Automatic Generation of a Neural Network Architecture Using Evolutionary Computation

E. Vonk *, L.C. Jain **, L.P.J. Veelenturf *, and R. Johnson ***

*Control, Systems and Computer Engineering Group (BSC), Laboratory for Network Theory, Department of Electrical Engineering, University of Twente, Postbus 217, 7500 AE Enschede, The Netherlands.
Tel. : +61 8 302 3984
Fax : +61 8 302 3384
Email: 940021a@lux.levels.unisa.edu.au

**Knowledge-based Engineering Systems Group, School of Electronic Engineering, University of South Australia, Adelaide, The Levels, 5095, Australia.
Tel. : +61 8 302 3315
Fax : +61 8 302 3384
Email: etlcj@lv.levels.unisa.edu.au

***Weapons System Division, Defence Science and Technology Organization, PO Box 1500, Adelaide, Salisbury, 5108, Australia.
Tel. : +61 8 259 5127
Fax : +61 8 259 5688
Email: rpj@mogwah.dsto.gov.au

Keywords - neural networks, evolutionary computation, genetic algorithms, genetic programming

Abstract

This paper reports the application of evolutionary computation in the automatic generation of a neural network architecture. It is a usual practice to use trial and error to find a suitable neural network architecture. This is not only time consuming but may not generate an optimal solution for a given problem. The use of evolutionary computation is a step towards automation in neural network architecture generation. In this paper a brief introduction to the field is given as well as an implementation of automatic neural network generation using genetic programming.

1. Introduction

The performance of a neural network depends on the network architecture. Its performance, depending on the given task, includes properties like learning speed and generalization capability. For example, a certain neural network topology used for a classification task may have learned to classify the training set correctly, but this says nothing of the network's performance on data outside the training set. This depends to a great deal on the topology of the network.
The automatic generation of a neural network architecture is a useful concept as in many applications the optimal architecture is not a priori known. Often trial and error is done before a satisfactory architecture is found. Construction-deconstruction algorithms can be used as an approach but they have several drawbacks. They are usually restricted to a certain subset of network topologies and as with all hill climbing methods they often get stuck at local optima and therefore may not reach the optimal solution. Using evolutionary computation as an approach to the generation of neural network architectures these limitations can be overcome. Sometimes instead of evolutionary computation, the term evolutionary algorithms is used but in this paper this is reserved for a special kind of evolutionary computation.

The organization of this paper is as follows. Section 2 introduces briefly evolutionary computation techniques used in neural network design. Section 3 describes the implementation of neural network architecture design using genetic programming and in section 4 the conclusions and future work is presented.

2. Evolutionary Computation in Neural Network Design

Evolutionary computation can be divided into three different approaches:

- genetic algorithms
- genetic programming
- evolutionary algorithms

Of these mostly genetic algorithms have been used in neural network design; see [4] for an extensive overview. Work within this field mainly differs in the representations of the neural network topologies used. Representations used can be roughly divided into 'strong representation' and 'weak representation' schemes. When a strong representation is used, the

0-8186-7085-1/95 $04.00

chromosomes of the genetic algorithm directly encode the neural network. In a weak representation the chromosomes represent more abstract terms like 'the number of hidden neurons'.

Strong representations include the use of connectivity matrices and graph grammars. Connectivity matrices have proven to be unsuccessfull when simple toy problems were scaled up to more real world problems. This is because of the enormous increase in chromosome length and accordingly in the search space when larger networks need to be represented.

Graph grammars have proven much more successfull because they use much shorter chromosome lengths and the networks generated are highly structured [5],[6],[7].

Genetic programming [1] offers a third approach to a strong representation scheme. This approach is described in [1] and [2], and consists of directly encoding a neural network in the genetic tree structure used by genetic programming. This approach differs from the above methods in that the neural network topology as well as the values of the weights are encoded in the chromosomes and that they are trained simultaneously.

3. Implementation of Neural Network Design using Genetic Programming

The last approach described in the above section is implemented here. It is founded mainly on [1] and [2], where the genetic programming paradigm showed good results when it was applied to the generation of a neural network that could perform the one-bit adder task.

A public domain genetic programming system called GPC++, version 0.40, was used [3]. It is a software package written in C++ by Adam P. Fraser, University of Salford, UK. Several alterations were made to use it for the application of neural network design. The GPC++ system uses Steady State Genetic Programming (SSGP) as discussed in §2.2. The probability of crossover is 100%; the new population is generated using the crossover operator only. Then on a certain percentage of members mutation is performed. The crossover operator swaps randomly-picked branches between two parents, but creates only one offspring. There is no notion of age in the SSGP system, which means that after a new member is created, it can be chosen immediately after to create a new offspring.

