NeuralMachine : neural network tool

Version 2.0 (October 2004)

NeuralMachine is a general purpose data-driven modelling tool which runs under Windows operating environment. It makes it possible to solve two types of problems:

- numerical (real-valued) function approximation (regression)
- classification.

Introduction

This software implements four major learning methods:

Artificial neural networks:
 (1) Multilayer Perceptron (MLP) – for regression and classification;
 (2) Radial Basis Function (RBF) networks – for regression;
Instance-based learning:
 (3) k-nearest neighbor method – for classification;
 (4) locally weighted regression – for regression.

The learning methods are often referred as "models" since this what they do – build models based on data. The first two methods (models) belong to the class of artificial neural networks (ANN). The other two methods belong to the class of instance-based learning. In case of MLP bias weig[illegible]nd direct input-output link are allowed. Nodes could be of sigmoid type and linear. The map[illegible] function of the RBF is of Gaussian type for the hidden layer and linear for the output layer. The software is referred to as a "neural network tool" since it was initially designed to buil[illegible]ral network models only, and still the neural network is seen as the primary method for data-[illegible]en modelling.

Full cycle of modelling, including training, cross-validation and verification (testing) is supported. The training, cross-validation and verification data sets could be made out of a single file, or supplied separately in different files. Several file formats are supported including Excel and ARFF format.

Important feature of this program (full version) is that after training, it allows for generating an executable file that encapsulates the trained network.

Trial (unregistered) and the full versions of NeuralMachine

The trial version is free to download and distribute. It has limitations on the following:

- saving the training and verification results
- generating the EXE file
- time limitation

(the previously existing limitation on the number of examples has been removed).

See Help/About on how to obtain the licence for the full version.

What NeuralMachine can and cannot do

What it can do:

- to solve function approximation problems (also called numerical prediction, or regression) for real-valued input data
- to solve classification problems (that is with nominal outputs) for real-valued input data
- to use ANNs (MLP and RBF), k-nearest neighbor method and weighted linear regression
- to read a number of file formats, including MS-Excel, comma-separated, matrices from MATLAB and ARFF
- to handle missing datato allow for keeping the training, cross-validation and verification data either in one file, or in three separate files
- to display the process of training in a clear graphical way
- to save in a file the calculated output for the newly supplied input examples
- to generate reports on the performance of the trained and verified networks

What it cannot do, but what is planned to implement gradually in the upcoming releases:

- to use other classification algorithms (eg., Bayesian, or decision trees)
- adaptively control MLP training by automatic adjustment of learning parameters
- handle close-to-singular regression problems that sometimes appear during RBF training.

Quick start: create and train your first neural network

Opening an existing project

Click "Open existing project" button that you see in the middle of the screen. Choose project SinCos79 from the Demos folder which is normally located in C:\Program Files\NeuralMachine. The project data contains information about the data sets and the model to be used – in this case it is a multilayer perceptron (MLP) artificial neural network. Now click "Train model" button on the toolbar.

Your first artificial neural network (ANN) is now being trained. The plots display the progress of training. ANN is trying to approximate the data presented by 79 values generated by sine and cosine functions (only sine is displayed). If you wait several seconds the fit will be almost ideal.

Click "Stop" button. The neural network is now trained.

You can now check its performance by clicking the 'Verify' button on the 'Training and verification' tab sheet. The appearing plot compares 12 output values calculated from the given inputs to the target ("measured") ones (which were not used to train ANN.)

Creating a new project

Click the ‘Data’ button on the left toolbar and then click ‘Close project’. Now you can start the new project by clicking the 'Start new project' button in the main window. (You can also select the 'Project/New project' menu option.) You will be taken through the process of creating a new project file. If you however, accept most of the defaults, you will have to specify only the names of the training and verification files and the model type. Model type is selected on the form that appears after you click the 'Set model' button. There is a choice of three algorithms: MLP, RBF and k-Nearest neighbor (instance-based learning).

In order to get a feeling of how the software works, you may want to select any of the existing data files from the Demos folder that is normally located in C:\Program Files\NeuralMachine.

Save this project under the name TEST in the folder where the training data file is located. Now you can click the Train button and perform training and then verification.

Note that next time you start NeuralMachine it will "remember" the folder where the training file was read from.

Opening a NeuralMachine project by double-click

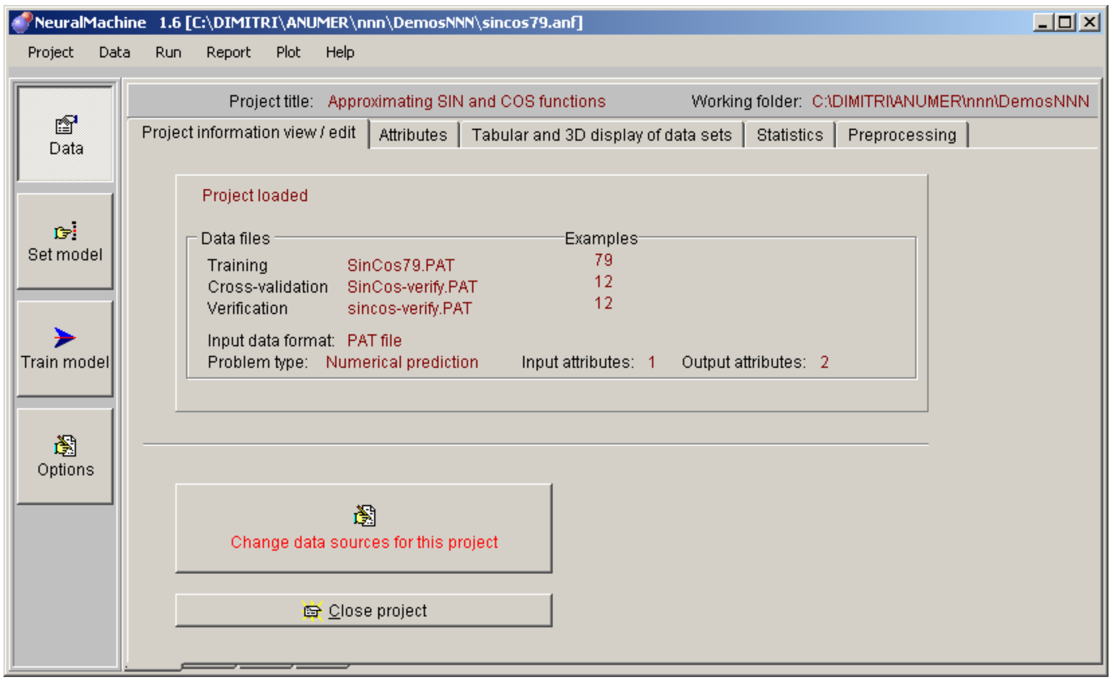
You can simply double-click on a project (ANF) file in Explorer. NeuralMachine will start and the project will be open.

Exiting NeuralMachine

If you have had enough of all this, choose 'Project/Exit' from the main menu.

Main Menu, Button Bar and modelling flow

The user interface of NeuralMachine follows the generally accepted norms. There are two ways to access the main functions: via the Main Menu on the top bar and the Button Bar on the left side of the main form.


NeuralMachine 1.6 [C:\DIMITRI\ANUMER\nnn\DemosNNN\sincos79.anf]
Project
Data
Run
Report
Plot
Help
Data
Set model
Train model
Options
Project information view / edit
Attributes
Tabular and 3D display of data sets
Statistics
Preprocessing
Change data sources for this project
Close project

Project title: Approximating SIN and COS functions   Working folder: C:\DIMITRI\ANUMER\nnn\DemosNNN

Project loaded

Data files		Examples
Training	SinCos79.PAT	79
Cross-validation	SinCos-verify.PAT	12
Verification	sincos-verify.PAT	12

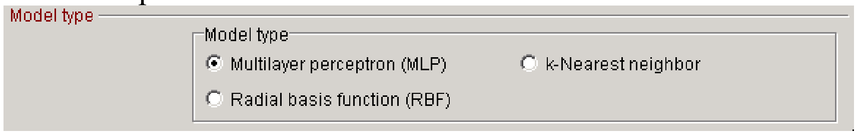
Input data format: PAT file
Problem type: Numerical prediction   Input attributes: 1   Output attributes: 2

NeuralMachine makes it easy to visualize the intermediate results while the network training is in progress. The user can easily access the project, pattern and result files while he/she is in the same application environment. It is also possible to see the plots which shows the performance of the network on the training and cross-validation data.

The main menu (top bar) is self-explanatory.

The Button Bar on the left side of the main form provide the easier and more comprehensive access to the main functions. These buttons are associated with the forms that appear on the right-hand side, and will be referred as "Button/Form":

> *Data Button / Form* displays the project data and giving access to the editors of project data, data sources and attributes. Functions controlled from this form partly correspond to the Project and Data menu options.
>
> *Set model Button / Form* allows for selecting the model type (one of the three) and setting the model parameters.


Model type

Model type
- [✓ Multilayer perceptron (MLP)]
- k-Nearest neighbor
- Radial basis function (RBF)

> *Training model Button / Form* provides control of the training process, cross-validation, verification and reporting. Functions controlled from this tab presented on this tab correspond to the Run, Report and Plot menu items.
>
> *Options Button / Form* allows for control of the display parameters.

The modeling flow, that is arranging data, setting the model, and running (training) it is in fact represented by the order of the first three buttons in the Button Bar.

Project file and other associated files

Project file

Project file (with ANF extension) contains names of the training, cross-validation and verification files, data about the network type, learning parameters, about attributes, etc. It is a text file so in principle you can edit it directly. It is advised, however, to use the Data Button/Form.

Editing project data

The Data Button/Form allows for editing all the data associated with the current project. It has several tab sheets for (1) project information, (2) attributes, (3) tabular and 3D display, (4) statistics, (5) preprocessing.

Note that the changes you make are stored in memory. When you exit the system you will be prompted to save the changes to disk.

Other files

In the process of working with ANNs NeuralMachine creates several files. They are named with respect to their file extensions:

WEI
Weights file contains the trained weights of ANN and the parameters values that were used for this particular run. Is always created after Pause or Stop button is clicked. Its name is the same as the name of the training file (but has different extension)

RPT
(only in full version). Report file containing the calculated values in training and verification. It is created when Verification button is clicked.

OUT
The calculated values when run is performed for one example (which is normally stored in INP file).

Attributes, input data and their formats

Attributes and data sets

Attributes are variables (properties) that characterize the data.

NeuralMachine automatically recognises the real-valued and nominal attributes in input data (in delimited and ARFF files). For classification problems only one output nominal attribute is allowed.

> Note: Since the whole concept of MLP and RBF ANNs is based on real-valued data, often there is not much sense to use nominal input attributes. However, NeuralMachine allows for that: it represents such attributes as integers (class numbers). In this case the user should be careful in interpreting the results.

Training and verification files should have the same number of attributes. If you use input file format without headers, then the attribute names are generated automatically and have the form "Var 1", "Var 2", etc.

In order to train ANN it is enough to have one training file. However, in the process of neural network modelling more data sets are normally used.

In order to control the performance of the network during training, the *cross-validation* data set is needed. During training the performance on the cross-validation data can be viewed and training could be stopped when the network error on this sets starts to grow.

In order to verify (test) ANN after it is trained, the verification data set is used.

Note on using k-Nearest neighbor model

If k-Nearest neighbor model is selected, there is no training phase (however, the training data set has to be supplied of course). Clicking on 'Train' button presents a simplified screen with only the 'Run' button. Clicking on it runs the algorithm and presents the verification result.

Two ways to supply cross-validation and verification data

The way this data is supplied is indicated on the 'Data Button/Form'. There are two ways of supplying data.

1. If data is in a separate files, you have to simply specify the file names.

2. If all data is in one file, specify the name of the training file and then indicate the percentage split of this file into training proper, cross-validation and verification data subsets. Another way is to indicate the number of examples for each of the subsets, assuming that in the data file they follow in the following order: training, cross-validation, verification.

