Salzburg Interest Group on Integrated Systems

siGis-003-1995 November 29, 1995

netGEN - A Parallel System Generating Problem–Adapted Topologies of Artificial Neural Networks by means of Genetic Algorithms

Fachgruppentreffen Maschinelles Lernen der GI-Fachgruppe 1.1.3,
August 14–16, 1995, Dortmund, Germany

Reinhold Huber
reini@cosy.sbg.ac.at

Helmut A. Mayer
helmut@cosy.sbg.ac.at

Roland Schwaiger
rschwaig@cosy.sbg.ac.at

Department of Computer Science and Systems Analysis
University of Salzburg

Correspondence to:

Universität Salzburg
Institut für Computerwissenschaften und Systemanalyse
Jakob Haringer Straße 2
5020 Salzburg
Austria

netGEN - A Parallel System Generating Problem–Adapted Topologies of Artificial Neural Networks by means of Genetic Algorithms

Reinhold Huber Helmut A. Mayer Roland Schwaiger

Department of Computer Science and Systems Analysis

University of Salzburg

Abstract

Artificial neural networks (ANNs) have shown to perform satisfactorily for pattern recognition tasks. It has also been shown that ANNs are superior to some of the classical statistical methods in pattern classification, but little is known how to design the ANN. A genetic algorithm (GA) based method can be used to determine the ANN architecture for a specific task. We describe the netGEN system which is an existing implementation of a GA evolving ANNs in parallel. A simple pattern recognition task is solved so as to demonstrate the performance of netGEN.

1 Introduction

We describe a parallel system for the pattern extraction task by the ANN approach. Compared to statistical classifiers, such as Bayesian a posteriori classifiers, ANN classifiers have the important characteristic that no underlying distributional form for the class densities is assumed [Lip93].

Due to the independency from a priori statistical parameters and their inherent parallel nature, we decided to use ANNs to solve pattern recognition task. In particular we use a form of multilayer feed-forward networks for our ANNs and supervised learning through back-propagation.

We use a GA to adapt the ANN topology to our specific classification task. Each individual of a generation, a neural network in a population of ANNs, is trained and tested so as to evaluate ANN performance and to assign a fitness to the individual. After a selection process the individuals with the highest fitness are used to form the next generation. In the following we will use the term *netGEN* for the implementation of this method.

netGEN has been implemented as a parallel system where the learning of the ANNs is distributed among several workstations. The learning of the individual ANN is performed by an ANN simulator, namely the *SNNS* (Stuttgart Neural Network Simulator) [ZMV$^+$94]. For the message passing between the individual modules we employed the *PVM* (Parallel Virtual Machine) library [GBD$^+$94].

2 Evolution of Artificial Neural Networks

ANNs have been successfully applied to a wide variety of problems, but an analytical rule for the construction of an optimal topology of the ANN for a given problem has not been discovered yet. There are some rules of thumb and a few theoretical results, e. g. the proof that every continuous function can be approximated by an ANN with just one hidden layer [HSW89]. However, increasingly good approximation properties of an ANN, i.e. excellent learning of the training data set, will result in a loss of generalization capabilities, a problem known as *overfitting*. It has been found that ANNs with lower complexity, i. e. small number of neurons, low connectivity, show a better generalization performance than more complex networks [RWL94].

The space of possible network topologies is enormous, e. g. there are $2^{\frac{n(n-1)}{2}}$ ways to connect n neurons (all partial connections included), which increases even more with various learning method parameters. Specifically, as we restrict our ANN topologies to *Feed-Forward Networks*, we use the well-known method of *Error Back-Propagation* [RMtPRG86]. Considering this huge search space with very limited knowledge about the function to be optimized and with a clear biological analogon in mind, concepts of *natural evolution* have been applied to this problem.

In an effort to solve difficult problems by simulation of genetic processes and biological organisms, mainly the principles of evolution first clearly stated by Darwin in *The Origin of Species*, several branches of Evolutionary Computation evolved [Whi93], [BBM93]. The most common paradigm employed in the artificial evolution of ANNs is the field of Genetic Algorithms (GA) in which pioneering work was done by *J. H. Holland* [Hol75].

