by pairs of inputs and outputs. A perceptron can only learn examples perfectly and infer an underlying rule if they were generated by another perceptron. A perceptron can learn only linearly separable tasks: inputs belonging to different classes are separated by a hyperplane in the input space (decision boundary). In two dimensions the decision boundary is a line. The default learning function is the perceptron learning rule `learnp`.
The training function `adapt` updates the network online according to the provided examples:

```
[net,error,output] = adapt(net,P,T);
```


Is is necessery to be in input space, because, some data may not be linearly separable in input space, but may be linearly separable in higher dimentional spaces?
Seems perceptrons cannot learn to classify those datasets.

What is error? Difference between estimated value and output?

where P and T are input and target vectors respectively; e.g.[1] `P=[2 1 -2 -1; 2 -2 2 1]`, `T=[0 1 0 1]`, `output` contains outputs after training and `error` contains the corresponding errors. All examples will be presented once to the network, which adapts its weights and bias accordingly to each example. It is possible that such a single pass is not enough to obtain perfect classification, and thus multiple passes might be necessary. This can be done by executing the `adapt` command above several consecutive times, or (better) by setting the wanted number of passes in the network parameters prior to executing the `adapt` command:

```
net.adaptParam.passes = 20;
```

We can use also the function `train` to let the perceptron perform the classification task. In this case the learning occurs in batch mode:

```
[net,tr_descr] = train(net,P,T);
```

here the second argument is a description of the learning process.
To set the number of iterations or epochs, we can use the command

```
net.trainParam.epochs = 20;
```

Another learning rule that can be used to teach a perceptron is `learnpn` (normalized perceptron learning rule) which is useful in cases with input vectors differing very much in length. To generate a perceptron with a different learning rule use:

```
net = newp([-2 2; -2 2],1,'hardlim','learnpn');
```

Finally, after training we can simulate the network on new data with the function `sim`:

```
sim(net,Pnew)
```

with `Pnew` an input vector. For our example `Pnew` has to be a (column) vector of length 2, with elements between -2 and 2, e.g. `Pnew = [1;-0.3]`. Multiple input vectors can be fed at once to the network by putting them together in an array.

Demos

The following demos can be run from the MATLAB prompt. For the graphical demos follow the instructions on screen. For non-graphical demos you can run them interactively with the command `playshow name_of_demo`, and see and edit (a copy of) the code with `edit name_of_demo`

`nnd4db`	decision boundary (2d input)
`nnd4pr`	perceptron learning rule (2d input)
`playshow demop1`	classification with 2d input perceptron
`playshow demop4`	classification with outlier (2d input)
`playshow demop5`	classification with outlier using normalized perceptron learning rule (2d input)
`playshow demop6`	linearly non-separable input vectors (2d input)

Exercises

- Classification:
  Create a perceptron and train it with examples (see demos). Visualize results. Can you always train it perfectly?
- Learning from another perceptron:
  To be sure that a task is learnable we can generate examples with another perceptron, let's say `nett` (teacher). Teach a student perceptron `nets` using examples generated by `nett`. Use different ways of generating examples. Use different learning algorithms
  Use the demo `perts` (available in the Exercise session 1 folder on Toledo) as an example (`playshow perts`).

[1] We can use both matrices or cell-arrays at this point. Since matrix notation is less involved we will prefer this notation over cell-arrays. The above example in cell-array notation would look like: `P={[2;2] [1;-2] [-2;2] [-1;1]}`, `T={0 1 0 1}`.

Functions and commands

`newp(PR,n)`	Creates a perceptron with `n` neurons (`n = size(T,1)`), `PR` contains ranges of inputs
`newp(P,T,TF,LF)`	Creates a perceptron with the right number of neurons, based on input values P, target vector T, and with transfer function `TF` and learning function `LF`.
`init(net)`	Initializes the weights and biases of the perceptron.
`adapt(net,P,T)`	Trains the network using inputs P, targets T and some online learning algorithm.
`train(net,P,T)`	Trains the network using inputs P, targets T and some batch learning algorithm.
`sim(net,Ptest)`	Simulates the perceptron using inputs Ptest.
`learnp, learnpn`	Perceptron and normalized perceptron learning rules
`hardlim`	Transfer function
`plotpv(P,T)`	Plots examples P with symbols depending on targets T, only for 2 and 3 dimensions
`plotpc(W,b)`	Plots decision boundary for 2d and 3d perceptron with weights W and bias b