Supported data formats

Data files are text files of simple structure. They are organized as matrices, with each example occupying one line, and columns corresponding to attributes. Some attributes are the input attributes (independent variables) and some are outputs ones. Only PAT format contains the information on which attributes are the outputs. For other data formats this information is stored in the project (ANF) file.

NeuralMachine can read 4 types of input data files:

1. **PAT files**. In such file the first line contains three numbers:
number of inputs, number of outputs, number of examples (separated by spaces). Each consecutive line contains 1 example (pattern): first all input values, then all output (measured) values. The values can be only numerical. They have to be *separated by spaces or tabs*. An example follows:

```
5 2 1
1.2  2.3  5.6
2.3  3.4  6.7
1.5  2.8  6.1
1.6  2.9  6.3
1.9  3.4  6.8
```

Note that this is the only format that explicitly provides information about which attributes are the inputs and the outputs.

2. **Delimited text files without headers**. Standard extension is TXT; they have the same format as the PAT files but without the first line. Separator can be space, tab or comma. Example follows.

```
1.2  2.3  5.6
2.3  3.4  6.7
1.5  2.8  6.1
```

```
1.6  2.9  6.3
1.9  3.4  6.8
```

3. Delimited text files with headers. Standard extension is TXT, same as (2) but with the first line containing the columns' headers. Separator can be space, tab or comma. The values can be numerical or nominal.

```
Input1  Input2  "Output One"
1.2  2.3  5.6
2.3  3.4  6.7
1.5  2.8  6.1
1.6  2.9  6.3
1.9  3.4  6.8
```

If a header has the separator as its part, enclose it in double quotes, for example, in case of using a space separator:

```
Input1  Input2  "Output One"
```

4. ARFF files. ARFF is a format developed by the University of Waikato in New Zealand and is used in machine learning community. It includes the information about the attributes types. The values can be numerical or nominal. Example follows:

```
@relation SiQ3
@attribute Input1 real
@attribute Input2 real
@attribute OutputOne real
@data
1.2  2.3  5.6
2.3  3.4  6.7
1.5  2.8  6.1
1.6  2.9  6.3
1.9  3.4  6.8
```

The contents of the data files can be changed via the 'Data/Edit data files' menu option or by using an external text editor like Notepad.

> **Note**: Notepad saves files as TXT files by default. In order to use a different file extension you have to specify the filename in double quotes. To handle text files it is recommended to use more advanced tools like TextPad (www.textpad.com).

5. MS-Excel files without header. Data is arranged in a table. You will be prompted to specify the top-left cell of this matrix (A1 by default).

6. MS-Excel files with header. Data is arranged in a table. You will be prompted to specify the top-left cell of this matrix (A1 by default). The first row should contain the headers for the attributes.

For the formats 2, 3, 4, 5 and 6 the last attribute is assumed to be the output, but the user is prompted to confirm that. At any moment this choice can be easily changed via the

Data/Attributes menu option and via the Data tab and Edit project properties dialog box (see the Attributes section).

Some tips on saving MS Excel data as text

NeuralMachine is capable of reading Excel XLS files. However if the data structure you keep in Excel is complex and for some reason would like to save a subset in one of the non-Excel text formats, you can do the following. In Excel, use 'File/Save As' option and choose for the 'Save as type' field the *"Text (Tab delimited) (*.txt)"*. If you use spaces in a column header, enclose it in double quotes.

An alternative is to choose the *"CSV (comma delimited) (*.csv)"* file type. This is actually a preferred option. Do *not* use "Formatted text (space delimited)" file type (it may not leave delimiters at all).

Note that the PAT file requires space or tab as a separator, so to create PAT files always use the *"Text (Tab delimited) (*.txt)"* type option.

Maximum size of networks and data sets

The current version of NeuralMachine supports the following dimensions.

- maximum number of inputs: 30
- maximum number of hidden nodes: 30
- maximum number of outputs: 10
- maximum number of examples: 25000

You can see these values by selecting 'Help/About' menu option.

Selecting attributes to be used in the model

Choosing attributes to be used in training

It is possible to use only selected attributes in modelling. Under Data Button/Form select the Attributes tab sheet and 'All attributes' button. You can select which attributes will be further used in training.

Choosing output attributes

Under Data Button/Form select the Attributes tab sheet and 'Output attributes' button. You can select the output attributes.

The selection of the output attribute to be displayed on the plots (Y axis) is done via the Options Button/Form.

Handling data in classification problem

NeuralMachine can solve classification problems as well as regression ones. The following gives an idea how it is done.

MLP or RBF cannot directly handle the nominal data since they explicitly deal with real numbers. However if data is encoded in a particular way, ANN appears to be a very good classifier. NeuralMachine does the necessary data encoding automatically.

If the output data is nominal (class), it has be presented in the following form. The class output having N possible nominal values (coded for simplicity as integers) should be replaced by N "fictitious" outputs. After such transformation, an input vector that had corresponding output equal to *k*, will have N output values { 0 0 ... 1 ... 0 } where output *k* has value 1. Due to the character of the non-linear sigmoid function used in MLP networks the better choice would be to use values 0.9 and 0.1 instead of 1 and 0.

When the trained network is used the N generated real output values will be calculated. The output with the maximum value should be found and its index (say, *m*) is taken as code of the original class.

For example if the original training data has 2 numerical inputs and 1 class output (its value is an integer number 1, 2 or 3 representing one of three classes):

```
1.2  2.3   1
2.3  3.4   3
1.5  2.8   2
1.6  2.9   2
1.9  3.4   3
```

then the data should be represented in this form:

```
1.2  2.3   0.9  0.1  0.1
2.3  3.4   0.1  0.1  0.9
1.5  2.8   0.1  0.9  0.1
1.6  2.9   0.1  0.9  0.1
1.9  3.4   0.1  0.1  0.9
```

Then you have to build the ANN with 2 inputs and 3 outputs. After the ANN is trained you may want to use to determine the class (1, 2 or 3) of a newly coming input vector, for example of this one:

```
1.4  2.7
```

This vector has to be fed into the trained ANN and the calculated result could be this:

```
1.4  2.7    0.22  0.88  0.31
```

The number of the output with maximum value is 2. Hence the presented example belongs to class 2.

NeuralMachine handles such pre- and post-processing of data automatically. The output plot can display to see the values of the generated fictitious output attributes. For classification problems NeuralMachine displays also the confusion matrix which is normally used to judge about the classification errors.

Controlling training and verification of ANN (MLP and RBF)

The theoretical foundations of ANN training are covered in separate sections:

Training MLP networks

Training RBF networks

This section outlines the training and verification process as it is implemented in NeuralMachine.

The methods how MLP and RBF networks are trained differ. However, there are managed by the same controls - buttons Train, Pause and Stop.

Training

When the project file is open or created, you may proceed with training. This process is described in the section "Quick start".

On the Train model Button/Form you will find many other buttons and input fields allowing you to control the process of neural network modelling. They allow for example, for performing the steps-wise training or to introduce the delay after each training iteration. (The Set model Button/Form gives you the possibility to change the learning parameters.)

Avoiding local minima during training (MLP only)

A known problem in training ANN is that the training process is trapped in a local minimum. There are various ways of dealing with this problem. One of them is a random “jump” in the space of weights that leads away from such a local minimum. This can be done by slight random change of the weights. For that the Shake button on the Training and Verification tab can be used. The amount of such “shake” is controlled via the parameter accessible through the Edit project properties button on the Data tab (or via the 'Data/Edit project properties' menu option).

Pausing or stopping the training process

Training of an MLP network will stop automatically after 20000 iterations. The weights of the best configuration are saved in the weights file.

Training of RBF network stops after all configurations having from 1 to 30 hidden nodes were tested. The weights of the best configuration are also saved in the weights file.

Training can be stopped earlier if the required accuracy is reached, if no improvement is expected or if you want to change learning parameters. Both the training and verification errors are the important indicators of the network quality. In order to stop training, click 'Pause' or 'Stop' button. When this is done the results of training are saved to the WEI file that has the same name as the training data file.

If you clicked the 'Pause' button, the caption of the 'Training' button changes to 'Continue'. The training process can be resumed immediately by clicking the 'Continue' button.

If you clicked the 'Stop' button you can also resume training by first checking the 'Load weights and continue training' option and then clicking 'Train' button.

Changing the learning and other parameters

If you are not satisfied with the training progress you may want to change learning parameters. Normally, you have to pause or stop the process first. Then click on the 'Learning parameters' tab sheet and change the necessary parameters. After that continue or restart training.

For MLP networks there is a possibility to change 4 parameters during training (without pausing the training). On the 'Learning parameters' tab sheet you can change Kappa, Phi, Theta and Mu parameters incrementally by clicking the small up and down arrow controls or using the slide bars. In order to see the main plot all the time you have to set the corresponding "Detach" checkbox.

> Note. The changed parameters are not saved to project file automatically. If you want to save them, use the 'Data/Edit project properties' menu option.

After 'Pause' or 'Stop' button is clicked the ANN is considered to be trained (at least partly) and you can verify its performance.

Verifying (testing) the trained network

To verify (test) the trained network, click the Verify button on the Train model Button/Form. The chart will display the output calculated by the trained network on the basis of input data taken from the verification file. The plot will also display the target (measured) data that was not presented to ANN during training but was kept for verification only. The report will be also generated and put into the file that has the same name as the training file but extension RPT.

Viewing the report, weights and the network structure

To see the weight file obtained after training or the result file after verification test choose '**Report/View report**' option from the main menu and select the corresponding entry from the menu items. It is also possible to view INP and WEI files.

You can display the MLP networks structure via the '**Plot/Display network structure**' menu option.

Using the trained network for the new data example

To use the trained and verified network to calculated the output for a new input data example which the network might not have seen before, choose 'Run/Use Network for one example in INP file's option from the main menu. Then select or type the INP file name where the new input data is kept from the corresponding open dialog box displayed. In a similar way specify the file name for the resulting output. By this time the output of the network for the given input will be calculated and the result is displayed on the screen.

Before using this option you have to train the network, After training is paused or stopped the resulting weights will be saved in the file with the same name as the input data file (PAT, TXT or ARFF) used for training but with the WEI extension. When you select Run/Use Network menu option, this WEI file is loaded to calculate the output. This calculated values of the output attributes are placed in the OUT file.

The format of INP file is simply a sequence of numbers separated by spaces or placed on separate lines. Their number must be equal to the number of attributes in the training file. The format of OUT file is a sequence of numbers separated by spaces.

You can also encapsulate the trained network into an executables file and use it to calculate the output for a new example.

Encapsulation of the trained network into the EXE file (MLP only)

After training and verifying a network, it is possible to generate executable code that can be used as an independent fast and small executable. To do so, click Generate code button or use Run/Generate EXE file menu option. The application uses the weight file created after training that corresponds to the present input file. The generated executable file name is composed of the name of the training file complemented with the substring "-trained" and it has the extension EXE.

This executable can be then used as a standalone program which reads input from INP file specified on the command line and places the output to the newly generated OUT file having the same name as the INP file.

Demo projects included with the distribution

There are several demo projects included:

SinCos79.anf

First attribute is the independent variable running from 0 to 6.24 with step 0.05. Other two attributes (both outputs) are the values of sine and cosine of the first attribute. There are 79 data examples in training file and 12 in the verification file. This project refers to the data files Sincos79.pat and Sincos-verify.pat (in PAT format).

SinCos79-txt.anf

This project uses the same data as the previous one, but it is presented in the delimited file format with the named columns. This project refers to the data files Sincos79.txt and Sincos-verify.txt.

SinCos79-xls.anf

This project uses the same data as the previous one, but it is presented in the MS-Excel file format with the named columns. This project refers to the data file Sincos79.XLS (it contains all data).