3.1. Setup

We have basically used the same setup as described in [1],[2]. A neural network is represented by a connected tree structure of functions and terminals. Both the topology as well as the values of the weights are defined within this structure. In this approach no distinction is made between the learning of the network-topology and its weights; it is done within the same algorithm.

The terminal set is made up of the data inputs to the network (D), and random floating point constant atom (R). This atom is the source of all the numerical constants in the network and these constants are used to represent the values of the weights. The neural networks generated by this algorithm are of the feed-forward kind. The terminal set for a two-input neural network is for example {D,R}, where D = {D0,D1}.

In [2] the function set is made up of six functions: {P,W,+,-,*,%}. P is the processing function of a neuron; it performs a weighted sum of its inputs and feeds this to a processing function (e.g. linear threshold, sigmoid). The processing function takes two arguments in the current version of the program; i.e. every neuron has two inputs only. The weight function, W, also has two arguments. One is a subtree made up of arithmetic functions and random constants that represents the numerical value of the weights. The other is the point in the network it acts upon which is either a processing unit (neuron) or a data input. The four arithmetic functions, AR = {+,-,*,%}, are used to create and modify the weights of the network. All take two arguments. The division function is protected in that it returns zero in the case of a division by zero.

After some experimentation it was found that for the problems under investigation, the system actually worked much better if the arithmetic functions were left out. The values of the weights are represented by a single random constant atom and their values can only be changed by a one-point crossover or mutation performed on this constant atom.

The output of the genetic program is a LISP-like S-expression, which can be translated into a neural network structure made up of processing functions (neurons), weights and data inputs. Initially no bias units were implemented.

The name given to this implementation of neural network design using genetic programming is GPNN.

3.2. Example of a Genetically Programmed Neural Network

An example of a GPNN-output is the following neural network which performs the XOR function.

```
( P ( W ( P ( W -0.65625 D1 ) ( W 1.59375 D0 ) ) 1.01562 )
  ( W 1.45312 ( P ( W 1.70312 D1 ) (W -0.828125 D0 ) ) ) )
```

The graphical representation and the corresponding neural network are illustrated in Figure 1 and Figure 2.

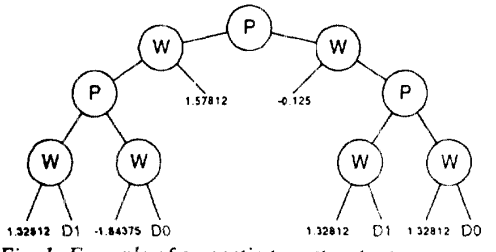

Fig. 1: Example of a genetic tree structure generated by GPNN

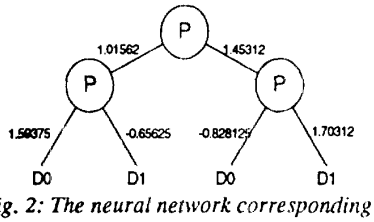

Fig. 2: The neural network corresponding to the structure of fig. 1

3.3. Creation and Crossover Rules for GPNN

In the standard GP paradigm, there are no restrictions concerning the creation of the genetic tree and the crossover operator, except a user-defined maximum depth of the tree. For the use of neural network design, several constrictions on the creation as well as the crossover operator have to be made.

3.3.1. Creation rules

The creation rules are:

- the root of the genetic tree must be a "list" function (L) of all the outputs of the network
- the function below a list function must be the Processing (P) function
- the function below a P function must be the Weight (W) function
- below a W function, one of the functions/terminals must be chosen from the set {P,D}, the other one must be {R}

These creation rules make sure the created tree represents a correct neural network. The root of the tree is a list function of all its outputs while the leafs are either a data signal (D) or a numerical constant (R). This tree can then be translated into a neural network structure as in Figure 2.

3.3.2. Crossover rules

The crossover operator has to preserve the genetic tree so that it still obeys the above rules. This is done by structure-preserving crossover which has the following rule: the points of the two parent-genes between which the crossover is performed (the branches connected to these points are swapped) must be of the same type.

The types of points are:

-a P function or a D terminal
-a W function
-a R terminal

In [2] P functions and D terminals are treated as being of different types, which means a branch whose root is a P function can never be replaced by a D terminal and vice versa.