2 Backpropagation in feedforward multi-layer networks

A general feedforward network consists of at least one layer, and it can also contain an arbitrary number of hidden layers. Neurons in a given layer can be defined by any transfer function. In the hidden layers usually nonlinear functions are used, e.g. `tansig` or `logsig`, and in the output layer `purelin`. The only important condition is that there is no feedback in the network, neither delay.
In MATLAB one can create such a network object by e.g. the following command:

```
net = feedforwardnet(numN,trainAlg);
```

This will create a network of one hidden layer with corresponding `numN` neurons, which will use the trainAlg algorithm for training (e.g. `traingd`). This network can be trained using the `train` function:

```
net = train(net,P,T);
```

Finally the network can be simulated in two ways:

```
sim(net,P);
```

or,

```
Y = net(P);
```

Some available training algorithms are:

`traingd`	gradient descent
`traingda`	gradient descent with adaptive learning rate
`traincgf`	Fletcher-Reeves conjugate gradient algorithm
`traincgp`	Polak-Ribiere conjugate gradient algorithm
`trainbfg`	BFGS quasi Newton algorithm (quasi Newton)
`trainlm`	Levenberg-Marquardt algorithm (adaptive mixture of quasi Newton and steepest descent algorithms)

To analyze the efficiency of training one can use the function `postreg` which calculates and visualizes regression between targets and outputs. For a network `net` trained with a sequence of examples `P` and targets `T` we have:

```
a=sim(net,P);
[m,b,r]=postreg(a,T);
```

where `m` and `b` are the slope and the y-intercept of the best linear regression respectively. `r` is a correlation between targets `T` and outputs `a`.

Demos

`nnd11nf`	network function
`nnd11bc`	backpropagation calculation
`nnd11fa`	function approximation
`nnd11gn`	generalization
`nnd12sd1`	steepest descent backpropagation
`nnd12sd2`	steepest descent backpropagation with various learning rates
`nnd12mo`	steepest descent with momentum
`nnd12vl`	steepest descent with variable learning rate
`nnd12cg`	conjugate gradient backpropagation
`nnd12m`	Marquardt backpropagation
`nnd9mc`	comparison between steepest descent and conjugate gradient

Exercises

- Function approximation: comparison of various algorithms:
  Take a simple nonlinear function and try to approximate it using a neural network with one hidden layer. Use different algorithms. How does gradient descent perform compared to other training algorithms?
  Use following examples as a basis:

`algorlm1`	Script that compares the performance of Levenberg-Marquardt 'trainlm' and simple batch steepest descent 'trainsd' algorithms
`algorlm1`	The same as `algorlm1` but with smaller number of figures

  You can generate some nonlinear functions in the following way:

```
x=-3:0.2:3;        y=tanh(x);
x=0:0.2:3*pi;      y=sin(x);
x=-pi:0.05:pi;     y=exp(-x.^2).*sin(10*x);
x=0:0.1:3*pi;      y=sin(x).*sin(5*x);
x=0:0.05:3*pi;     y=sin(x.^2);
x=0.01:0.05:3*pi;  y=log(x).*sin(x).*sin(x.^2);
x=-pi:0.05:pi;     y=sign(x).*sign(x-1).*sign(x+1).*sign(x-2).*sign(x+2).*sin(2*x).*log(x+4);
```

- Learning from noisy data: generalization
  The same as in the previous exercise, but now add noise to the data (small random numbers to each datapoint). Compare the performance of the network with noiseless data. You may have to increase the number of data.

3 Report

Based on the previous exercises of section 2, write a report of maximum 2 pages (including text + figures) to discuss speed, overfitting, generalization of different learning schemes.

References

[1] H. Demuth and M. Beale, Neural Network Toolbox (user's guide),
`http://www.mathworks.com/access/helpdesk/help/toolbox/nnet/nnet.shtml`