SinCos.anf

A similar example, but the training file contains 315 examples and the data is in PAT format. You will be able to see that the ANN is trained on 315 examples faster than on 79 examples. This project refers to the data files Sincos.pat and Sincos-verify.pat. (Note that the unregistered version may not be able to handle that many examples.)

Cpu.anf

A widely used case study of estimating performance of various computers (CPUs) based on the memory, machine cycle time and other parameters.

Apure.anf

These project presents the data used in the hydrodynamic/hydrologic modelling of a catchment in Latin America. 25 inputs are comprised of runoffs from 21 subcatchments, 3 releases from reservoirs and the current water level. Output is the water level at a certain point downstream next week. The training file reflects data from years 1981-82, and the test file - from 1983. File Ap-train.pat with 104 examples is used for training, and file Ap-veify.pat with 53 examples – for verification. The projects differ in the format of the input data.

Apure-csv.anf

This project is analogous to the previous one but refers to the data files in delimited format (Ap-train.csv and Ap-verify.csv).

Ap-trainXLS.xls, Ap-verifyXLS.xls

These Excel files contain data used to generate the corresponding comma-delimited CSV files. You can create a project and use these files directly; they are given as an example of how data could be organized in Excel.

A bit of theory: what is a neural network?

Artificial neural networks

Originally studied in the framework of AI, *artificial neural networks (ANNs)* has become now one of the primary technologies in machine learning, and a mainstream technology for data-driven modelling. Various types of networks are used in clustering, classification and prediction. ANNs loosely imitate functioning of neurons in a human's brain, and it appeared it is possible to combine these neurons in such a way, that the network would reproduce any multi-variable multi-valued function, given enough points and values of this function. By analogy with the brain, the operation of the trained (learned) network is often called *recall.*

ANNs exhibit three features, namely, distributed processing, adaptation and nonlinearity, and it have been mathematically proven that adding up simple functions, as a ANN does, allows for universal approximation of functions (Kolmogorov 1957). This means that neural networks can approximate any function that best characterizes a time series. It is this property that has stimulated civil engineers to adapt, investigate, and improve the performance of neural networks associated with their applications.

Neural network is a universal function approximation method. It includes a large number of unknown parameters that are identified by solving an optimization problem. An efficient method of solving this problem in the context of neural networks is called a *backpropagation* algorithm.

NeuralMachine makes it possible to build two types of networks (covered in subsequent sections):

(1) multilayer perceptron (MLP) and
(2) radial basis function (RBF) network

More information on the Web

See Help/About for the reference to a Web site where you can find material on machine learning, data mining in general and on neural networks in particular.

Some references

As an introductory text on neural networks, the following book could be recommended:

M. Smith. Neural networks for statistical modelling. Van Nostrand Reyngold, N.Y., 1993.

More extended and advanced texts are:

S. Haykin. Neural networks: A Comprehensive Foundation. Prentice-Hall, 1999.

M.H. Hassoun, Fundamentals of Artificial Neural Networks. The MIT Press, 1995.

L.H. Tsoukalas and R.E. Uhrig. *Fuzzy and neural approaches in engineering*. Wiley, 1997.

Excellent book on various aspects of machine learning is:

Mitchell, T.M. *Machine learning*. McGraw-Hill, 1998.

Good reference on the issue of data preparation is:

D. Pyle. *Data preparation for data mining*. Morgan Kaufmann, 1999.

NeuralMachine (and its predecessor) was used in many practical problems. The relevant references are given below:

D.P. Solomatine, A. Avila Torres. Neural Network Approximation of a Hydrodynamic Model in Optimizing Reservoir Operation. Proc. 2nd International Conference on Hydroinformatics. Zurich, Switzerland, August 1996.

R. K. Price, J.N. Samedov and D. P. Solomatine. An artificial neural network model of a generalised channel network. Proc. 3rd International Conference on Hydroinformatics. Copenhagen, Denmark, August 1998.

Solomatine D.P. Two strategies of adaptive cluster covering with descent and their comparison to other algorithms. Journal of Global Optimization, 1999, vol. 14, No. 1, pp. 55-78.

Maskey, S., Dibike, Y. B. Jonoski, A. and D.P. Solomatine. Groundwater model approximation with artificial neural network for selecting optimal pumping strategy for plume removal, In: AI methods in Civil Engineering Applications (O. Schleider, A. Zijderveld, eds), Cottbus, 2000, pp. 67-80.

B. Bhattacharya, D.P. Solomatine. Application of artificial neural network in stage-discharge relationship. Proc. 4th Int. Conference on Hydroinformatics. Iowa, USA, July 2000.

Dibike Y.B., Velickov S., Solomatine D.P. and Abbott M.B. Model induction with support vector machines: introduction and applications. Journal of Computing in Civil Engineering, American Society of Civil Engineers (ASCE), vol. 15, No.3, 2001, pp. 208-216.

D.P. Solomatine. Computational intelligence techniques in modeling water systems: some applications. Proc. World Congress on Computational Intelligence, USA, May 2002.

Some of the papers you can download also at http://www.data-machine.com.

The following sections provide details of the MLP and RBF learning.

Multilayer perceptron (MLP): structure

The multilayer perceptron (MLP) is generally considered the most powerful and universally applicable type of ANN. The widely accepted way of training MLP is the backpropagation algorithm. There are various ways of updating weights in this algorithms; NeuralMachine uses the *adaptive learning rate* (the 4 corresponding parameters can be controlled by the user). The details of MLP structure and training are covered below.

Regression as a simplest neural network

A possibly simplest machine learning algorithm is linear regression equation that can be "trained" to reproduce the input-output relationship (that is the coefficients of the linear function can be found). Figure 1 shows a "network" made of one node with the linear processing element (PE).

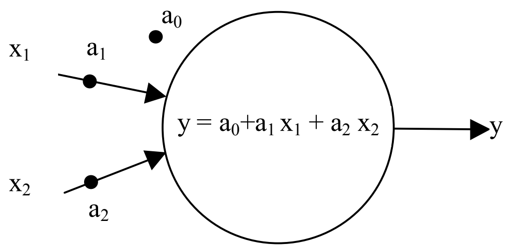


Figure 1. Processing element (PE) in linear regression (two inputs)

In one-dimensional case (one input *x*), given *T* vectors (initial data)

$$\{x^t,\ y^t\},\ t = 1,\ldots T$$

the coefficients if the equation

$$y = f(x) = a_1 X + a_2$$

can be found and then for the new V vectors

$$\{x^v\},\ v = 1,\ldots V$$

this equation can approximately reproduce the corresponding functions values

$$\{y^v\},\ v = 1,\ldots V$$

Adding complexity: layers and connections

The regression model presented above is not too accurate – it just presents a linear relationship, and most processes to be modelled are highly non-linear. In order to make the model nonlinear, it is necessary to introduce the nonlinear components in order to be to reproduce (approximate) much more complex relationships with several inputs and outputs (Figure 2).

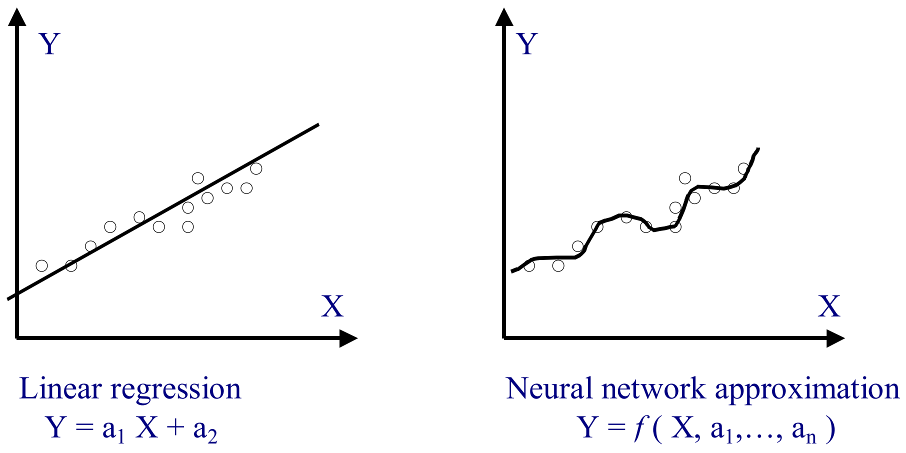


Figure 2. ANN vs linear regression model.

If N_{out} functions, each with N_{inp} independent (input) variables are given, and T instances (vectors)

$$\{ x_1^{(t)}, x_2^{(t)}, \ldots, x_{N_{inp}}^{(t)}, f_1^{(t)}, f_2^{(t)}, \ldots, f_{N_{out}}^{(t)} \},\ t = 1, \ldots, T$$

are given, then, on the basis of these instances ANN can be *trained*, so that, fed with calculated with other V instances (vectors)

$$\{ x_1^{(v)}, x_2^{(v)}, \ldots, x_{N_{inp}}^{(v)} \}, \quad v = 1, \ldots, V$$

it would approximately reproduce the corresponding functions values

$$\{ f_1^{(v)}, f_2^{(v)}, \ldots, f_{N_{out}}^{(v)} \}, \quad v = 1, \ldots, V$$

One type of the ANNs commonly used is *multi-layer perceptron neural network (MLP)*. It is made up of a number of interconnected nodes (called neurons, the same as PEs), arranged into three types layers: input, hidden (could be several of them) and output. The lines represent weighted connections between PEs.

The input layer in the Figure 3 does not perform any operation upon the input signal but simply sends the x_i to the units in the hidden layer A processing element simply multiplies input by a set of weights, and linearly or nonlinearly transforms the result into an output value. The result can be the input to other PEs in any layers except the input one. By adapting its weights, the neural network works toward an optimal solution based on a measurement of its performance.

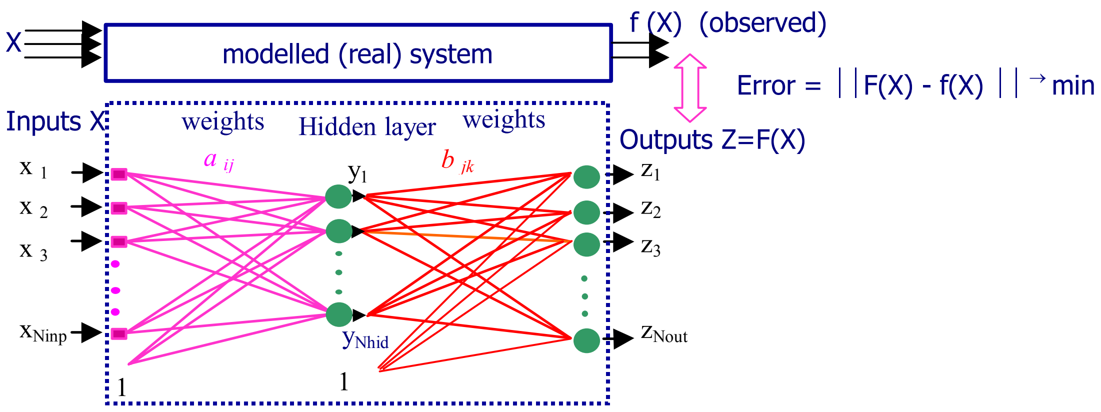


Figure 3. Multi-layer perceptron with one hidden layer

Biological motivation

In many ways the study of ANNs has been inspired by the observation that biological learning systems (for example, a human brain) are built of very complex networks of interconnected neuron (Figure 4).

