Artificial neural networks - Exercise session 1

Supervised learning and generalization

2018-2019

1 The perceptron

The perceptron is the simplest one-layer network. It consists of R inputs connected to 1 neuron arranged in a single layer via interconnection weights. 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(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]

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.

We can teach a perceptron to perform certain tasks which are defined by pairs of inputs and outputs. A single 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`.

We can use 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;
```

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.

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

Exercises

Create a perceptron and train it with examples (see demos). Visualize results. Can you always train it perfectly?

Functions and commands

`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

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 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
`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
`nnd9mc`	comparison between steepest descent and conjugate gradient

Exercises

- Function approximation: comparison of various algorithms:
  Take the function $y = \sin(x^2)$ for $x = 0 : 0.05 : 3\pi$ 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` (can be found on Toledo): Script that compares the performance of Levenberg-Marquardt 'trainlm' and quasi-Newton backprogration 'trainbfg' algorithms.

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

- Personal Regression example:

  - **Problem Description and data preparation:** In this problem, the objective is to approximate a nonlinear function using a feedforward artificial neural network. The nonlinear function is unknown, but you are given a set of 13 600 datapoints uniformly sampled from it.

    You have to build your individual dataset from 5 existing nonlinear functions $f_1, f_2, \ldots, f_5$. In the given matlab datafile, you will find the following variables: $X1, X2, T1, T2, T3, T4$ and $T5$. The vectors $X1$ and $X2$ contain the input variables (in the domain $[0, 1]\times[0, 1]$). The vectors $T1$ to $T5$ are the 5 independent nonlinear functions evaluated at the corresponding point from $(X1, X2)$. In other words, $f_1(X1(i), X2(i)) = T1(i)$, $f_2(X1(i), X2(i)) = T2(i)$, ..., $f_5(X1(i), X2(i)) = T5(i)$, for $i = 1, \ldots, N$, where N is the length of the vectors mentioned above. The datapoints are noise free (the evaluation is exact).

    You have to build a new target $Tnew$, which represents an individual nonlinear function to be approximated by your neural network. For this, consider the largest 5 digits of your student number in descending order, represented by d_1, d_2, d_3, d_4, d_5 (with d_1 the largest digit). You have to build your individual target as follows:

    $Tnew = (d_1T1 + d_2T2 + d_3T3 + d_4T4 + d_5T5)/(d_1 + d_2 + d_3 + d_4 + d_5)$.

    For example, if your student number is m0224908, then your list of largest digits in descending order is 9, 8, 4, 2, 2 and therefore your target is:

    $Tnew = (9T1 + 8T2 + 4T3 + 2T4 + 2T5)/(9 + 8 + 4 + 2 + 2)$.

– **Exercises:** The problem can be split into three tasks:

1. Define your datasets: your dataset consists now of $X1, X2$ and $Tnew$. Draw 3 (independent) samples of 1 000 points each. Use them as the training set, validation set, and test set, respectively. Motivate the choice of the datasets. Plot the surface of your training set.