3.4. Implementation of the Fitness Function

The fitness function is calculated as a constant value minus the total performance error of the neural network. A training set consisting of input and target-output patterns (facts) needs to be supplied. The error is then calculated as:

$$Error = \sum_{f=1}^{NumFacts} \sum_{i=1}^{NumOutputs} (Output(i) - Desired(i))^2$$

Since a lower error must correspond to a higher fitness, the fitness function is then calculated as:

$$Fitness = Error - MaximumError$$

The maximum performance error is a constant value equal to the maximum error possible, so that a network that has the worst performance possible on a given training set (maximum error) will have a fitness equal to zero. When a linear threshold function is used as the neurons' processing function, only output values of '0' or '1' are possible. The range of fitness values is then very limited and it is impossible to distinguish between many networks. In order to increase this range the output neuron could be chosen to have a continuous sigmoid processing function.

In using a supervised learning scheme, there are many other ways to implement the fitness function of a neural network. Instead of the sum of the square errors, for example, we could use the sum of the absolute errors or the sum of the exponential absolute errors. Another definition of the fitness could be the number of correctly classified facts in a training set.

The fitness function could also reflect the size (= structural complexity) and/or the generalization capabilities of the network. For example smaller networks having the same performance on the training set as bigger networks would be preferred, as they have better generalization capabilities in general. The generalization capability of a network could be added to the fitness function by performing a test on test data that lies outside the training data.

3.5. Experiments with GPNN

The GPNN algorithm has been implemented using the code of GPC++ with several alterations / additions. The neurons in the resulting neural networks initially did not have bias units. The fitness function used was the total performance error over the training set multiplied by a factor to increase the range. The fitness value was then made into an integer value as this is required by the GPC++ software. The mutation operator was implemented so that it only acted on terminals, not on functions. The maximum depth of a genetic tree in the creation phase was set to 6. During crossover, the genetic trees were limited to a maximum depth of 17. These values were used as a default value by Koza [1], to make sure the trees stay within reasonable size.
Simulations have been done on automatically generating a neural network architecture for the XOR problem, the one-bit adder problem and the intertwined-spirals problem.

3.5.1. The XOR problem

Our attempts were directed to find a neural network that correctly performs the XOR problem. The processing function used for the neurons was a simple threshold function: thres(x) = 1 if x > 1, 0 otherwise.
The following statistics for the genetic programming algorithm were used :

Population Size:	500
Number of ADFs:	0
Max. depth at creation:	6
Max. depth at crossover:	17
Reproduction mechanism:	tournament
(tournament size = 5)	
Crossover:	100 %
Mutation:	10 %

After several runs were performed we found that a neural network which performed the given task occurred every time between generation 1 and generation 5. Figure 3 shows a solution that was found in a particular run in generation 5. All solutions found had a number of neurons ranging from 3 to 5. When the roulette wheel reproduction mechanism was used instead of the tournament mechanism, the convergence to a solution took on average 2 generations longer.

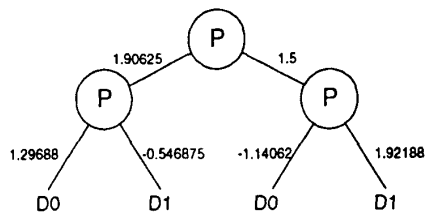

Fig. 3: a generated neural network that performs the XOR problem

The GPNN system was extended with a bias input to every neuron by means of an extra random constant (in the range [-4,4]) added to every P function. The effect of this on the XOR problem was a somewhat slower convergence. The reason might be that the search space is increased, while for a solution to this simple problem bias-inputs are not needed.
For this problem no ADFs were used, as they did not seem necessary for such a simple task.

3.5.2. The one-bit adder problem

It was then tried, as in [2], to find a solution to the slightly more difficult one-bit adder problem. The network has to solve the following task:

input	target output
0 0	0 0
0 1	0 1
1 0	0 1
1 1	1 0

In effect this means that the first output has to solve the AND function on the two inputs, and the second output the XOR function.

The same characteristics as used in the XOR problem were used. A solution to the problem was found on all 10 runs between generation 3 and generation 8. One of them is shown in Figure 4. The convergence is much faster than in [2], where a solution was only found after 35 generations also using a population of 500.

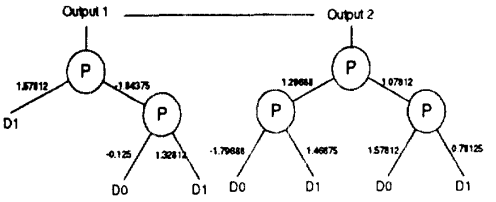

Fig. 4: a generated neural network that performs the one-bit adder problem

As can be seen from the figure, the neural network found is indeed made up of an AND and an XOR function. On average the generated neural networks had more neurons than just 5; the largest network found had 20.