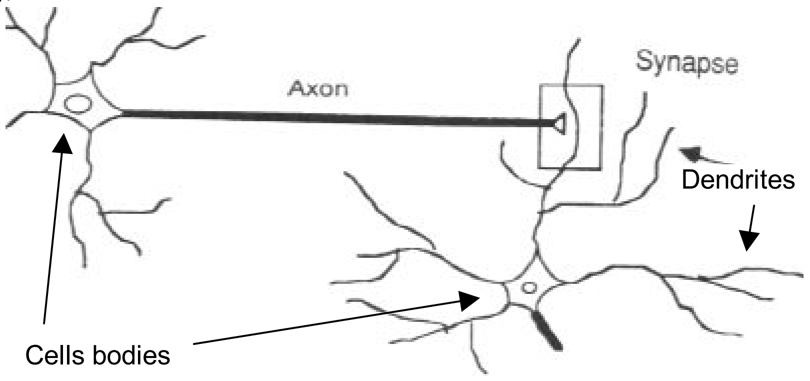


Figure 4. Connected neurons

Signals are transmitted between neurons by electrical pulses (spikes) travelling along the long thin stand called *axon*. These pulses are received by the receiving neuron at terminals called *synapses*. (They are found on a set of branches emerging from the cell body (soma) and known as *dendrites*). These pulses lead to certain chemical activity in the dendrites which may inhibit or excite the generation of pulses in the receiving neuron – this depends on the geometry of the synapse and type of chemical activity. The neuron sums up or integrates the effects of thousands of impulses over its dendritic tree and over time. If the integrated potential exceeds a threshold, the cell 'fires' and generates a spike which starts to travel along its axon. This then initiates the whole sequence of events further in the connected neurons.

Learning is a complex process of changing the effectiveness of the synapses so that the influence of one neuron on another changes. Research in ANN was inspired by neuroscience but did not attempt to be biologically realistic in detail – it simply appeared to be too difficult to achieve. As a result most ANNs are *connectionists* models combining simple *processing elements* (called also *neurons*, *units*, or *nodes*). Learning in ANN is typically in changing the strength of connections (*weights*) between neurons. In some types of ANNs neurons may have local memory.

Nodes and transfer functions

A typical node in a hidden layer is presented on Figure 5. It receives signals (values) from the nodes of the input layer and transforms them into signals which are sent to all output nodes (Figure 6); which, in turn, transform them into outputs.

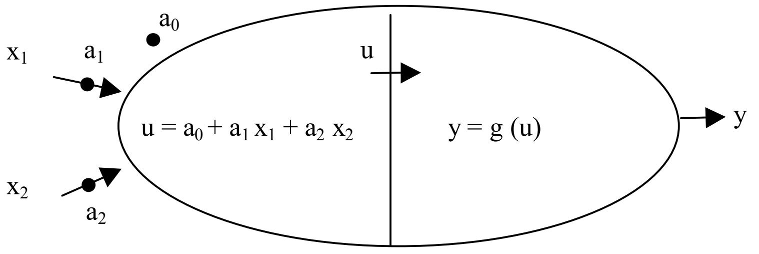


Figure 5. Processing element (PE) in ANN – a hidden node (with two inputs)

Inputs to the network are:

$$x_i\,,\quad i = 1, \dots, N_{inp}$$

Output of the *j*-th node of the hidden layer is (Figure 6):

$$y_j = g\left(a_{0j} + \sum_{i=1}^{N_{inp}} a_{ij}\, x_i \right),\quad j = 1, \dots, N_{hid}$$

Output nodes receive signals form the hidden layer and are very similar (Figure 6). They also have two components: the first one is a linear one, and the second one is usually nonlinear (however, for regression problems the second component is reduced to a linear one).

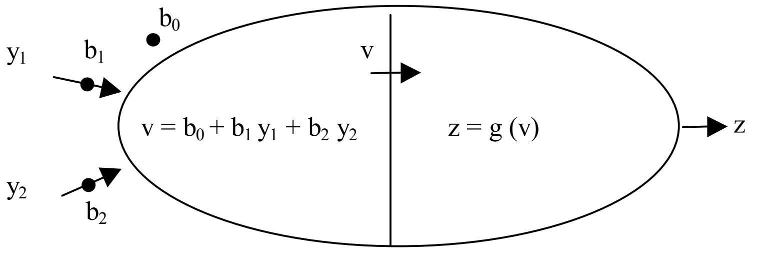


Figure 6. An output node with two inputs and coming from the hidden layer

Outputs of the *k*-th output nodes are:

$$z_k = g\left(b_{0k} + \sum_{j=1}^{N_{hid}} b_{jk}\, y_j \right),\quad k = 1, \dots, N_{out}$$

The free terms a_0 and b_0 are called bias weights. NeuralMachine allows for having the these bias weights or setting them to zero.

The transfer function *g* in hidden layer is usually a non-linear, bounded, and piecewise differentiable function (usually of sigmoidal shape). The widely used sigmoid (logistic) function (Figure 7) is bound between 0 and +1:

$$g(u) = \frac{1}{1 + e^{-\alpha u}}$$

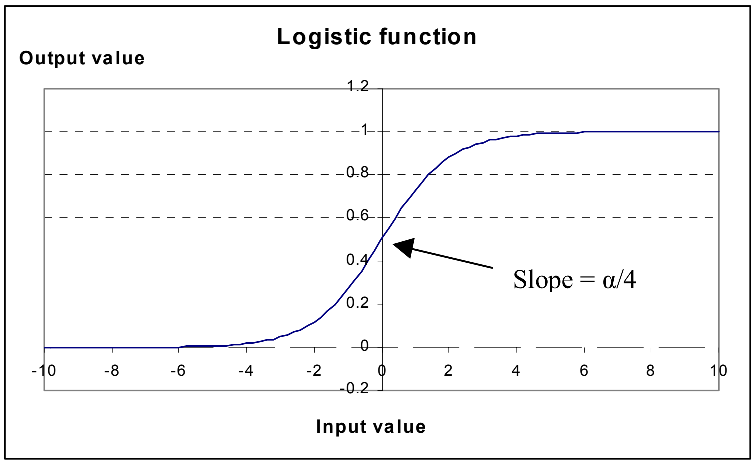


Figure 7. Logistic function (show here for α=1) – a non-linear transfer function commonly used in ANNs

Another commonly used function is hyperbolic tangent (Figure 8) is bound between –1 and +1:

$$g(u) = \tanh(\alpha u) = \frac{e^{\alpha u} - e^{-\alpha u}}{e^{\alpha u} + e^{-\alpha u}}$$

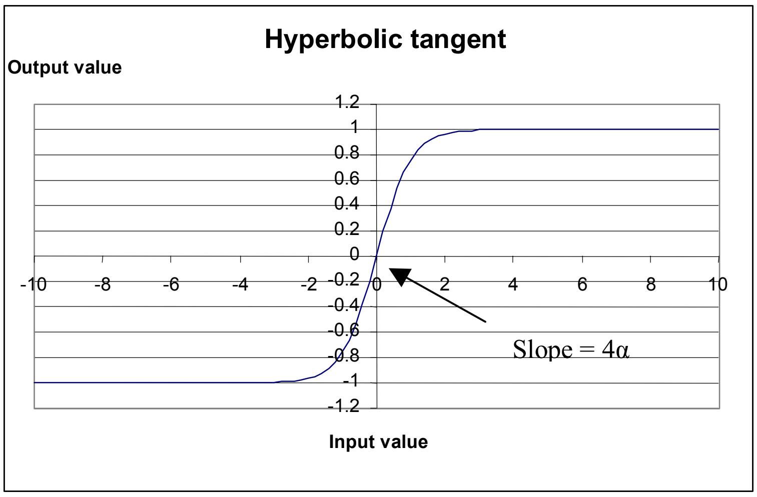


Figure 1. Hyperbolic tangent (show here for α=1) – another popular transfer function

It is useful now to calculate derivative of logistic function since it will be used later. The derivative of the logistic function *g(u)* is:

$$\begin{aligned}\frac{\partial g(u)}{\partial u} &= \frac{\partial (1+e^{-\alpha u})^{-1}}{\partial u} \\ &= (-1)(1+e^{-\alpha u})^{-2} e^{-\alpha u}(-\alpha) \\ &= \alpha e^{-\alpha u}(1+e^{-\alpha u})^{-2} \\ &= \alpha e^{-\alpha u} g^2(u)\end{aligned}$$

It can be seen that

$$e^{-\alpha u} = \frac{1-g(u)}{g(u)}$$

This makes it possible to express the derivative in the following way:

$$\frac{\partial g(u)}{\partial u} = \alpha \frac{1-g(u)}{g(u)} g^2(u)$$
$$= \alpha (1-g(u))\, g(u)$$

Note that for the problems of classification (when the values are from a limited range) both transfer functions in the hidden and output layer are sigmoidal. For regression problems, however, in order not to limit too much the output values, the transfer function in the output layer is often (but not always) selected to be a linear one with the unit slope, so that the output nodes become simply summation nodes. ANN's nonlinearity is ensured then by the sigmoidal functions in the nodes of the hidden layer.

Network complexity and its approximation ability

The ability to approximate is highly dependent on the network complexity, which in term is determined by the number of nodes in a hidden layer. More hidden a network has, more complex functions it can approximate.

In order to illustrate this, we will use a simple example. In it, the mapping to be approximated has one input *x* and 2 outputs (sine and cosine functions). They are represented by a matrix 315 x 3 represnting 315 instances (generated by running *x* from 0 to 6.28 with the step 0.02. The fragment of this matrix follows:

```
0.0000  0.00000  1.00000
0.0200  0.02000  0.99980
0.0400  0.03999  0.99920
...
6.2600 -0.02319  0.99973
6.2800 -0.00319  0.99999
```

NNN tool was used to train an MLP network with different number of hidden nodes. (A so-called radial basis network was trained as well, which is reported later). Figures below demonstrate the approximation ability of a network with 1, 2, 3 and 4 hidden nodes. It can be seen on Figure 9 (depicting only the first output – the sine function) that only in case of using 4 nodes (and more) the fit is becoming almost perfect. Training in this case required around 200 iterations.

