An Introduction to the Backpropagation Neural Network Code Developed at TRIP

Philippe Ecoublet [*] *and William W. Symes* [†]

ABSTRACT

A backpropagation neural network algorithm has been developed at The Rice Inversion Project as an inversion tool for seismic reservoir characterization. Neural networks can be implemented in a straighforward way for tackling a broad variety of prediction and classification problems as an alternative to model based inversion methods. This document is an introduction to the backpropagation neural network, probably one the most widely used neural network, and describes the procedure for using the program available at TRIP.

Backpropagation Neural Network

Neural networks have been widely used for many different applications of pattern recognition. A comprehensive review and theoretical development of neural networks can be found in (Bishop, 1996) and (Wasserman, 1993). Applications of backpropagation neural networks (BPNN) appeared in geophysics in recent years for log-to-log prediction (Rogers *et al.*, 1992).

TRIP's backpropagation neural network algorithm is coded by calling the BFGS (Broyden-Fletcher-Goldfarb-Shanno) (Polak, 1971) optimization algorithm part of the Hilbert Class Library (Gockenbach and Symes, 1996a; Gockenbach and Symes, 1996b). Two slighly different versions of the BPNN are available to solve two different classes of problems: *prediction* and *classification* for cluster analysis. The distinction between these two versions is detailed throughout the following description of the BPNN.

A BPNN is defined by an input layer, an output layer and a variable number of so-called hidden layers. A schematic representation of a backpropagation neural network is

[*] The Rice Inversion Project, Department of Computational and Applied Mathematics, Rice University, Houston TX 77005-1892 USA, email `ecoublet@caam.rice.edu`

[†] The Rice Inversion Project, Department of Computational and Applied Mathematics, Rice University, Houston TX 77005-1892 USA, email `symes@caam.rice.edu`

given on Figure 1. Each layer is composed of one or more "neurons". Each neuron in a layer is connected to each neuron of the prior and the next layers. The connection between two neurons is parameterized by a weight. A value for the weight is estimated during the training process so as to minimize the difference between the computed output of the network and the targeted output. Each individual neuron is connected to an additional weight independent of any inter-layer connection. This additional weight plays the role of an offset or bias. The BPNN represented on Figure 1 is an example with five neurons in the input layer and three neurons in the output layer. The network has one hidden layer constituted of five neurons. That network has 25 weights connecting the input layer to the hidden layer, 5 weights for the bias of each neuron in the hidden layer, 15 weights connecting the hidden layer to the output layer, and 3 weights for the bias in the output layer.