3.5.3. The intertwined-spirals problem

The intertwined-spiral classification problem was tried as well as it is often regarded as a benchmark problem for neural network training. The training set consists of two sets of 97 data points on a two-dimensional grid, representing two spirals that are intertwined making three loops around the origin. A 2-input, 2-output neural network is needed.
The results were very poor. When the same settings as in the above experiments were used, not much more than half of the training set was classified correctly. Automatically Defined Functions (ADFs) were introduced, but no improvements were observed.

3.6. Discussion of GPNN

Restrictions that apply to the GPNN system are:

- There are quite severe restrictions on the network topologies generated: only tree structured networks are possible.
- The number of arguments of a function is always fixed; e.g. a processing function (a neuron) can and must only have two inputs.
- Because of the way the terminal set is stored in memory, only 255 different random floating point constants (R terminals) can be used. These values are chosen from the interval [-2,2].
- The learning of the topology and weights is done simultaneously within the same algorithm. This has the drawback that a neural network with a perfectly good topology might have a very poor performance and therefore be thrown out of the population just because of the value of its weights.

Some other application-independent problems in using GP are:

- How do you know what functions to include in the function set ? For example in the GPNN system only two functions are used : {P,W}. We could easily extend this function set. In order to decide on what functions are useful to the problem some knowledge of the final solution is needed.

- So far very little research has been done on the generalization capabilities of GP (and GA); i.e. the testing of the solution on data outside the training set. Problems similar to the ones in the training of neural networks apply: when to stop training, how to choose the training set and the problem of overfitting on the training data. In GP/GA a major obstacle is how to decide on what fitness measure to use, since there are so many varieties.

4. Conclusions and Future Work

Similar to the work in [2], it has been shown that the genetic programming paradigm can be used to generate a neural network that works on the task of the XOR problem and the one-bit adder. These very simple 'toy'-problems only show that the GPNN system actually works and care should be taken to draw any conclusions from them. It was found that the GPNN system does not scale up well to larger real world applications. This is mainly due to the restrictions of this approach described in §3.6. To make sure that neural networks with good topologies are not discarded, the learning of the topology and the learning of the weights should be seperated. The restriction of network topologies to tree structures is very severe. Many problems may be very hard or even impossible to solve using a tree structured neural network. Furthermore, the restriction on the number of arguments for a P function (i.e. the number of inputs to a neuron) is another severe drawback of GPNN.

Future work will focus on finding a neural network representation that does not suffer from these restrictions, for example graph grammars, and using this in the genetic programming or the genetic algorithm paradigm.

Acknowledgments

Thanks are due to the Defence Science and Technology Organisation, Salisbury, Adelaide, South Australia, for the financial support (contract number 340479). Edgar Vonk wishes to thank the Control, Systems and Computer Engineering Group (BSC), Laboratory for Network Theory, Department of Electrical Engineering, University of Twente, for the

permission to undertake this project in the Knowledge-based Engineering Systems Group, University of South Australia.

References:

[1] Koza, John R., *Genetic Programming, On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, 1992.

[2] Koza, J. R. and Rice, J. P., "Genetic Generation of both the Weights and Architecture for a Neural Network", *IEEE International Joint Conference on Neural Networks*, 1991.

[3] Fraser, Adam P., "Genetic Programming in C++, A Manual for GPC++", Technical Report 040, University of Salford, Cybernetics Research Institute, 1994.

[4] Schaffer, J. D., Whitley D. and Eshelman, L. J., "Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art", *IEEE International Workshop on Combinations of Genetic Algortihms and Neural Networks (COGANN-92)*, Baltimore, pp. 1-37, 1992.

[5] Gruau, F., "Genetic Synthesis of Boolean Neural Networks with a Cell Rewiting Developmental Process", *IEEE International Workshop on Combinations of Genetic Algortihms and Neural Networks (COGANN-92)*, Baltimore, pp. 55-74, 1992.

[6] Boers, E.J.W and Kuiper, H., "Biological Metaphors and the Design of Modular Artificial Neural Networks", Technical Report, Departments of Computer Science and Experimental and Theoretical Psychology, Leiden University, The Netherlands, 1992.

[7] Kitano, H., "Designing Neural Networks Using Genetic Algorithms with Graph Generation System", *Complex Systems*, Vol. 4, pp. 461-476, 1990.