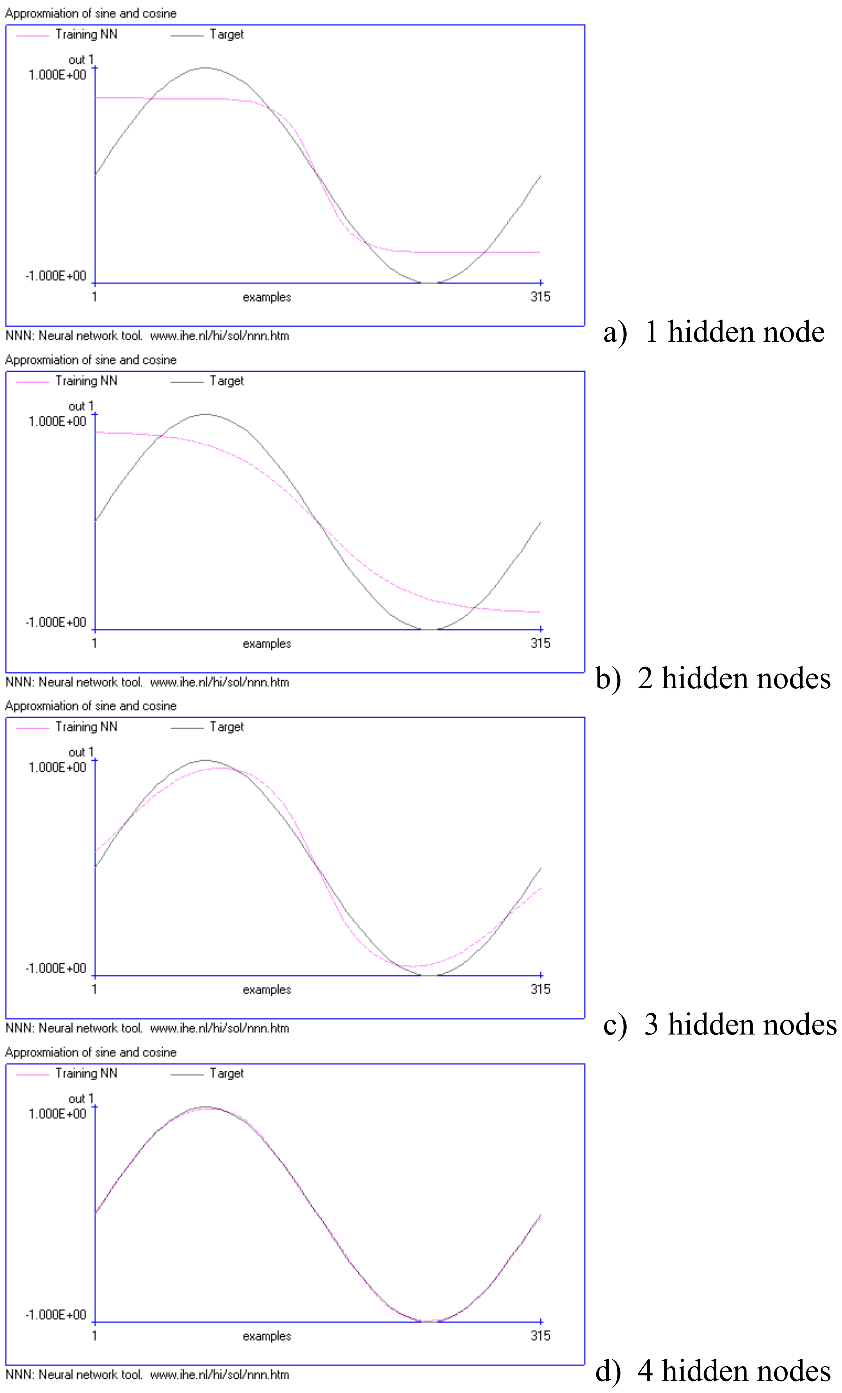


Figure 2. Approximating ability of an MLP network with 1, 2, 3 and 4 hidden nodes

Training of MLP as an optimization problem

Now the question is how to train the MLP. In total, there are

$$(N_{inp}+1)\,N_{hid}+(N_{hid}+1)\,N_{out}$$

parameters (weights ***a*** and ***b***) to be identified; this is achieved by *training*, or *calibrating* the ANN in such a way that the error in reproducing actual function values is minimal.

Let us assume that the following *T* instances are used for training:

$$\{\, x_1^{(t)}, x_2^{(t)}, \dots, x_{N_{inp}}^{(t)}, f_1^{(t)}, f_2^{(t)}, \dots, f_{N_{out}}^{(t)} \,\},\ t=1,\dots,T$$

This means that when N_{inp}-component vector $\boldsymbol{x}^{(t)}$ (also called t-th *input pattern*) was fed into the system as input, the actual (measured) values of the k-th output variable $f_k^{(t)}$ were generated, constituting vector $\boldsymbol{f}^{(t)}$ (also called the t-th *actual output pattern*). For the same input the ANN generated for the k-th output variable value denoted as $z_k^{(t)}$ that together constitute the vector $\boldsymbol{z}^{(t)}$ (also called the t-th *output pattern*).

It is good to summarize that the following variables and indices will be used:

t	for instances (input and output patterns) – denoted in parentheses as a superscript, running from 1 to T;
i	for the input variables x, running from 1 to N_{inp};
j	for the hidden nodes (there output is y), running from 1 to N_{hid};
k	for the output nodes and the output variables z and the measure values f, running from 1 to N_{out};
a_{ij}	for the weights between the input and the hidden layer;
b_{jk}	for the weights between the hidden and the output layer;
w_s	as a generic reference to the weights a and b to be found.

The response of ANN to one input pattern differes from the actual pattern – this difference is squared. For output k it is

$$E_k^{(t)} = (f_k^{(t)} - z_k^{(t)})^2$$

The error for all outputs can be presented in different ways, but the common way to express it is to use the *least squares error*:

$$E^{(t)} = \frac{1}{2}\sum_k (f_k^{(t)} - z_k^{(t)})^2$$

Total error is the summation of the errors for all output nodes for all T instances:

$$E_{tot} = \frac{1}{2}\sum_t \sum_k (f_k^{(t)} - z_k^{(t)})^2$$

The error function can be expressed in the following way:

$$\begin{aligned} E_{tot} &= \frac{1}{2}\sum_t \sum_k (f_k^{(t)} - z_k^{(t)})^2 \\ &= \frac{1}{2}\sum_t \sum_k [f_k^{(t)} - g^{out}(b_{0k} + \sum_j b_{jk} y_j)]^2 \\ &= \frac{1}{2}\sum_t \sum_k [f_k^{(t)} - g^{out}(b_{0k} + \sum_j b_{jk}\ g^{hid}(a_{0j} + \sum_i a_{ij} x_i^{(t)}))]^2 \end{aligned}$$

In order to find the minimum value of E_{tot}, it is necessary to solve an optimization problem:

find such values of all weights ***a*** *and* ***b*** *that bring* E_{tot} *to minimum.*

Since function E is known analytically, and it is differentiable, it is possible to use gradient-based methods like steepest descent (Smith 1993), or more efficient conjugate gradients method.

Backpropagation algorithm

The process of solving this optimization problem is called backpropagation – since it involves computation of the ANN errors and the propagation of the errors back through the network in order to update weights accordingly. It includes the following steps:

Backpropagation algorithm (instance-based)

1 Randomize the weights $\{w_s\}$ (denoted above as matrices ***a*** and ***b***) to small random values (both positive and negative) to ensure that the network is not saturated by large values of weights.

2 Select an instance *t*, that is the vector $\{x_k^{(t)}\}$, $i = 1,...,N_{inp}$ (a pair of input and output patterns), from the training set.

3 Apply the network input vector to network input.

4 Calculate the network output vector $\{z_k^{(t)}\}$, $k = 1,...,N_{out}$.

5 Calculate the errors for each of the outputs *k* , k=1,...,N_{out}, the difference between the desired output and the network output:

$$E_k^{(t)} = (f_k^{(t)} - z_k^{(t)})^2$$

(for simplicity we will denote it as simply *E*).

6 Calculate the necessary updates for weights Δw_s in a way that minimizes this error (discussed below).

7 Adjust the weights of the network by Δw_s.

8 Repeat steps 2 – 6 for each instance (pair of input–output vectors) in the training set until the error for the entire system (error *E* defined above or the error on cross-validation set) is acceptably low, or the pre-defined number of iterations is reached.

Often it is reasonable not to update weights immediately after processing each instance, but accumulates (sums up) the necessary changes across a subset of training instances (call an *epoch*) and only then updates the weights. This allows for faster convergence (Smith 1993). Epoch can be the part or the whole training set. After the whole training set is processed (this sequence of steps is called an *iteration*), the whole process is repeated again in an iterative fashion – until the total error is acceptably low. Number of such iterations may sometimes be as high as several thousand.

So, in case of updating weights after the whole epoch is processed, the backpropagation algorithm will have the following form (actually this version was implemented in NNN software):

Backpropagation algorithm (epoch-based, with cumulative updates)

1 – 6 *as above*

7 add up the calculated weights' updates $\{\Delta w_s\}$ to the accumulated total updates $\{\Delta W_s\}$.

8 Repeat steps 2 – 7 for several instances comprising an epoch.

9 Adjust the weights $\{w_s\}$ of the network by the updates $\{\Delta W_s\}$.

10 Repeat steps 2 – 9 until all instances in the training set are processed. This constitutes one iteration.

11 Repeat the iteration of steps 2 – 10 until the error for the entire system (error *E* defined above or the error on cross-validation set) is acceptably low, or the pre-defined number of iterations is reached.

Question is now how to calculate these updates $\{\Delta w_s\}$ of the weights. Since the training is an optimization problem, the algorithm for solving it can be simply one of the algorithms developed for non-linear optimization problems, in particular following the ideas of *steepest descent* (also called *gradient descent,* or *hill climbing*). In it, "steps" are made in the space of variables (in this case, weights) in the direction opposite to the direction of the gradient of the minimized function (in this case *E*):

$$\boldsymbol{w}(N+1) = \boldsymbol{w}(N) - \eta \nabla E(\boldsymbol{w}(N))$$

where $\boldsymbol{w}$ is the vector of independent variables $\{w_s\}$, η determines how big a step we make, and N is the number of iteration involved. This means that the changes in individual variables should be the following:

$$w_s(N+1) = w_s(N) - \eta \left.\frac{\partial E}{\partial w_s}\right|_{w_s = w_s(N)}$$

or in other words, the update step for variable s is,

$$\Delta w_s = -\eta \frac{\partial E}{\partial w_s}$$

Widrow and Hoff (1960) used this approach (called *delta rule*) for identifying weights in a single linear perceptron (processing element). This approach was later used for multi-layer perceptrons as well. Actually the paper of Rummelhart, Hinton and Williams (1986) presenting this algorithm was one of the key steps in making ANNs really practical. (The algorithm was however first proposed by J.P. Werbos in 1974 as part of his Ph.D. dissertation at Harward University).

We will be considering the following chain of neurons depicted on Figure 10.

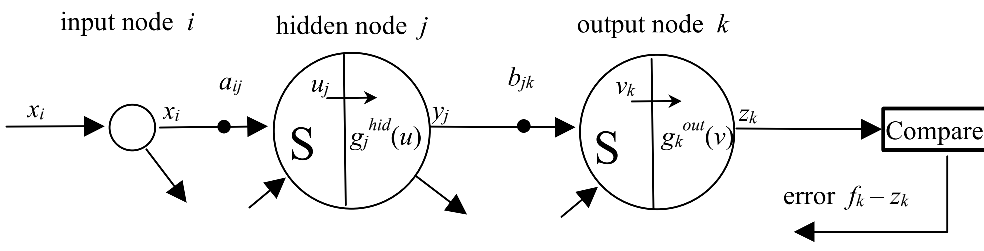


Figure 3. The chain of nodes considered in the back propagation algorithm

For generality, we use different notations for the activation function: g^{hid} for all nodes in the hidden layer, and g^{out} for all nodes in the output layer (sometimes this superscript will be omitted). We assume that these functions are either logistic or linear. (In case of hyperbolic tangent function the derivation is very similar.)

We will now separately derive the necessary changes in the output weights $\boldsymbol{b}$ and input weights $\boldsymbol{a}$. The derivations are a bit different since there is no target (measured) values for the hidden nodes' outputs.

Calculation of weights for the output layer

For the output weight b_{ij} the change $b_{ij}^{new} - b_{ij}^{old} = \Delta b_{ij}$ can be written as follows:

$$\Delta b_{jk} = -\eta \frac{\partial E}{\partial b_{jk}}$$

where η is the coefficient of proportionality called *learning rate* (but determining actually the size of the step in the steepest descent algorithm). Recall that

$$z_k = g(v_k) = g(b_{0k} + \sum_j b_{jk} y_j)$$

To evaluate the partial derivative of $E_k^{(t)}$ (denoted for simplicity as E), we use the chain rule of differentiation:

$$\frac{\partial E}{\partial b_{jk}} = \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial v_k} \frac{\partial v_k}{\partial b_{jk}}$$

Each of these terms will be evaluated in turn. The first term is

$$\frac{\partial E}{\partial z_k} = -2(f_k - z_k)$$

The second term is actually the derivative of the activation function (logistic or linear). Using the expression for the derivative of the logistic function *g* given above, it can be expressed as

$$\frac{\partial z_k}{\partial v_k} = \frac{\partial g_k(v)}{\partial v_k} = \alpha\, g_k(v)(1 - g_k(v))$$

For the linear function with the unit slope,

$$\frac{\partial z_k}{\partial v_k} = \frac{\partial g_k(v)}{\partial v} = 1$$

At last, the third term for $j > 0$ can be expressed as:

$$\frac{\partial v_k}{\partial b_{jk}} = \frac{\partial (b_{0k} + \sum_j b_{jk} y_j)}{\partial b_{jk}} = y_j$$

(for $j = 0$ it is equal to 1).

