The Cascade-Correlation Neural Network Growing Algorithm using the Matlab Environment

José Demisio Simões da Silva
Instituto Nacional de Pesquisas Espaciais
Laboratório de Computação e Matemática Aplicada
LAC-INPE
12202-010 - São José dos Campos - SP
E-mail: demisio@lac.inpe.br

Abstract
This paper describes the cascade-correlation algorithm as proposed by [2], as well as its implementation in the Matlab environment. The cascade-correlation is presented as a neural network growing technique which allows one to gradually build a network architecture without the need to redefine the number of neurons to be used in a feed forward neural network if it does not converge. Some results are presented for three different experiments that show the capability of the algorithm.

Key words: Cascade-correlation, neural network, growing technique.

1 - Introduction

One of the major issues in designing a neural network is to obtain the appropriate size of the network to perform a certain task. Different tasks and specific applications require different network sizes. Small networks may require a lot of time for training and/or result in an insufficient number of free parameters to capture all the necessary features from the input data to make the categorization possible.

As it is not possible to guess the appropriate size of a neural network (minimum size and good performance [3]) to perform the task at hand, we may spend some time trying different architectures varying the number of free parameters and/or the number of hidden layers in the network. However, there are techniques which make it possible to start with: a) a huge network and prune it as it is trained and tested over some amount of data; or b) a small network, monitored for the solution of the problem and gradually increased to a final appropriate size [3]. Also, a combination of both pruning and growing techniques can be used [1].

In growing network two approaches can be considered: 1) input space partitioning and 2) prototype selection [1]. The first is concerned with the partition of the input space using the fewest hyperplanes as possible, which is equivalent to designing a neural network with the smallest number of nodes as possible. Some algorithms that provide mechanisms to gradually construct neural networks are described in [1].

The prototype selection involves the use of a pattern classifier network consisting of a cascade of two subnetworks: the matching score (MS) and the MAXNET or a winner-take-all network which is a recurrent network [1]. In training a pattern classifier network, a large set of input of different categories is given, and the network creates a smaller set of prototypes that statistically represent patterns of different categories. The MS subnetwork stores the prototypes for matching with input testing pattern when it scores the matching. The prototype
selection reduces the complexity and increases the generalization ability of the classifier [1]. Given that the number of categories in a classification problem is not known and/or the training inputs are produced on-line or in real time, the classification network should select or create a new prototype incrementally in the training process. This is equivalent to the process of node growing, once each prototype is represented by a node in the MS network [1].

2 - Development

This paper shows the implementation aspects of the cascade-correlation algorithm in the Matlab environment. The cascade-correlation algorithm can be classified as an input space partitioning scheme mentioned before.

The algorithm was proposed by [2] as an architecture and a supervised learning algorithm for artificial neural networks. It automatically trains and adds new hidden neurons one by one, leading to a final multi-layer structure. In [2] are the reasons they decided to develop the cascade-correlation algorithm.

This paper addresses the description of the algorithm as well as the implementation aspects and suggests the use of another growing scheme for adding the hidden neurons.

Two ideas are behind the algorithm: a) the cascade-architecture and b) the learning algorithm [2]. The cascade-architecture means neurons are added one at a time and with their input connections frozen. The learning algorithm creates and installs the new hidden neurons in the existing network. Figure 1 illustrates the cascade-architecture.

![Figure 1 - Cascade-Architecture with three added hidden neurons.](image)

To build an architecture as in figure 1, the algorithm starts with a network with the input and output layers only. As the training proceeds the algorithm searches for the completion of learning through the observation of the error achieved so far. If the target error is not achieved within a period of training cycles established as a patience, in which no significant error reduction has occurred [2], than the network growing module adds a hidden neuron with its input connections frozen. The network is then retrained and if the error reduction is not significant again within the reinitiated patience period, a new neuron is added. The process continues until the training is successful when a final architecture is obtained.

The addition of a neuron involves the analysis of a candidate neuron which is connected to all the neurons in the input layer and all the pre-existing hidden neurons [2]. The candidate neuron is not connected to the neurons in the output layer at first. Instead, its output is compared to the actual network output through a correlation analysis, which tells how correlated the candidate neuron output is to the actual network output. It is desirable to have the most correlated output as possible. During this analysis the candidate's input weights are adjusted with the aim of maximizing the correlation S (the covariance is used in [2]) between the candidate's output and the network output. S is defined as:

\[ S = \sum_{p} \sum_{o} (v_p - \bar{v})(E_{p,o} - \bar{E}_o) \]

where: \( o \) is the network output where the error is measured; \( p \) is the training pattern; \( v \) is the candidate neuron’s output value; \( E_{p,o} \) is the residual error measured at the output \( o \); \( \bar{v} \) and \( \bar{E}_o \) are the averaged values of \( v \) and \( E_o \) over all patterns.