In order to structurize the various approaches to artificial evolution of ANNs we will use a slightly adapted taxonomy proposed in [BH95].

The *Genotype Encoding Scheme* can be broadly classified into two categories:

- Direct Encoding

  The number of neurons and the connections between specific neurons are explicitely encoded in the genotype which usually results in a *connection matrix*. The decoding effort to map the genotype to the corresponding *phenotype* is small.

- Indirect Encoding

  Here the genotype represents a set of rules for constructing the phenotype, e. g. the number of layers, the connectivity of the layers etc., so the decoding schemes are more complex.

The *Network Topology* can be roughly divided into two categories:

- Feed-Forward Networks (without feedback loops)
- Recurrent Networks (with feedback loops)

The *Variables of Evolution* may be grouped in two main categories:

- Evolving Topologies with Conventional Training

  The ANN topology and possibly the learning parameters of a learning method are subjected to artificial evolution.

- Evolving Topologies with Genetic Training

  The ANN topology plus the weights of the connections are evolved by means of artificial evolution.

The fourth taxonomy criterion is the *Application Domain* of the ANN. This easily extendable taxonomy is evidently simple, but nevertheless useful in classifying different approaches.

3 The netGEN System

The netGEN system has been implemented on a network cluster of SPARC and NeXT workstations in order to parallelize the learning phase of ANNs.

3.1 System Implementation

In figure 1 the main components, the Simple Genetic Algorithm (SGA) [SGE91], the Genotype Phenotype Mapping (GPM) and the Neural Network Manager (NNM), of the netGEN system are shown.

SGA generates a blueprint for each ANN, a genotype in GA terminology, which is processed by the GPM module mapping the ANN genotype to a phenotype adapted to the Neural Network Simulator. The mapping module's task is to prepare valid configuration files and to dismiss ANN topologies which cannot be processed by netGEN. If the topology is valid, a command, e.g. *learn*, is passed to the NNM module. The NNM interfaces to the ANN simulator, evaluates the fitness of each ANN and passes it back to the SGA, hereby, closing the evolutionary cycle. In the parallel implementation a set of NNM processes run on different machines and/or different processors.

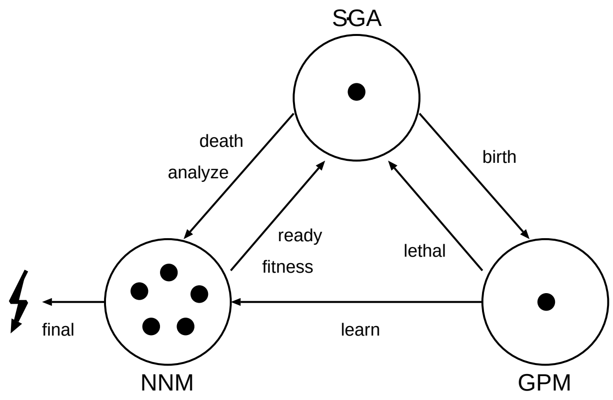

Figure 1: The netGEN system.

3.2 Implementation Details

netGEN uses the *SPMD* (Single Program Multiple Data) paradigm which is an extension to the model of computation introduced by *Flynn* where *SISD, MISD, SIMD* and *MIMD* machines are distinguished [AG89]. In SPMD the same program is run on the processors of a MIMD machine, in our case copies of NNM programs run on different machines which make up the Parallel Virtual Machine (PVM). A multiple data stream is observed because the ANNs are trained with different network configurations.

Amdahl noted that computing time can be divided into a serial and parallel portion and no matter how high the degree of parallelism in the parallel portion is, the speedup will be asymptotically limited by the serial portion [AG89]. Let t_S be the amount of time spent by one processor on serial portions of the program and t_P be the amount of time spent by one processor on parts of the program that can be done in parallel. The total execution time of a sequential program is defined as $t_{seq} = t_S + t_P$ and for a program using parallel computation $t_{par} = t_S + t_P/N$ where N is the number of processors. Therefore the speedup becomes:

$$Speedup = \frac{t_{seq}}{t_{par}} = \frac{t_S + t_P}{t_S + t_P/N}$$

Let us consider netGEN for the example given in the next section. We assume that we have 50 generations with 50 individuals each. The serial portion of netGEN consists the SGA and GPM processes, the NNM process could run in parallel. The overall time consumed by SGA and GPM is about 20 seconds. Training of one ANN on one machine for about 5000 epochs takes 30 seconds on average. We get the following estimation for t_{seq}:

$$t_{seq} = t_S + t_P = 20 + 50 * 50 * 30 = 75000 sec.$$

Whereas the parallel version using 15 machines achieves:

$$t_{par} = t_S + t_P/N = 20 + \frac{50 * 50 * 30}{15} = 5020 sec.$$

The resulting speedup of 14.94 is very close to the theoretical limit of 15. The speedup actually observed compared very well to this theoretically derived value.

Another limiting value for the speedup is the population size because a new generation can only be formed when each individual's fitness has been evaluated. Therefore, the speedup is not increased, if we use more machines than individuals, in this case 50. The theoretical maximum speedup, machines of equal speed of 30 seconds for NNM execution assumed, becomes 49.36.

Furthermore, we like to mention that we achieve *implicit load balancing* which means that more ANNs are passed to the faster machines within the parallel virtual machine. A simple time-out mechanism prevents the system from deadlock and performance reducing situations when waiting for results from crashed or very slow machines.

4 Natural Computation Methods

4.1 ANN Encoding

The encoding scheme for ANNs in netGEN is based on a *strong specification scheme* introduced by *Miller & Todd* ([MTH89]). Their encoding scheme - in the following called *Miller-Matrix* (MM) - sets up an adjacents matrix for the connections of the neurons and an additional column for the biases. Hence, for a network with n units the MM has dimension $n \times (n+1)$. The MM is constructed row by row from the bit-string genotype which just contains the lower triangle matrix restricting the topology of the ANN to feed-forward Networks. Crossover is achieved by exchanging rows of two parent individuals.

In our encoding scheme – called *Modified-Miller-Matrix* (MMM) – some substantial changes have been introduced.

First, we force the number of input/output units to stay constant during evolution, though, for some problems it would be definitely interesting to evolve the number of input neurons, as we often do not know which parameters of the input vector are actually important.

Second, we packed the bits of the lower triangle matrix into one bit string, so the GA operates on a "conventional" chromosome rather than on a matrix representation of a chromosome. This gives a more global effect of the crossover operator in comparison to the original scheme. Furthermore, all the "classical" crossover operators can be used.

Third, we introduced a *Neuron Marker* for the presence or absence of an unit. This marker is just valid for hidden neurons and is encoded in the main diagonal of the MMM. Use of the neuron markers allows a much better adaptation of the number of neurons towards a minimal-complexity network with better generalization capabilities than a more complex network. The original MM scheme enabled only an implicit change in the number of neurons, e. g. isolated units.

Fourth, we dropped the bias column, as the bias for each neuron is determined by the back-propagation training method.

The structure of our bit-string genotype of the ANN looks like:

$$\underbrace{x_1 x_2 x_3 \ldots x_h}_{s_d} \underbrace{y_1 y_2 y_3 \ldots y_s}_{s_r},$$

where s_d denotes the main diagonal sub string and s_r denotes the rows sub string.

4.2 GA Selection Methods

We achieved good results with *Tournament Selection*, where a number of n individuals (usually $n = 2$) is chosen randomly from the population and the individual with the highest fitness is selected and placed in the mating pool.

4.3 Fitness Functions with Penalty

The *standard fitness* of an ANN is the mean value of the differences of wanted to actual activation of the output units. A single misclassification within a test pattern results in a penalty which increases quadratically with each additional erroneous output.