Finally, combining the three terms, for $j > 0$ the error derivative gives:

$$\frac{\partial E}{\partial b_{jk}} = -2(f_k - z_k)\frac{\partial g_k}{\partial v} y_j = -\delta_k\, y_j$$

and for $j = 0$ (bias weights):

$$\frac{\partial E}{\partial b_{0k}} = -\delta_k$$

where

$$\delta_k \equiv 2(f_k - z_k)\frac{\partial g_k(v)}{\partial v}$$

which, in case of using the logistic function, is:

$$\delta_k \equiv 2\alpha(f_k - z_k)\; g_k(v)(1 - g_k(v))$$

The update rule can be then written as

$$b_{jk}(N+1) = b_{jk}(N) + \eta\,\delta_k\, y_j$$

and for the bias weight as

$$b_{0k}(N+1) = b_{0k}(N) + \eta\,\delta_k$$

Calculation of weights for the hidden layer

This derivation is similar, but the difference is that the outputs from the hidden nodes do not have explicit values of an error. Such errors are propagating from each of the nodes of the output layer to each of the nodes in the hidden layer. This means that the consideration of some node *j* of hidden layer should involve all errors in the output layer being summed up which is normally expressed as the least squares error:

$$E^{(t)} = \frac{1}{2}\sum_k (f_k^{(t)} - z_k^{(t)})^2$$

Since the error is now the summation of the errors for each output, the weights' change for the update rule will expressed in the following way:

$$\Delta a_{ij} = -\eta\frac{\partial E^{(t)}}{\partial a_{ij}} = -\eta\sum_k \frac{\partial E_k^{(t)}}{\partial a_{ij}}$$

Using the chain rule, the error derivative can be presented as

$$\frac{\partial E^{(t)}}{\partial a_{ij}} = \frac{\partial E_k^{(t)}}{\partial y_j}\frac{\partial y_j}{\partial u_j}\frac{\partial u_j}{\partial a_{ij}}$$

Since the hidden nodes themselves do not make errors themselves; they contribute to the errors of the output nodes, so the first term is the sum of the hidden node's contributions to the errors of the output nodes (sum runs through all output nodes *k*):

$$\frac{\partial E^{(t)}}{\partial a_{ij}} = \sum_k \frac{\partial E_k^{(t)}}{\partial z_k}\frac{\partial z_k}{\partial v_k}\frac{\partial v_k}{\partial y_j}\frac{\partial y_j}{\partial u_j}\frac{\partial u_j}{\partial a_{ij}}$$

Each of these terms is similar to the terms derived above. The details of this derivation can be found for example in Smith (1993) or in Tsoukalas and Uhrig (1997). The final update rule for the weights in the hidden layer is:

$$a_{ij}(N+1) = a_{ij}(N) + \eta\ x_i \frac{\partial g_j(u)}{\partial u}\sum_k \delta_k b_{jk}$$

If we use the previously introduced δ_k to define

$$\delta_{jk} = \delta_k b_{jk}\frac{\partial g_j(u)}{\partial u} = \alpha\delta_k b_{jk}\ g_j^{hid}(1-g_j^{hid})$$

then the update rule for $i > 0$ is

$$a_{ij}(N+1) = a_{ij}(N) + \eta\ x_i \sum_k \delta_{jk}$$

and for the bias weights as

$$a_{0j}(N+1) = a_{0j}(N) + \eta\sum_k \delta_{jk}$$

Updating the weights

Remember that the ANN's error $f_k^{(t)} - z_k^{(t)}$ is incorporated in the δ_k and so the error is propagated backwards through the network and used to update the weights. The presented derivation is made only for one instance *t*. As presented in the two versions of the backpropagation algorithms above, after the weights are updated, the next instance is used to calculate the output, compute the errors, calculate the weights' updates, etc. Then either weights are updated, or they are updated for the whole epoch. The training stops when the error (either the total square error E_{tot} or the error on the basis of a cross-validation set) is low enough, or the pre-defined number of iterations is reached.

Improvements in backpropagation training

Backpropagation, as a version of a simple gradient descent iterative optimization algorithm may have the stability and convergence problems. Some of them can be addressed by using a faster and more stable optimization algorithm (for example, using the second derivatives), and some of them, due to the complexity of the modelled functions, are not solvable and could be only helped when an analyst obtains an experience in what is called the "art of modelling".

Momentum

After the BP algorithm was first introduced in its pure form, it was soon found that it is quite slow. This is of course no surprise, since the steepest descent algorithm is known for its low speed, especially close to a minimum (since the gradient is disappearing). One of the reasons is that it uses the fixed-size step. In order to take into account the changing curvature of the error surface, many optimization algorithms use steps that vary with each iteration.

Another reasonable idea, also used in optimization, is to keep the minimization process going in the same general direction not allowing too much "wagging" as the steepest descent algorithm often does, and jumping out of a local minimum. All this involves adding a term to the weight adjustment that is proportional to the amount of the previous weight change. That is, the previous adjustment is "remembered" and used for the modification of the next change in weights. Such term is called a "momentum" term since it makes the movement similar to the to the way a heavy object (i.e. having a momentum) moves. The weight update is then

$$\Delta w_s(N+1) = -\eta \frac{\partial E}{\partial w_s} + \mu\, \Delta w_s(N)$$

where μ is the *momentum coefficient* (its value is typically set to 0.9).

Adaptive learning rate used in NeuralMachine

It is possible to go full way and to explicitly introduce the adaptive step (learning rate $\eta_s(N)$), being different for all weights. This allows to make the algorithm even more adaptive. The possible formula for the weight update follows (Smith 1993):

$$\Delta w_s(N+1) = -(1-\mu)\eta_S(N)\frac{\partial E}{\partial w_s} + \mu \Delta w_s(N)$$

where $\eta_s(N)$ = the learning rate which is updated according to the following rule:

$$\begin{aligned} \eta_s(N) &= \eta_s(N-1) + \kappa, \quad \text{if } \frac{\partial E}{\partial w_s} RAED(N-1) > 0 \\ &= \eta_s(N-1)\,\varphi, \quad \text{otherwise} \end{aligned}$$

Here *RAED* is the recent "average" of the error derivative that is recursively calculated:

$$RAED(N) = (1-\theta)\frac{\partial E}{\partial w_s} + \theta\, RAED(N-1)$$

and the conditions actually say: "increase the learning rate (step) if the error decrease is in the same direction as recently, and reduce otherwise".

Choice of the learning constants

There are no fixed rules for that. Some recommendations can be given but the choice of these constants is often part of the "art of modelling". Smith (1993) recommends the following values:

$\kappa = 0.1$ (0.05 is often better)
$\varphi = 0.5$
$\theta = 0.7$ (or 0.5)
$\mu = 0.9$ (or 0.5).

In NeuralMachine these values (denoted on the 'Learning parameters' tab sheet as Kappa, Phi, Theta and Mu) are used as defaults.

Direct in-to-out link

Sometimes it may be useful to introduce the direct in-to-out link, going directly from the inputs ***x*** to ***z***. This allows to capture direct input-output relationships if they exist. NNN tool allows for such link.

Dealing with the local minima

Sometimes during training it is observed, that after long training, the algorithms seem to stall – error is still high but the continuous training does not lead to its reduction. One of the explanations is that the optimization algorithm found a local minimum, but not the global (overall) one.

Being trapped in a local minimum is a common problem for traditional nonlinear optimization algorithms. Since BP algorithm is a version of steepest descent, it assumes (as many other nonlinear optimization algorithms) that the error surface slope is always negative and hence constantly adjusts the weights towards the minimum. The problem is that the error surface in high dimensions may be very complex, including hills, valleys, folds and gullies (ravines). The process may be trapped in one of these local minima, whereas the global minimum should be searched somewhere else. Actually these problems are studied in the framework of the so-called *global oprimization* (Torn and Zilinskas 1989; Solomatine 1999).

So, in case if BP seems to stall, some "help" is needed. The versions of BP with the adaptive learning rate mentioned earlier may sometimes help. However, one of the widely used solutions (however, not theoretically supported) is to use "shock", that is changing all weights by a fixed or random amount. If this fails, then the weights can be re-randomized and the process repeated. NeuralMachine uses this approach.

Another approach is to use a global optimization algorithm in case if BP seems to stall – it may also help to jump out of the area around a local minimum. Global optimization algorithms that can be used are: simulated annealing (example of this see in Tsoukalas and Uhrig 1997), genetic algorithm, or adaptive cluster covering (Solomatine 1999).

Radial basis function (RBF) networks

Function approximation

Machine learning can be seen as a problem of function fitting, and this problem was studies in mathematics for more than 150 years. In function fitting (Gershenfeld 2000) an attempt is made to find a set of functions that, being combined, would approximate a given function (or a data set). Traditionally, linear (as in linear regression), or polynomial functions were used. Examples could be *splines* – combinations of functions that use cubic polynomials to match the values and first and second derivatives (at the boundaries) of the function to be approximated. Another example is orthogonal so-called orthogonal polynomial functions, e.g., Chebyshev polynomials.

The problem of combining complex polynomial functions is that in order to improve the fit it is necessary to add higher-order terms, and they may diverge very quickly (being a good approximator close to one point, and a very bad one a bit further). Matching data requires a delicate balancing of the coefficients, resulting in a function that is increasingly "wiggly". Even if it passes near the training data, it will be useless for interpolation or extrapolation. This is particularly true for functions that have sharp peaks (which are actually so frequent in many engineering applications).

Radial basis functions

Radial basis functions (RBF) (quite simple in nature) could be seen as a sensible alternative to the mentioned attempts to use complex polynomials for function fitting. Consider a function $z = f(\mathbf{x})$, where $\mathbf{x}$ is a vector $\{x_1,..., x_I\}$ in I-dimensional space. The idea is approximate a function $z = f(\mathbf{x})$ by another function $F(\mathbf{x})$ in a proximity to some "representative" locations (centers) $\mathbf{w}_j$,j=1,...,J (Figure 12).

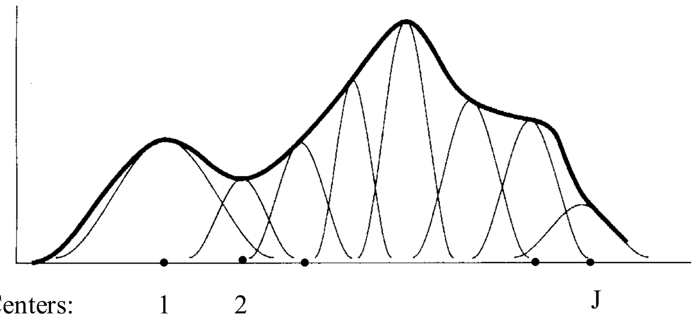


Figure 4. Approximation by RBFs in one dimension

Such basis functions $F(\mathbf{x})$ depend only on the distance (radius) from these centers, are identical and are defined everywhere but quickly drop to zero as the distance from the location $\mathbf{w}_i$ increases. For some point **x** the approximating function is then always expressed as the sum of the basis functions:

$$z(\mathbf{x}) = \sum_{j=1}^{J} F(|\mathbf{x} - \mathbf{w}_j|; \mathbf{b}_j)$$

where $\mathbf{b}_j$ are coefficients associated with the j-th center $\mathbf{w}_j$. If the centers are fixed and the centers are entered linearly, then finding an RBF becomes a linear problem:

$$z(\mathbf{x}) = \sum_{j=1}^{J} b_j F(|\mathbf{x} - \mathbf{w}_j|)$$

Common choice for these F is the fixed-width Gaussian ($f(r) = exp(-r^2)$). Other functions that can be used include linear ($f(r) = r$) and cubic ($f(r) = r^3$). (These two last functions, however, do not drop with the distance from $\mathbf{w}_i$ as the Gaussian does).