2. Build and train your feedforward Neural Network: use the training and validation sets. Build the ANN with 2 inputs and 1 output. Select a suitable model for the problem (number of hidden layers, number of neurons on each hidden layer). Select the learning algorithm and the transfer function that may work best for this problem. Motivate your decisions. When you try different networks, clearly say at the end which one you would select as the best for this problem and why.
3. Performance Assessment: evaluate the performance of your selected network on the test set. Plot the surface of the test set and the approximation given by the network. Plot the error level curves. Compute the Mean Squared Error on the test set. Comment on the results and compare with the training performance. What else could you do to improve the performance of your network?

3 Understanding Bayesian Inference: the Case of Network Weights

For simplicity of exposition, we begin by considering the training of a network for which the architecture is fixed in advance. More precisely we focus on the case of a one-neuron network. In the absence of any data, the distribution over weight values is described by a prior distribution denoted $p(w)$, where w is the vector of adaptive weights (normally also biases, but in our example we will consider the case without bias). We also denote by D the available dataset. Once we observe the data, we can write the expression for the posterior probability distribution of the weights using Bayes theorem:

$$p(w|D) = \frac{p(D|w)p(w)}{p(D)} \tag{1}$$

where $p(D)$ is a normalization factor ensuring that $p(w|D)$ gives unity when integrated over the whole weight space. $p(D|w)$ corresponds to the likelihood function used in maximum likelihood techniques.
Since in the beginning we do not know anything about the data, the prior distribution of weights is set to (for example) a Gaussian distribution. The expression of the prior is then:

$$p(w) = (\frac{\alpha}{2\pi})^{\frac{W}{2}} \exp(-\frac{\alpha}{2}||w||^2) \tag{2}$$

with W the number of weights, and α is the inverse of the variance. For simplicity we will choose $\alpha = 1$.
This choice of the prior distribution encourages weights to be small rather than large, which is a requirement for achieving smooth network mappings. So when $||w||$ is large, the parameter of the exponential is large, and thus $p(w)$ is small (small probability that this is the correct choice of weights. Things are reversed for small $||w|$).

A concrete example: binary classification We consider the case where input vectors are 2-dimensional $x = (x_1, x_2)$, and we have four data points in our dataset as in figure 1. We take a network of a single neuron (compare to the perceptron of the first exercise session), so there is only a single layer of weights, and choose the logistic function as transfer function:

$$y(x; w) = \frac{1}{1 + \exp(-w^T x)} \tag{3}$$

The weight vector $w = (w_1, w_2)$ is two-dimensional and there is no bias parameter. We choose a Gaussian prior distribution for the weights (with $\alpha = 1$). This prior distribution is plotted in figure 2.
The data points can belong to one of the two classes (cross or circle), the output y giving the membership to one of these classes. The likelihood function $p(D|w)$ in Bayes theorem will be given by a product of factors, one for each data point, where each factor is either y or $(1 - y)$ according to whether the data point belongs to the first or the second class.