4.4 Parameters

Following GA parameters have been used with all the experiments in this paper:
Crossover probability $p_c = 0.6$
Mutation probability $p_m = 0.005$
Crossover: 2-point
Selection Method: Tournament with 2

Design Taxonomy [BH95]:
Encoding: Direct

Network Topology: Feed–Forward
Variables of Evolution: Evolving Topologies with Conventional Training
Application Domain: Pattern Recognition

Back propagation training:
Number of epochs: 5000 (fixed)
Learning parameter $\eta = 0.2$
No momentum term

5 A Simple Example

This section describes the problem of detecting lines in a simple 3×3 black and white pattern. The input to the ANN is a block of 3×3 pixels, each corresponding to an input neuron. The output layer consists of four neurons activated depending on the line(s) contained in the input pattern. Figure 2 shows the four input patterns the ANN was trained on and combinations of trained and untrained pattern elements with the corresponding output layer:

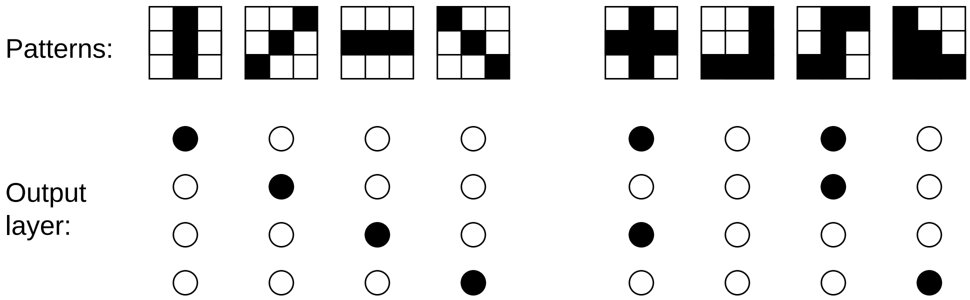

Figure 2: The problem of detecting lines.

In figure 3 the GA performance measures *on- and off-line performance* [DeJ75] are depicted for the adaptation of the ANN topology to the line problem. The on-line performance is given by:

$$P_{on} = \frac{1}{nT} \sum_{t=1}^{T} \sum_{i=1}^{n} f_i(t)$$

where n is the population size, i a specific individual, f the fitness function and T the total number of generations evaluated. The off-line performance is given by:

$$P_{off} = \frac{1}{T} \sum_{t=1}^{T} f^*(t)$$

where f^* is the best individual ever discovered by the GA up to generation t. The fitness function with penalty is given by:

$$f = \frac{\sum_{i=1}^{\tau} f_i'}{\tau(\sum_{i=1}^{\tau} e_i^2 + 1)}$$

where τ is the number of test patterns, f_i' the standard fitness of the ANN for the test pattern i and e_i the number of errors (misclassifications) for this test pattern.

The off-line performance stays around 0.48 (as the standard fitness of a "medium" ANN is around 0.9, this is an indication for ANNs with 1 misclassification) and rises when a zero-error net has been found. The performances of both experiments are quite similar, however, with neuron markers the best net generated has 21 neurons, 92 connections and a fitness of 0.955, whereas without neuron markers 29 neurons, 140 connections and a fitness of 0.943 characterize the best net. In all netGEN runs but those employing *uniform crossover* a zero-error net has been discovered.

For a comparison the fitness and *Mean Square Error* (MSE) of two ad-hoc ANN topologies with two hidden layers (fully connected) can be seen in figure 4. The left graph corresponds to a $9-4-4-4$

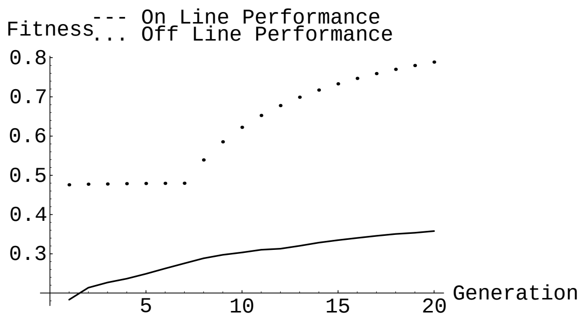

Figure 3: GA performance using the modified Miller–Matrix with (left) and without (right) neuron markers and a fitness function with penalty.

ANN, the right one to a $9-8-8-4$ ANN. The fitness of these two networks is significantly below those netGEN discovered. Furthermore, the overfitting problem – the fitness of the network even *decreases* with decreasing MSE – can be clearly observed which leads to the conclusion that the number of training epochs is of great importance for a (sub)optimal network and should be included in the genotype.