In order to allow for higher accuracy, we can introduce the "width" parameter δ into Gaussian functions for each center: $f(r) = exp(-r^2 / \delta^2)$. ($\delta$ is analogous to the standard deviation in a Gaussian normal distribution). Distance $|\mathbf{x} - \mathbf{w}_j|$ is usually understood in Euclidean sense and denoted as δ_j :

$$\delta_j = \sum_{i=1}^{I} (x_i - w_{ij})^2$$

In this case the approximation becomes:

$$z(\mathbf{x}) = \sum_{j=1}^{J} b_j \exp(-\delta_j^2 / \sigma_j^2)$$

where σ controls the Gaussian's width.

The problem of approximation requires:

- the placement of the localized Gaussians to cover the space (positions of the centers $\mathbf{w}_i$);
- the control of the width of each Gaussian (parameter σ);
- the setting of the amplitude of each Gaussian (parameters b_i).

Radial basis functions in a network setting

A Radial Basis Function (RBF) ANN is basically a structure that represent the idea just described (Figure 13; it may have however more than one output). RBF ANN can be seen hence as another type of feed-forward ANN. Additionally, it uses the technology of ANN training to identify the values of the mentioned parameters, and a clustering algorithm to identify positions of centers. Typically in an RBF network, there are three layers: one input, one hidden and one output layer. The hidden layer uses Gaussian transfer function instead of the sigmoid function. In RBF networks, one major advantage is that if the number of input variables is not too high, so the learning is much faster than in other types of networks. However, the required number of the hidden units increases geometrically with the number of the input variables. It becomes practically impossible to use this network for a large number of input variables.

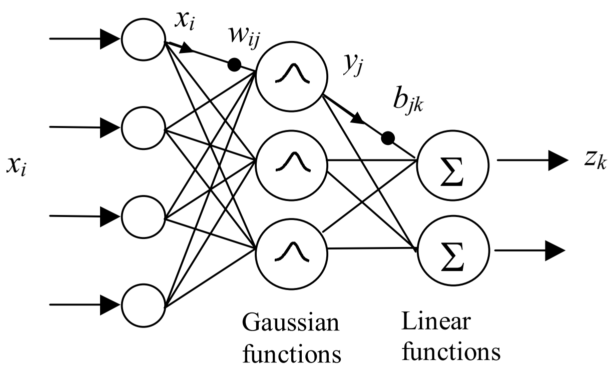


Figure 5. RBF neural network with four inputs and two outputs

The hidden layer in RBF network consists of an array of nodes that contains a parameter vector called a 'radial centre' vector (Schalkoff, 1997). The hidden layer performs a fixed non-linear transformation with non-adjustable parameters. The approximation of the input-output relation is derived by obtaining a suitable number of nodes in the hidden layer and by positioning them in the input space where the data is mostly clustered. At every iteration, the position of the radial centers, their width (variation) and the linear weights to each output node are modified. The learning is completed when each radial centre is brought up as close as possible to each discrete cluster centers formed from the input space, and the error of the network's output is within the desired limit.

The centers and widths of the Gaussians are set by the unsupervised learning rules, and the supervised learning is applied to the output layer. For this reason RBF networks are called hybrid networks.

Learning algorithm for RBF

Given the number of the hidden nodes (centers) *J* is chosen, the learning algorithm is formulated as follows:

1 Find the positions of centers $\{\mathbf{w}_j\}$. This can be done by the following procedure:

- Choose randomly J instances $\mathbf{x}_j$ and use them as the positions of the centers $\{\mathbf{w}_j\}$
- All the remainder of the instances (training patterns) are assigned to a class j of the closest centre $\mathbf{w}_j$, and the locations of each center are calculated again using for example k-nearest neighbor method.
- The above steps are repeated until the locations of the centers stop changing.

2 Calculate the output from each hidden neuron as a function of a radial distance from the input vector to the radial center. Calculated distance between the center and the input vector is passed through a non-linear mapping Gaussian function.

3 Weights $\{b_{jk}\}$ for the output layer are calculated using methodologies as in MLP, using backpropagation. The output from the output node can be expressed as

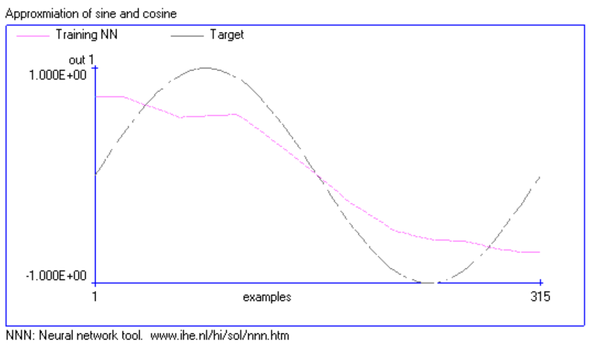
$$z_k = \frac{\sum_{j=1}^{J} b_{jk} y_j}{\sum_{j=1}^{J} y_j}$$

where b_{jk} – the weight on the connection from the hidden node j to the output node k,
y_j - the output from the hidden node j

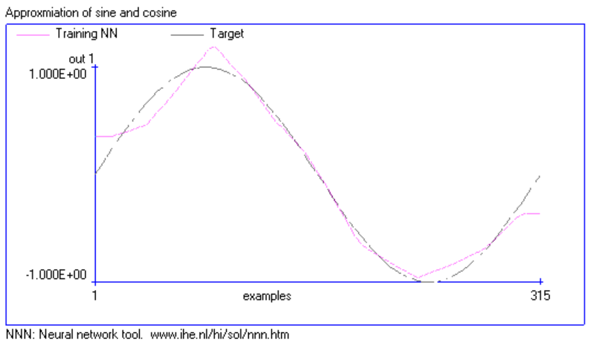
4 Calculate the error between the network's output and the target output and if the error of the network's output is more than the desired limit then the number of the hidden units are changed and all the steps are repeated again.

Gaussian width and the RBF performance

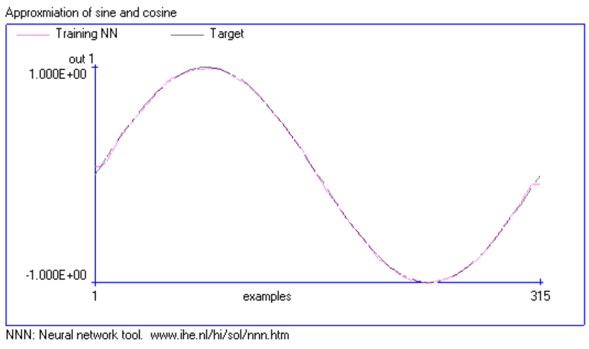
Parameter δ (Sigma) in the Gaussian plays an important role when we try to increase accuracy of RBF approximation. In order to illustrate this, we will use the same example of approximating sine and cosine, as in the section on MLP. Figure 14 shows the training performance with 3 different values of δ: 40.0, 10.0 and 3.0.



a) σ = 40.0, 30 centers



b) σ = 10.0, 26 centers



c) σ = 3.0, 20 centers

Figure 6. Approximation ability of a RBF network with different Gaussian widths (Sigma)

Characteristics of RBF networks

In conclusion, it can be stated that RBF networks:

- provide a global approximation to the target function, represented by a linear combination of many local kernel functions;
- this can be viewed as the smooth linear combination of piece-wise (local) non-linear functions – that is the best function chosen for a particular range of input data;
- training is faster than backpropagation networks since it is done in two steps;
- the accuracy of the solution is highly dependent on the range and quality of data;

it is an so-called eager method, but used an idea of local approximation as in lazy methods such as k-nearest neighbours. (Lazy methods that do not build a model on the basis of all available data, as ANN do, but rather "lazily" process new instances as they appear and the result is generated on the basis of training examples).

How NeuralMachine trains RBF networks

The following characterises the way NeuralMachine trains RBF networks:

Sigma is found automatically in such a way that it is best on average, so it is the same for all nodes. It is also possible to set it to a predetermined value.

Optimization is performed on the number of hidden nodes. NeuralMachine builds RBF networks with the number of hidden nodes varying from 1 to 30 and selects the best (most accurate) network

Practical issues of training ANNs

There are three aspects related to practical use of ANN:

- the size and scaling of data
- the number of hidden nodes, choice of algorithms and activation functions
- the performance criteria.

Setting up the data scene: training and verification data sets

ANN is one of the data-driven methods, and all the considered issues of data preparation, selection of the training, cross-validation and testing sets, checking the generalization accuracy, etc. are valid here as well. Training data set must cover all the character of the problem classes. Obviously it is desirable that the training data includes the maximal and minimal values. The only general rules are to use a lot of representative data. Understanding of the problem to be solved is fundamentally important in this step.

Scaling the data to prevent network paralysis

It is always useful to normalize data to 0–1 interval – in this case the ranges of weights will not differ too much and this is a desirable condition for optimization algorithms.

Another type of scaling may be needed for MLP networks. Due to the characteristics of the logistic function it gets to saturation very quickly:

```
g(1.0) = 0.762
g(2.0) = 0.964
g(3.0) = 0.995
g(4.0) = 0.999
```

This means that the range of input variables should be kept quite narrow, often between –3 and +3. If this is not done, the so-called *neural network paralysis* may be observed, when the inputs to the logistic functions are so high that their outputs are very close to –1 or +1, and the training process is blocked.

Another useful transformation is transforming the output values to the [0.1..0.9] range (in case of using the logistic output nodes in MLP). The reason is that the sigmoid output nodes cannot produce values outside of the range [0..1], and if we attempt to train the network to fit target values beyond this range, gradient descent will force the weights to grow without bound. On the other hand, values 0.1 and 0.9 are achievable using a sigmoid node with finite weights. Most tools perform the mentioned types of scaling automatically.

NeuralMachine does scaling presented above automatically.

Number of hidden nodes

Hidden nodes number associates the mapping ability of the network - the larger the number, the more powerful the network. However, if this number is too large, the generalization may get worse. This is due to overfitting the training set, which can be solved by using cross-validation. The best hidden nodes number can be obtained by trial-and-error approach.

For MLP networks one of the “rules” is that the number of hidden nodes should be approximately

$$N_{hid} \approx \sqrt{N_{inp}\ N_{out}}$$

In RBF networks the number of hidden nodes is determined automatically.

Choice of the activation functions for MLP

As mentioned before, in regression problems the best results are achieved with the nonlinear sigmoidal functions in the hidden layer and the linear functions in the output layer.

> **Note**. Using the 'Data/Edit project properties' form you can only select the same type of the transfer function (sigmoid or linear) for all nodes (separately hidden and output). However NeuralMachine allows to choose the node types for for every hidden or output node independently. This should be done by direct editing the ANF file ('Data/Edit project file as text' menu option).

Performance criteria

Up to now the only criterion of the ANN’s performance was its least square error LSE (or mean square error, MSE), and its quadratic nature was explicitly used in deriving the BP update rules. With only minor changes in formulas, the root mean square error (*RMSE*) can be used as well. Other criteria, like correlation coefficient (*R*), mean absolute error (*MAE*), and maximum absolute error can also be used to evaluate the performance of ANN, but in an indirect way – after the ANN is trained on the basis of LSE. There are reports on using, for example, cubic expressions for error with the conclusion that there is not much sense in doing so (Tsoukalas and Uhrig 1997).

NeuralMachine uses MSE error in training process but reports both MSE and RMSE.

When to stop training

In case of RBF networks this is happening automatically.

If you train MLP, after 20000 iterations training stops. However, a better strategy is to watch both training and cross-validation errors. Typically, training error is going down and cross-validation error first also goes down but then may begin to rise. This is the moment to stop training. But of course all generalizations are wrong and even this one. The decision when to stop training is what is called "art of modelling".