First we consider just the points labeled (1) and (2). Then we consider all four points and recompute the posterior distribution of the weights. Note: For an example of this case, run demo `bayesNN`. After training with only the first two data points, we see that the network function is a sigmoidal ridge (w_1 and w_2 control the orientation and the slope of this sigmoid). Weight

vectors from approximately half of the weight space will have probabilities close to zero, as they represent decision boundaries with wrong orientation.When using all 4 data points, there is no boundary to classify all 4 points correctly. The most probable solution is the one corresponding to the peak point of the sigmoid, the others having quite low probabilities (so the posterior distribution of the weights is relatively narrow).

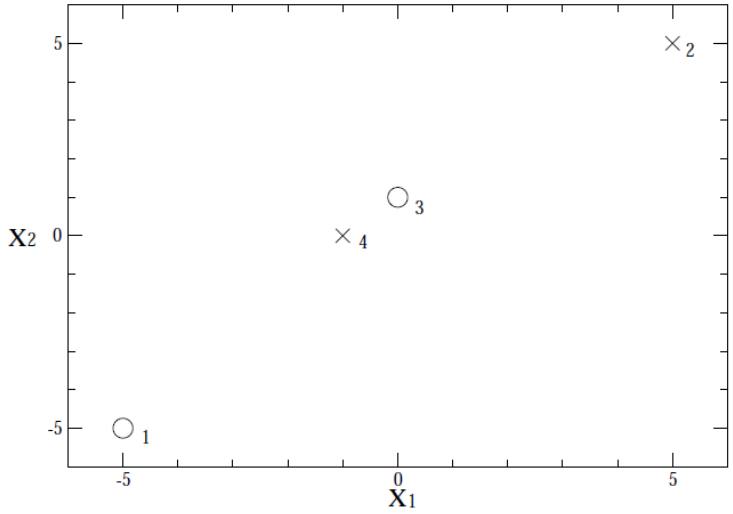

Figure 1: The four numbered data points in the dataset.

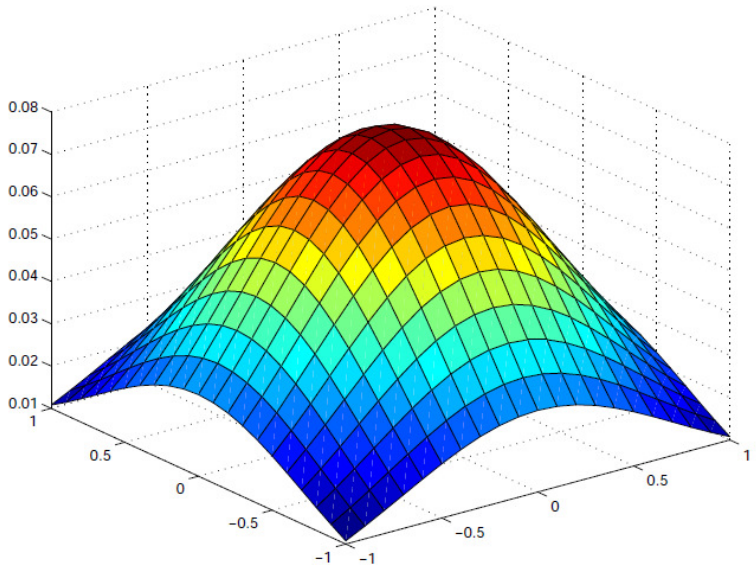

Figure 2: The prior distribution: a Gaussian.

4 Bayesian Inference of Network hyperparameters

From an optimization point of view the training is performed by iteratively adjusting w so as to minimize an objective function $M(w)$. In a maximum likelihood framework, M is taken to be the error function $E_D(w)$ that depends on the data D. A common prescription to avoid overfitting is to include in M a regularization term $E_W(w)$ (a.k.a. weight decay) to favor small values of

the parameter vectors, as discussed above. In particular, if we let

$$M(w) = \beta E_D(w) + \alpha E_W(w) \tag{4}$$

we can immediately give to this a Bayesian interpretation. In fact it is not difficult to show that βE_D can be understood as minus the log likelihood for a noise model whereas αE_W can be understood as minus the log prior on the parameter vector. Correspondingly, the process of finding the optimal weight vector minimizing M can be interpreted as a Maximum a Posteriori (MAP) estimation. Notice that we have implicitly considered α and β fixed here. However these are hyperparameters that control the complexity of the model. Once more within a Bayesian framework one can apply the rules of probability and find these parameters accordingly. In MATLAB for a general feedforward neural network this can be accomplished by means of `trainbr`. This training function updates the weight according to Levenberg-Marquardt optimization. It minimizes (4) with respect to w and determines at the same time the correct combination of the two terms so as to produce a network that generalizes well.

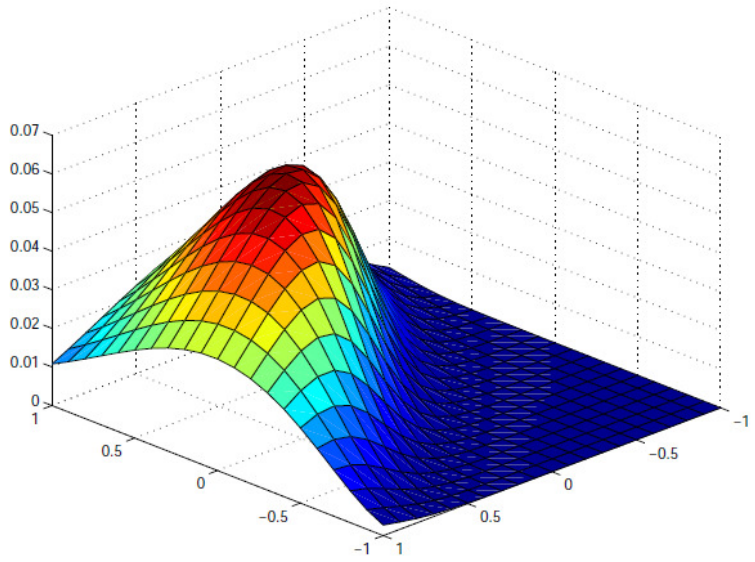

Figure 3: The distribution after presenting all four data points. Half of the weight space has become very improbable.

5 Exercises

- Use `trainbr` to analyse the first two datasets of Exercise Session 1, section 2 (the function $y = \sin(x^2)$ and the noisy version) and compare it with the training functions seen during the first exercise session. Compare the test errors. Consider overparametrized networks (many neurons): do you see any improvement with `trainbr`?

6 Report

Based on the previous exercises of section 2 and 5, write a report of maximum 3 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`

[2] C. M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press.

[3] Lecture slides and references therein.