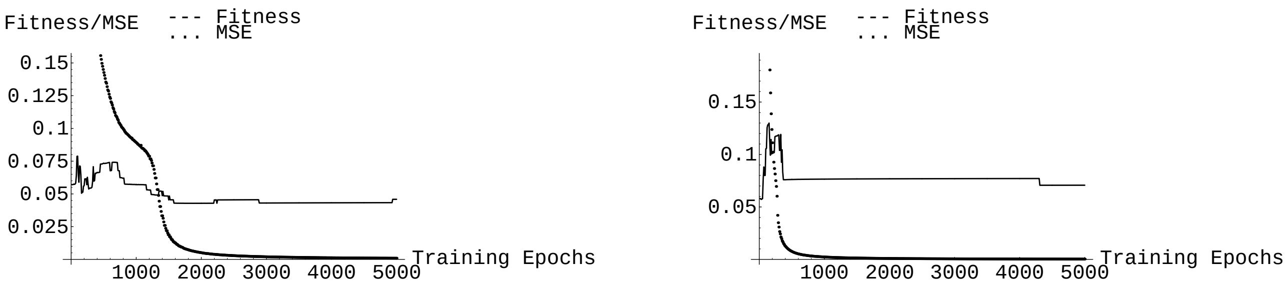

Figure 4: Fitness and Mean Square Error for ANNs with two hidden layers (fully connected) and 21 neurons (left), 29 Neurons (right)

6 Outlook

We have presented the netGEN system which has been applied to a variety of reference problems taken from comparable work with promising results. Further research interests and implementation issues include:

- Co-evolution of training epochs and the number of input neurons.
- Employment of a variety of genetic operators.
- An evolutionary approach to the generation of training sets for the ANN.
- Replacement of the ANN software simulation system.
- Enhancing the fault tolerance of netGEN in a heterogeneous workstation cluster.
- Application of the netGEN system to other problem domains, especially to landuse classification from remotely sensed imagery.

References

[AG89] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, 1989. ISBN 0-8053-0177-1.

[BBM93] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 1, fundamentals. *University Computing*, 15(2):58–69, 1993.

[BH95] Karthik Balakrishnan and Vasant Honavar. Evolutionary design of neural architectures – a preliminary taxonomy and guide to literature. Technical Report CS TR #95-01, Iowa State University, Department of Computer Science, Ames, Iowa 50011–1040, U.S.A., January 1995.

[DeJ75] K. DeJong. *The Analysis and Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, 1994.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1975.

[HSW89] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[Lip93] Richard P. Lippman. Neural networks, bayesian a posteriori propabilities, and pattern classification. In *Proceedings of the NATO Advanced Study Institute From Statistics to Neural Networks, Theory and Pattern Recognition Applications, Les Arcs, France*, June 1993.

[MTH89] Geoffrey F. Miller, Peter M. Todd, and Shailesh U. Hegde. Designing neural networks using genetic algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 379–384, San Mateo, California, 1989. Philips Laboratories, Morgan Kaufman Publishers, Inc.

[RMtPRG86] David E. Rumelhart, James L. McClelland, and the PDP Research Group. *Parallel Distributed Processing. Explorations in the Microstructure of Cognition*, volume 1: Foundations. MIT Press, 1986.

[RWL94] David E. Rumelhart, Bernard Widrow, and Michael A. Lehr. The basic ideas in neural networks. *Communications of the ACM*, 37(3):87–92, March 1994.

[SGE91] Robert E. Smith, David E. Goldberg, and Jeff A. Earickson. Sga-c: A c–language implementation of a simple genetic algorithm. TCGA Report 91002, The Clearinghouse for Genetic Algorithms, The University of Alabama, Department of Engineering Mechanics, Tuscaloosa, AL 35487, May 1991.

[Whi93] Darrell Whitley. A genetic algorithm tutorial. Technical Report CS-93-103, Colorado State University, November 1993.

[ZMV+94] Andreas Zell, Guenter Mamier, Michael Vogt, Niels Mach, Ralf Huebner, Kai-Uwe Herrmann, Tobias Soyez, Michael Schmalzl, Tilman Sommer, Artemis Hatzigeogiou, Sven Doering, and Dietmar Posselt. *SNNS Stuttgart Neural Network Simulator, User Manual*. University of Stuttgart, 1994.