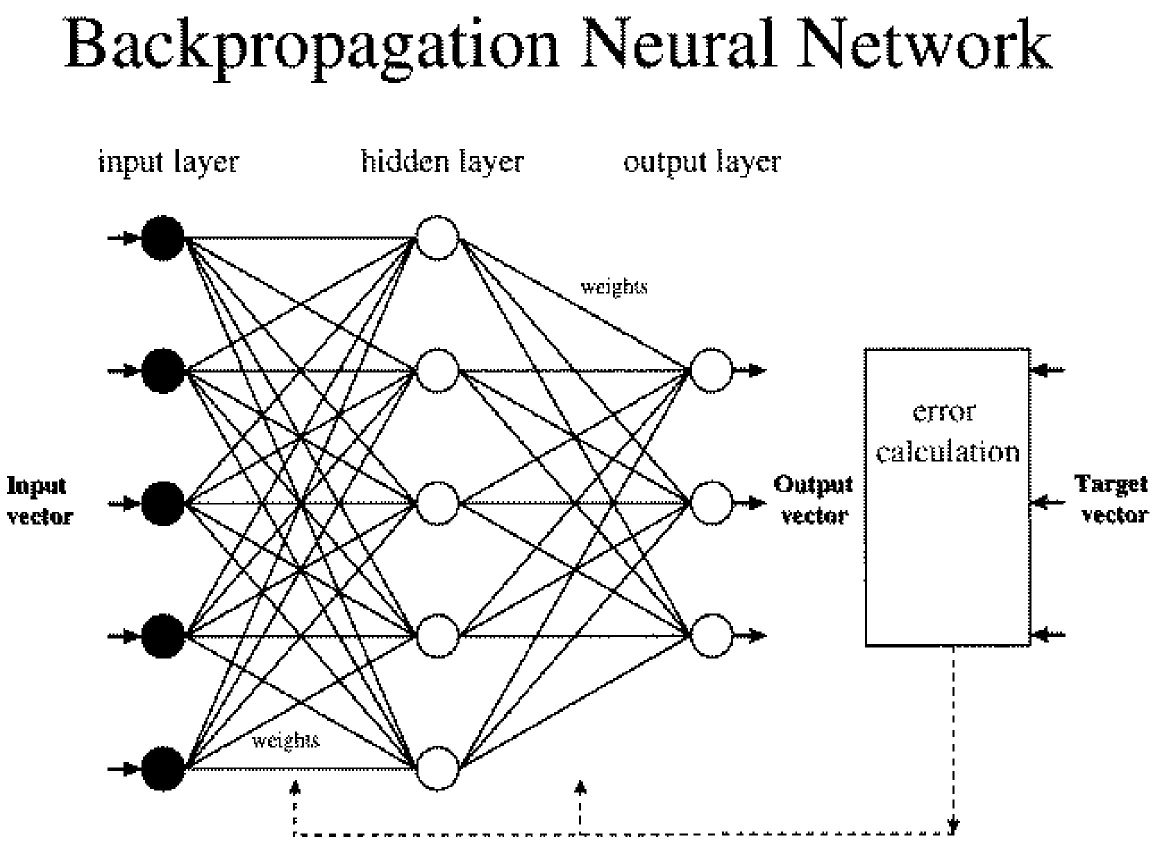

FIG. 1. Schematic representation of a backpropagation neural network defined with an input layer of five neurons, one hidden layer with five neurons, and an output layer consisting of three neurons.

The neurons in the input layer do not perform any operation, they are the access gate to the data. Each neuron in the hidden layer sums the values of the input neurons multiplied by the weight of the connecting neurons. The bias of the hidden layer neuron is added to the sum. This sum is then filtered by a transfer function f, the standard sigmoid (Figure 2) defined by

$$f(x) = \frac{1}{1 + \exp(-x)},$$

which acts as a gain function and outputs a value between 0 and 1. The same operation is repeated by each hidden layer.

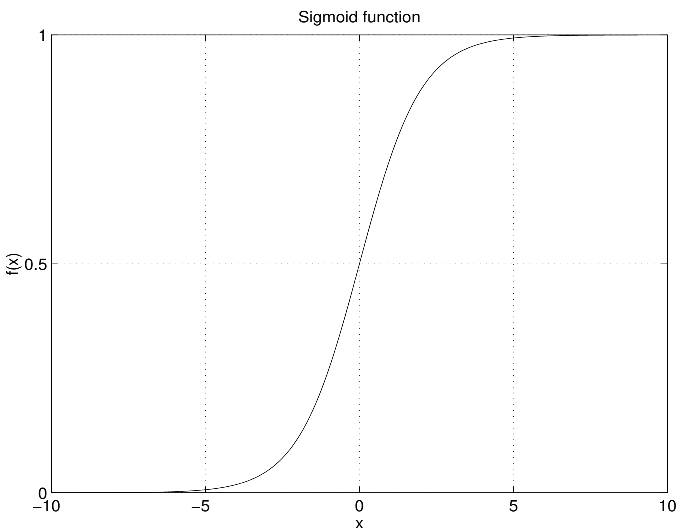

FIG. 2. Transfer function (sigmoid) applied by the neurons in the hidden layer.

The neurons in the output layer compute the sum over each neuron in the precedent hidden layer, multiplied by the weights connecting the hidden layer to the output layer, and add the bias. Depending on which version of the BPNN is considered, the neurons in the output layer may or may not apply the transfer function.

For classification, the transfer function is applied by the neurons in the output layer, so that each neuron in the output layer produces a value between 0 and 1.

The prediction version of the BPNN does not apply the transfer function in the output layer. Each neuron in the output layer produces the sum over the neurons in the precedent hidden layer affected by the connections weights, plus the bias of the output neuron.

Refering to the BPNN showed in Figure 1, for an input value at the i^{th} neuron of the input layer, denoted I_i, the *classification* BPNN produces at the k^{th} neuron of the output layer the value O_k, given by

$$O_k = \sum_j f(\sum_i I_i w_{ij} + b_j) w_{jk} + b_k,$$

even though the *prediction* BPNN produces

$$O_k = f(\sum_j f(\sum_i I_i w_{ij} + b_j) w_{jk} + b_k),$$

where w_{ij} are the weights connecting the input layer to the hidden layer, w_{jk} the weights connecting the hidden layer to the output layer, b_j the bias of the j^{th} neuron of the hidden layer, and b_k the bias of the k^{th} neuron of the output layer.

The network is trained to produce a targeted output vector, T, when fed with an input vector I. During the training phase, the weights of the network are adjusted in such a way as to minimize the misfit $\mathcal{E}$ defined as the mean square error between the n^{th} targeted output vector $\mathbf{T^n}$, of components T_k^n, and the n^{th} calculated output vector $\mathbf{O^n}$, of components O_k^n,

$$\mathcal{E} = \frac{1}{N_{tr} N_{ne}} \sum_{n=1}^{N_{tr}} \sum_{k=1}^{N_{ne}} (O_k^n - T_k^n)^2.$$

In the expression of the misfit $\mathcal{E}$, N_{ne} denotes the number of neurons in the output layer and N_{tr} the number of training pairs. The n^{th} training pair is composed of an input training vector $\mathbf{I^n}$ and a corresponding targeted output vector $\mathbf{T^n}$.