The maximization of \( S \) is done through a gradient ascent calculated after the computation of the following partial derivative,

\[ \frac{\partial S}{\partial w_i} = \sum_{p,o} \sigma_o (E_{p,o} - \bar{E}_o) a_p I_{i,p} \]

where : \( \sigma_o \) is the sign of the correlation between the candidate's output and the network output; \( a_p \) is the derivative for pattern \( p \) of the candidate neuron’s activation function with respect to the sum of its inputs, and \( I_{i,p} \) is the input the candidate neuron receives from neuron \( i \) for pattern \( p \). The computation of \( \frac{\partial S}{\partial w_i} \) is achieved by the use of the chain rule.

After computing \( \frac{\partial S}{\partial w_i} \) it is possible to apply a gradient ascent algorithm to maximize \( S \). This process is equivalent to training a two layer network and can be performed using the delta learning rule. When \( S \) stops improving, the new neuron is added to the current network with its input weights frozen, a least square solution is used to solve for the output layer weights, the error is checked, and the cycle is continued. This process is shown in the following flow chart.
In order to accelerate the process, [2] suggest that a pool of candidate neurons be used, each with different random initial weights. The input signals and the same residual error for each training pattern are seeing by all of them. These candidates can all be trained in parallel because they do not interact with one another or affect the active network during training. After all the correlations stop improving the neuron with best correlation is chosen and installed in the network. Besides the use of a pool of candidate neurons, the maximization process can be accelerated if a heuristic is used to modify an adaptive parameter multiplying the gradient in the weight update phase. This heuristic can deal with the speed at which $S$, the covariance (or the correlation), is maximized.

3 - Implementation and Results

This implementation of the cascade-correlation algorithm uses functions available in the Matlab Neural Network toolbox [5] as well as other functions constructed using the information available in [2]. Following is the description of the functions that form the cascade-correlation algorithm.

The algorithm is implemented with three modules: 1) module1.m - initializes the network from the input/output pattern pairs given and sets the parameters needed for training. It also controls the installation of candidate neurons and the monitoring of the target error in order to decide if a new neuron has to be added. 2) module2.m - is a function that receives the input vectors and the desired outputs as well as the learning parameters information and implements the training algorithm chosen by the user to perform the learning phase. 3) module3.m - performs the correlation analysis of a pool of 5 different candidate neurons and maximizes these correlations until no more improvement is observed. It then selects the candidate with best correlation to be added to the network. A fourth module, module4.m, simulates the cascade correlation algorithm to test the final network.

Three different experiments with the cascade-correlation algorithm are described. The experiments are written in script files. script1.m implements the XOR problem, script2.m implements a function approximation experiment, and script3.m is an application in which the network is supposed to learn the model of a system. The testing of the resulting network is performed through the scripts tstscrp1.m, tstscrp2.m, and tstscrp3.m, respectively.

For the XOR problem a initial network was built with two inputs and one output. The actual network had only two layers at first. The training resulted in a network with 1 hidden neuron added. Figure 2 shows a plot of the error after two epochs of training.

![Figure 2 - Training resulted with 1 hidden neuron added by the cascade-correlation algorithm.](image-url)
For checking validation of the trained network the following data was provided:

\[
\text{test} = [-1 -1; -1 1; 1 -1; 1 1]
\]

The function \( F = \sin(t) + 0.2 \times \text{rand(size}(t)) \) for \( t=0 \) to \( 32 \) incremented 1 by 1, was given to a network.

Figure 3 - Comparison of the actual vectors and the computed ones. \( x \) represents the actual values; \( o \) represents the computed values.

Another experiment tested was a function approximation. The function \( F = \sin(t) + 0.2 \times \text{rand(size}(t)) \) for \( t=0 \) to \( 32 \) incremented 1 by 1, was given to a network.

The training resulted in 30 hidden neurons added to the actual network. Figure 4 shows the training error curve obtained.

A plot showing both the desired and the calculated outputs is presented in figure 3.

After training the network was validated with the same training data. The error achieved was 0.000234. Figure 5 shows a comparison between the actual and approximated functions.