Instance-based learning and k-Nearest neighbor algorithm

Introduction

The methods for data-driven modelling considered up to now were first building a model of the available (training) data (process of learning), and then were put to operation, when classification or numerical prediction was taking place. These methods are sometimes referred to as *eager learning* (since they are eager to build a model first). However, there is a group of learning methods when a model is not actually constructed – such learning is called *instance-based learning*, or *lazy learning*.

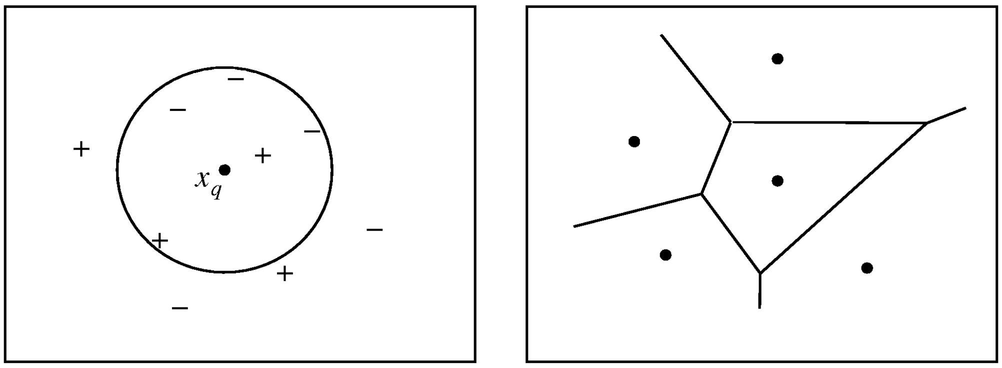
Instance-based (IB) learning methods simply store the training examples and postpone the generalization (building a model) until a new instance must be classified or prediction made. (This explains another name for IB methods – lazy learning – since these methods delay processing until a new instance must be classified). The model that is built by IB methods is not a *global* model that uses all training data, but rather a *local* model involving only some of the instances. The IB methods are used both for classification and for regression. Most important methods are: nearest neighbor method, locally weighted regression, and case-base reasoning. Other names for IB methods are: exemplar-based, case-based, experience-based, edited *k*-nearest neighbor.

Classification problem: k-Nearest neighbor algorithm

k-nearest neighbor (*k*-NN) method assumes all instances correspond to points in the *n*-dimensional space. The nearest neighbors of an instance are defined in terms of the standard Euclidean distance. The problem of classification is posed in the following way:

model a discrete-valued target function F: $\Re^n \rightarrow V$, where V is the finite set $\{v_1,...v_s\}$.

An example for 2-dimensional space is presented on Figure 7a.



a) classifying the new instance x_q to + or – b) Voronoi diagram

Figure 7. k-*nearest neighbor method*

Instance here are points in 2-dimensional space, the number of classes s = 2, so the output is boolean (denoted as "+" or "–"). New instance x_q (called also a *query point)* is classified with respect to proximity of nearest training instances. If we apply 1-NN method, we will consider only 1 such training instance, and x_q will be classified to "+" (since the nearest training instance belongs to class "+"). If, however, 5-NN method is used, we consider 5 training instances, and x_q will be classified to "–" (since among 5 instances there are 2 "+" and 3 "–"). This example explains the essence of the *k*-NN algorithm – to classify a new x_q it finds the most common value of the nearest training instances.

Figure 6.1b shows the Voronoi diagram – a decision surface induced by the 1-NN algorithm for a typical set of training instances. The convex polygon surrounding each training instance indicates the region of instance space closest to that point.

In this section we will denote an arbitrary instance x as $\{a_1(x) \ldots a_n(x)\}$ where $a_r(x)$ denotes the value of the r-th attribute of instance x. The distance between two instances x_i and x_j is defined to be $d(x_i, x_j)$ where

$$d(x_i, x_j) = \sqrt{(a_r(x_i) - a_r(x_j))^2}$$

The algorithm follows.

k-*Nearest neighbor algorithm*

Training
Build the set of training examples D.

Classification
Given a query instance x_q to be classified,

Let $x_1 \ldots x_k$ denote the k instances from D that are nearest to x_q
Return

$$F(x_q) = \arg\max_{v \in V} \sum_{i=1}^{k} \delta(v, f(x_i)) \qquad \text{Eq. 1}$$

where $\delta(a, b)=1$, if $a = b$, and $\delta(a, b)=0$ otherwise.

Distance weighted *k*-NN algorithm

A refinement of the *k*-NN classification algorithm is to weigh the contribution of each of the k neighbors according to their distance to the query point x_q, giving greater weight w_i to closer neighbors. This can be accomplished by replacing the final line in the algorithm by

$$F(x_q) = \arg\max_{v \in V} \sum_{i=1}^{k} w_i \; \delta(v, f(x_i))$$

where the weight is

$$w_i = \frac{1}{d(x_q, x_i)^2}$$

(in case x_q exactly matches one of x_i, so that the denominator becomes zero, we assign $F(x_q)$ to be $f(x_i)$ in this case.

For the version of k-NN for real-valued output the final line of the algorithm will be:

$$F(x_q) = \frac{\sum_{i=1}^{k} w_i f(x_i))}{\sum_{i=1}^{k} w_i}$$

If weighting is used, it makes sense to use all training examples, not just k – the algorithm then becomes a *global* one, since all training instances are used. The only disadvantage is that the algorithm will run more slowly.

Remarks on *k*-nearest neighbor algorithm

The distance weighted *k*-NN algorithm appears to be robust to noisy training data and effective when provided with sufficiently large set of training data. This is a local method, approximating the underlying target function locally, and this can help in smoothing out the impact of isolated noisy training examples. There is one problem, however, which is linked to the way the distance is calculated. In contrast to the decision tree method, for example, where instances are split on the basis of the most relevant attributes, the Euclidean distance involves all the attributes, independent of their relevancy to a particular classification problem. The distance between neighbors may be dominated by the large number of irrelevant attributes. There are some possibilities to overcome this problem:

- to exclude the non-relevant attributes from consideration at the data preparation stage;
- to weight each attribute differently when calculating the distance between two instances.

Interesting to note that when number of training examples is very large, k-nearest neighbor method approaches the Bayesian optimal classification.

Instance-based learning in numerical prediction (regression)

Extension of k-Nearest neighbor algorithm

k-NN algorithm is easily adopted to perform a real-valued prediction, that is to approximate the continuous-valued target function. Instead of calculating the most common value of the nearest training instances, it has to calculate their mean value. The problem of regression can be posed as follows:

model a real-valued target function F: $\Re^n \to \Re$.

In this case the final line (Eq. 1) on the *k*-NN algorithm should be replaced by the line

$$F(x_q) = \frac{\sum_{i=1}^{k} f(x_i))}{k}$$

Locally weighted regression

Local weighted regression (numerical prediction) is a generalization of the nearest-neighbor approaches. It constructs an explicit approximation $F(x)$ of the target function $f(x)$ over a *local* region surrounding the new query point x_q. The type of this approximation can be any function – linear, quadratic or even an ANN. This method is called *weighted* because the contribution of each training example is weighted by its distance to the query point x_q.

If $F(x)$ is linear then this is called locally weighted linear regression

$$F(x) = w_0 + w_1 a_1(x) + \ldots + w_n a_n(x)$$

If we construct a global model, we have to minimize the error for all training examples D:

$$E = \frac{1}{2} \sum_{x \in D} (f(x) - F(x))^2$$

However, for local models it is necessary to look only at the proximity of x_q. In this case there are the following possibilities:

1 Minimize the squared error over just k nearest neighbors:

$$E_1(x_q) = \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - F(x))^2$$

2 Minimize the squared error over entire set *D* of training examples, while weighting the error of each training example by some decreasing function *K* of its distance from x_q:

$$E_2(x_q) = \frac{1}{2} \sum_{x \in D} (f(x) - F(x))^2 \, K(d(x_q, x))$$

3 Combine 1 and 2 (to reduce computational costs):

$$E_3(x_q) = \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - F(x))^2 \, K(d(x_q, x))$$

The problem of minimization of *E* can be solved by various iterative gradient-based methods developed in non-linear optimization. In case of linear *F*(*x*) this can be done analytically.

Instance-based learning in NeuralMachine

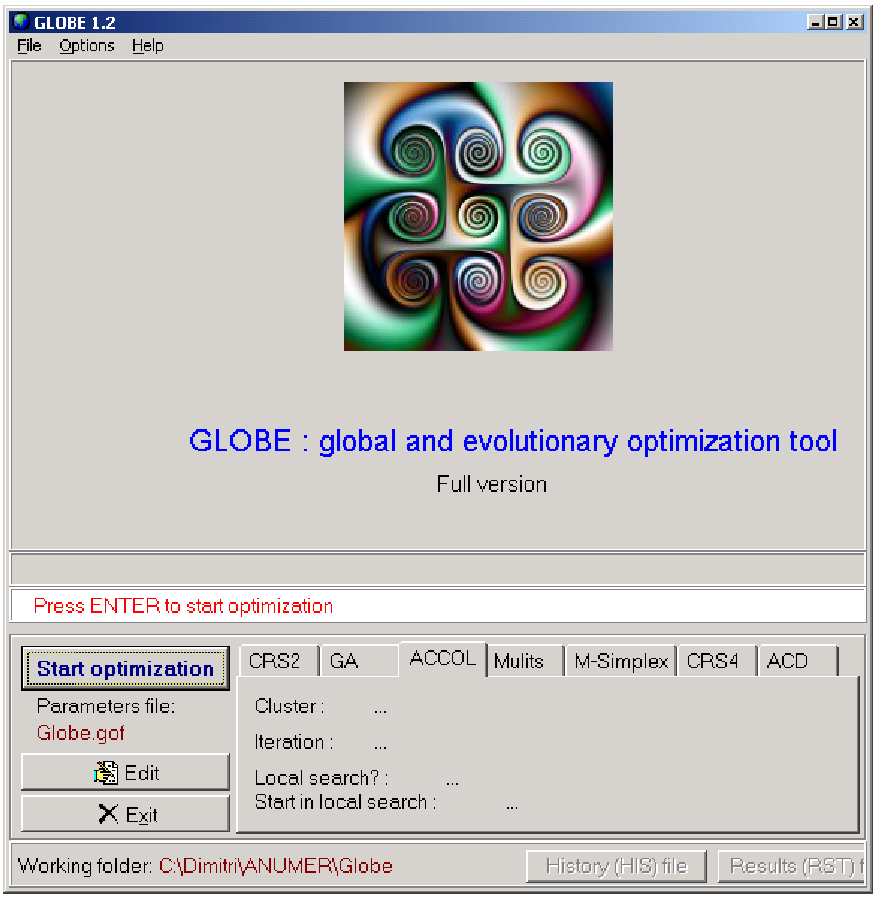
In NeuralMachine, you can choose which algorithm to use using the 'Set model' button/form. There is a choice of three algorithms: (1) MLP; (2) RBF and (3) k-Nearest neighbor (instance-based learning).

Depending on the problem type (that is on the type of the output variable – numerical or class), NeuralMachine automatically selects either the correct version of instance based learning – either the version of the k-Nearest neighbor algorithm for classification, or Local weighted regression for numerical prediction.

Note that since there is no training phase, clicking on 'Train' button presents a simplified screen with only the 'Run' button. Clicking on it runs the algorithm and presents the verification result.

Other computational intelligence software of the same author

Check GLOBE : global and evolutionary optimization tool. See Help/About for details.



GLOBE : global and evolutionary optimization tool

Full version

Press ENTER to start optimization

Parameters file: Globe.gof

Cluster : ...

Iteration : ...

Local search? : ...

Start in local search : ...

Working folder: C:\Dimitri\ANUMER\Globe

(c) 1988-2003, D.P. Solomatine