When the network is trained for *classification*, each output neuron corresponds to a different "class" or cluster. The network is trained to assign an input training vector to a specific "class". The neurons of the output layer corresponding to the targeted "class" is trained to equal one. All the other neurons are trained to equal zero.

Training the network

The weights are initialized to random numbers ranging between -1 and 1. The BPNN is trained by iteratively adjusting the weights of the connections in such a way that the error function is minimized. The original training algorithm, often called "delta rule" in the literature, is a steepest descent algorithm which, at each iteration, updates the weights by propagating the error back into the network's weights. Steepest descent algorithms require thousands of iterations over the training process before reaching an acceptable solution. We significantly improved the convergence rate of the training process by minimizing the misfit using a BFGS optimization program of the Hilbert Class Library.

The transfer function causes nonlinearity in the optimization problem. This nonlinearity takes the form of local "plateaus" in the error function, where the error is quasi-invariant. This occurs when the sigmoid function is at saturation, that is to say when the output becomes invariant to a small change in the weights of the hidden layer. A consequence is an extremely slow convergence rate.

If the initialization of the weights is close enough to the solution, the algorithm will eventually converge towards the global minimum of the error function.

Instead of implementing a global optimization algorithm, we adopted a pragmatic optimization strategy which consists in looping over random initializations of the weights until the error function gets smaller than a threshold or the number of re-initialization reaches a maximum in which case the best solution is kept.

When the network is trained, i.e. when the convergence conditions are met, the network is ready to generalize with new data sets for *prediction* or *classification*.

Running the BPNN code

First, select the version of the BPNN to be implemented by entering "P" for *prediction* or "C" for *classification*.

The program requests a job name. All the I/O files of the application start with the job name, for example "*job*". To run the program requires two parameter files: *job*_**param.dat** and *job*_**exec.dat**, and two data files: *job*_**train.dat** and *job*_**run.dat**.

The BPNN is defined in the file *job*_**param.dat**. These parameters remain invariant during the training process, i.e. the number of layers and neurons per layer cannot be changed during the training of the network. The parameters file *job*_**param.dat** also defines the number of training and running data.

The optimization parameters such as maximum number of random re-initializations and convergence threshold are set in the file *job*_**exec.dat**.

The network is constructed and the total number of weights displayed for information.

Then, the choice is offered to train the network or to run a trained network. If training the network is selected, the program reads the training data from the file *job*_**train.dat**.

The option is offered to train iteratively over randon re-initialization of the weights or to initialize the weights values by reading from a file. The network trains until the convergence criteria are met. The misfit is displayed at each iteration of the BFGS algorithm. At each re-initialization of the weights, the initial values of the weights are saved in files *job*_**weights_init.tmp**, and the values of the weights at convergence are saved in file *job*_**weights_final.tmp**. The best estimate of the weights, i.e. corresponding to the minimum misfit, is saved in file *job*_**weights_optimum.dat**.

A log of the training process is recorded in file *job*_**exec.out** and provides the initial and final misfit at each iteration of the random re-initialization. The program also generates an output information file *job*_**stat.dat** listing the job parameters.

When the option of running the network is selected, the program requests the weights file name. The input data fed into the network are read from the file *job*_**run.dat**. The output vectors calculated by the network are saved in file *job*_**result.dat**.

Acknowledgement and Disclaimer: This work is funded by the Strategic Research Center of Mobil Technology Company. The development of the Hilbert Class Library, used in this project, was partially supported by the National Science Foundation under grant numbers DMS 9404283 and DMS 9627355; the Office of Naval Research under grant number N00014-96-1-0156; the Air Force Office of Scientific Research, Air Force Materials Command, USAF, under grant number F49620-95-1-0339; and The Rice Inversion Project. TRIP Sponsors for 1997 are Advance Geophysical, Amoco Production Co., Conoco Inc., Cray Research Inc., Discovery Bay, Exxon Production Research Co., Interactive Network Technologies, Mobil Research and Development Corp., and Shell International Research. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the Office of Naval Research, the National Science Foundation, or the US Government.

REFERENCES

Bishop, C. M., 1996, *Neural Networks for Pattern Recognition*: Oxford, Clarendon Press.

Gockenbach, M. S., and Symes, W. W., 1996a, The Hilbert Class Library: a library of abstract C++ classes for optimization and inversion: *Computers and Mathematics with Applications*, 32:6.

Gockenbach, M. S., and Symes, W. W., 1996b, The Hilbert Class Library: a library of abstract C++ classes for optimization and inversion: Fourth Object Oriented Numerics Conference: OON-SCI'96 Proceedings.

Polak, E., 1971, *Computational Methods in Optimization.* New York: Academic Press.

Rogers S. J., Fang, J. H., Karr, C. L., Donald, D., and Stanley, F. A., 1992, Determination of lithology from well logs using a neural network: *AAPG Bulletin*, 76:731-739.

Wasserman P. D., 1993, *Advanced methods in neural computing.* New York: Van Nostrand Reinhold.