The last experiment deals with the modeling for sensor and associated instrument channel calibration validation. The experiment uses a set of data with 9 inputs to make the network learn the desired model of the system. This network is used for sensor and associated instrument channel calibration validation. The idea is to have the network in parallel to the actual system and let it learn the features of the system. Thus the validation provides a condition-alternative to cycle-based maintenance calibration validation, which can provide a condition-based alternative to cycle-based maintenance in nuclear power plants as well as many other complex systems.

Figure 6 shows a block diagram of the assembled system.

After the training the network is tested for accuracy and generalization using a testing set. Figure 7 shows the training results with 25 hidden neurons.
Activation of the network with the same training data leads to the validation phase. Figure 8 qualitatively shows the validation result.

The averaged error achieved was 0.2454 which is close to the error achieved in training the network.

For generalization tests a testing set was created from the same original data used in the creation of the training set. Both sets are mutually exclusive. And the testing set has a greater number of components. Figure 9 illustrates the testing data.

For generalization tests a testing set was created from the same original data used in the creation of the training set. Both sets are mutually exclusive. And the testing set has a greater number of components. Figure 9 illustrates the testing data.

The averaged error achieved in the generalization test was 1.6119 which is 6.5685 times bigger the averaged error achieved in the activation using the training data. However, as can be seen in figure 10 the general model of the systems was learned by the network. Thus the generalization can be considered to be a good one, once the aim was to learn the model of the system.

In the experiments, the error targets were dependent upon the number of feature vectors in each training set.

module1.m, the main module of the algorithm, receives the data, the learning parameters, and the chosen algorithm to be used to train the network in module2.m. The training algorithms are the ones available in the Matlab toolbox: solvelin, trainbpx, and trainlm. Other algorithms can also be added to increase the number of possibilities to train the network. All the experiments performed used the solvelin.m function or the modified version
solvlin1.m, which uses the Matlab pseudo inverse function pinv.m, as suggested in [4], to avoid numerical instabilities and reduce the chance of overfitting noise. The choice between them depends on the size of the training set.

The patience under which the error is analyzed to decide if a new neuron has to be added, is the maximum number of epochs an algorithm has to learn. In case solvlin.m is chosen as the learning algorithm, this patience does not have any effect and a new neuron is added whenever the network does not find the answer to the problem.

An issue one may be concerned with, is the time taken by the algorithm to maximize the correlation of the candidate neurons to the actual network output. This time can be reduced if there is a maximum number of epochs set for this process (in the experiments it is set to 350). In the implementation addressed by this paper, the correlations of the pool of neurons are checked for the maximum which is then analyzed. If it decreases from one epoch to the next, then the last pass is not considered and the corresponding neuron is installed. This process can enhance the maximization phase, which is dependent upon the size of the training set.

Another issue that can be addressed is the number of neurons added to the actual network. [2] suggests an alternative implementation to add two neurons at a time. This strategy may significantly reduce the number of neurons added to the final network.

4 - Conclusions

The results found show that, except for the first experiment which involves a very small amount of data (final network with 1 hidden neuron added only), the final networks were very big with 30 hidden neurons in the function approximation experiment, and 25 hidden neurons in the modeling experiment. However, in the three experiments it was possible to get to a final network that solved the problem at hand, without the need to try different architectures and re-do the whole experiment to achieve an appropriate network size for the problem. An explanation for the high number of hidden neurons is probably the fact that each one stores certain features of the data and then its connections to the input neurons are frozen when it is installed in the network. Thus the added neurons function as preprocessing of the data, solving the existing non-linearities in the data.

Alternatives algorithms for the maximization process can be used. The XOR and function approximation experiments were also performed using other training providing similar results to the ones reported in this paper.

The cascade-correlation algorithm seems to be a good alternative to design networks which will take very big training sets such as the modeling experiment reported. It should be used whenever one desires a network but has no idea as to the number of hidden neurons or hidden layers to start with. In case one knows the size of the network, then it is preferable to use another training approach that does not involve the maximization process, which can mean a problem to the network designer, depending on the application he has at hand.

[2] in section 4 discusses the advantages of using the cascade-correlation algorithm. One aspect mentioned is the possibility of using different non-linearities for the added hidden neurons. This possibility may, however, increase the amount of time needed for the maximization process, once it would be necessary to try each candidate twice for the correlation analysis, thus making the algorithm non-attractive for applications with small training sets such as experiment the XOR problem as described in the paper.

Acknowledgments

This work was sponsored by the CNPq under a RHAE fellowship. The author is grateful to Chris Black from the Nuclear Engineering Department at the University of Tennesse for providing the data for the modeling experiment.

References:


