SEARCHING NEURAL NETWORK STRUCTURES WITH L SYSTEMS AND GENETIC ALGORITHMS

Isto Aho, Harri Kemppainen, Kai Koskimies
Erkki Mäkinen and Tapio Niemi

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1997-15

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1997-15, DECEMBER 1997

SEARCHING NEURAL NETWORK STRUCTURES WITH L SYSTEMS AND GENETIC ALGORITHMS

Isto Aho, Harri Kemppainen, Kai Koskimies
Erkki Mäkinen and Tapio Niemi

University of Tampere
Department of Computer Science
P.O.Box 607
FIN-33101 Tampere, Finland

ISBN 951-44-4272-5
ISSN 0783-6910

Searching Neural Network Structures with L Systems and Genetic Algorithms

Isto Aho, Harri Kemppainen,
Kai Koskimies, Erkki Mäkinen and Tapio Niemi

{tyisah,koskimie,em,tapio}@cs.uta.fi
cshake@uta.fi

Department of Computer Science
University of Tampere
B.O. 607
FIN-33101 Tampere, Finland

Abstract

We present a new method for using genetic algorithms and L systems to grow up efficient neural network structures. Our L rules operate directly on 2-dimensional cell matrix. L rules are produced automatically by genetic algorithm and they have "age" that controls the number of firing times, i.e. times we can apply each rule. We have modified the conventional neural model so that it is easy to present the knowledge by birth (axon weights) and the learning by experience (dendrite weights). A connection is shown to exist between the axon weights and learning parameters used e.g. in back propagation. This system enables us to find special structures that are very fast for both to train and to operate comparing to conventional, layered methods.
Keywords genetic algorithms, Lindenmayer systems, back propagation, network structure, xor problem

1. Introduction

We start by describing the biological motivation for our work. For further biological background, see e.g. [9].

We first introduce a neuron (a neural cell) in the level of accuracy sufficient for our purposes. A neuron can be divided into three main parts: the cell body, the dendrites and the axon. The dendrites collect signals from other neurons, and transmit them to the cell body. The cell body receives the signals from its dendrites, and depending on the impulses received and on the internal structure of the cell body itself, the body does or does not activate the axon. The internal structure of the neuron is modeled by its activation function. The axon then transmits the possible impulse to the dendrites of other neurons. The interaction between neurons takes place on synaptic connections. This simplyfied model is used since the pioneering work of McCulloch and Pitts [13], and popularized in the computer science literature e.g. by Minsky [15].

The complex functions of a nervous system depend on the precise interconnections formed by thousands of cell types. The establisment of neuronal connections can be considered to occur in six major stages [9](Chapter 57). First, a uniform population of neural cells is induced. Second, these cells begin to diversify giving rise to immature neurons. Third, immature neurons migrate to their final positions. Fourth, neurons extend axons. Fifth, axons form synaptic connections. And sixth, some of the synaptic contacts that are formed initially are modified to generate the mature pattern of neural connections.

In our model, the stages 1–3 are realized by applying a two dimensional generalization of the well-known rewriting formalism, Lindenmayer systems (or L systems, for short). Our variant of L systems is introduced in Section 3. The current set of L rules is applied resulting a set of neurons of different types in a two dimensional matrix. Hence, the process of applying the L rules includes also the stages 2 and 3. Next, the axons are extended. In living organisms the axon growth is guided by various mechanisms including chemotropic molecules [9](Chapter 58). Our system imitates chemotropic molecules as desribed in subsection 4.2. The synaptic connections and synaptic contacts of our system are formed such that the resulting network is acyclic, because we restrict ourselves to back propagation networks.

The overall plan for building up the nervous system of a living organism is stored in the genes. The direct counterpart of genes in our system is the L rules applied when creating the network. Genes are affected by natural selection via crossovers and mutations. Similarly, our artificial genes, the L rules, are affected by the operations of genetic algorithms as described in subsection 2.3.

In addition to their inherent ability, living organisms can *learn* things. Learning can be considered as changes in the effectiveness of synaptic transmission. Again, this has a straightforward interpretation in our system: the weights of axon and dendrite connections can be changed accordingly.

As a whole, our system can be considered as a cycle where natural selection (genetic algorithm) and learning (back propagation) periods take turns, and hopefully, the resulting network becomes smaller and easier to teach.

Each individual has some knowledge by birth - consider a spider spinning its web. The ability to spin (or at least the ability to learn spinning easily) is coded into the genes. Individuals of higher species also learn by experience and even by thinking. The survival ability depends on the combination of these and is of crucial importance for the species.

We have applied these biological facts for designing artificial neural networks. The optimal structure of neural networks cannot be found easily. In normal cases we know the inputs and results of a problem but we do not know which kind of structure we should use. These biological facts enable us to find out efficient structures automatically and effectively. The structure itself is not important - its usefulness is the only thing we are interested in.

The human brain is modular: it is divided into two main blocks and these blocks are further divided into smaller blocks. The blocks are specialized to perform different tasks. This solution works (often) well.

Genes contain the information needed to construct this modular structure. The genetic information has to be coded in some meaningful way. In this paper we assume that Lindenmayer systems (L systems) can be used to code this information.

Genes order the modularity of the brain and the overall structure of the modules but they cannot set the precise states and placements of all the cells, because of the amount of the information needed to build exact brain (f. ex. twins are not fully similar). So the genetic information is coded somehow and we have assumed that the L rules correspond to this coding in general. The growth is performed by the application of the L rules.

The modular (structured) neural networks outperform the conventionally layered networks in real work [3]. We gain the modularity by using L rules which are obtained from genetic data. These L rules also initialize the cells that corresponds to knowledge given by birth. When the neural network is ready the network is taught. This process is "learning by experience."

While implementing our system we have learned that there already exist systems with about the same motivation as ours. Such systems are introduced at least by Boers and Kuiper [3], Gruau [5], Kitano [10], and Lucas [12]. However, our system seems to have features not implemented in the earlier systems. Especially, we follow the six stage development of a nervous system described above more closely than the other authors, e.g. the growth of axons is not treated separately in any existing system that we know. Moreover, we explicitely use learning through genes via axon weights.

2. Preliminaries

2.1. L systems

L system is a rewriting formalism originally introduced to model the biological growth of plants [11]. Alike normal (Chomsky type) formal grammars, L systems operate on strings of characters. However, since L systems model developing organisms, they rewrite strings in a parallel fashion.

We omit here the formal definition of L systems. An interested reader can consult e.g. [18]. Instead, we next give a very informal description of L systems.

Suppose we are given an alphabet Σ. An 0L system G (where '0' stands for '0-sided' or '0 context') consists of a string ω over Σ, the *axiom*, and a morphism h defined on Σ^*. The language $L(G)$ generated by G contains the strings obtained from ω by repeatedly applying h, i.e. $L(G) = \{h^i(\omega) \mid i \geq 0\}$, where $h^0(\omega) = \omega$. Depending on the properties of h we obtain several different subtypes of 0L systems which are intensively studied in the literature [18].

Obviously, 0-sided L systems are too restricted for our purposes, since we want to model organisms in which cells communicate and interact with each other. If we still consider L systems operating on strings, we can use 1-sided or 2-sided L systems [7, 11]. In these systems an interaction with neighbouring cells on one or both sides of a given cell is possible. The neighbouring cells in question define the context in which a change is possible, or in mathematical terms, when we can apply the morphism h. However, we want to use a more general platform for our L systems than strings. A straightforward generalization is to use matrices instead of strings [10]. This transformation implies some changes to our L system terminology.

Consider an $n \times n$ -matrix of characters. Here n can be any fixed natural number. In the tests reported in this paper a typical value for n has been 40. The size of the matrix depends on the number of parallel rewriting steps to be performed. In our tests, we have used 10-15 parallel steps in 40×40 matrices. Instead of a specific string, the axiom, we now start with some initial configuration of the matrix. Instead of morphism h we now speak about *rules*. The left hand side of a rule is a 3×3 matrix of characters. If the left hand side of a rule matches to a submatrix of the main matrix, called the cell matrix in the sequel, we replace the submatrix with the right hand side of the rule in question. Further details for our matrix based L systems are given in Section 3.

2.2. Formal neural networks

In this subsection we introduce some neural computing models. The main architectures and learning strategies are presented. The back propagation networks are handled in greater detail. For further details concerning these topics, the reader is referred e.g. to [6, 8].

First we look at a single neuron. In general a neuron i has n inputs, whose total weighted sum is called the *internal activity level* of i. The output of i is achieved through the use of activation function, which can be step, linear or sigmoid (either tanh or $i/(i + e^x)$) function. The different architectures are categorized by the

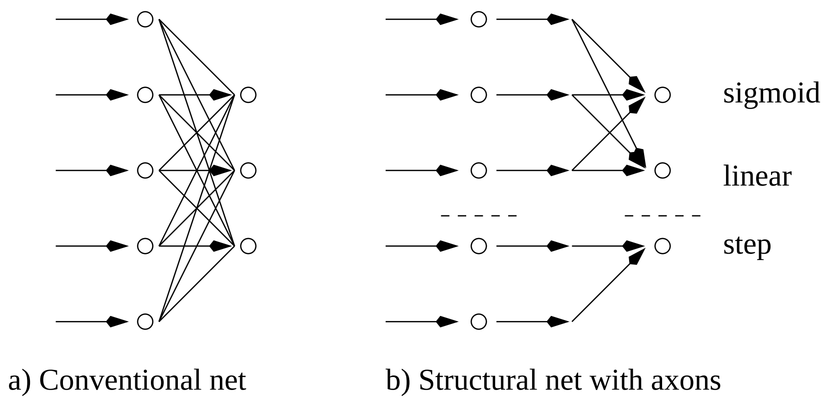

Figure 1: The calculation models.

relations of many individual neurons.

Layered feedforward networks have their neurons organized into the layers. Figure 1 (a) can be considered as a two-layer network, where the layers are fully connected, i.e. there is a connection from every neuron of layer one to every neuron of layer two. If there are only two layers, they have to be the input and output layers and the network is called a *single-layer feedforward network*. If there are more layers, the network is called a *multilayer feedforward network*. The layers other than input and output are called *hidden layers*. It has been shown that multilayer networks with one hidden layer can simulate any function to a wanted accuracy, if there is an approriate number of neurons in the hidden layer. (See [6].)

A layered feedforward network can be fully or partially connected. If it is partially connected, then some connections between the layers are missing.

A *recurrent network* have at least one feedback loop in its structure. The recurrent networks may or may not have hidden neurons. By using unit-delay elements in the recurrent networks, we obtain nonlinearly dynamically behaving networks. Recurrence increases the learning capacity of a network.

The learning strategies can be divided roughly into two parts: supervised and unsupervised learning. Supervised learning is a learning with a teacher which (who) gives examples to the network (student) while in unsupervised learning the network somehow determines itself what to do. Thus, self-organisation occurs in unsupervised learning. Next we shortly discuss a few main learning strategies (models).

In error-correction learning the network gives an answer that can be compared to the desired answer. According to the difference between answers, we can calculate corrections to the weights.

In Hebbian learning two simultaneously activated neurons strengthen the connection between neurons and asynchronously activated neurons weaken the connection. The synaptic efficiency in connection is also increased, if the mechanism in synapse is time-dependent, highly local and strongly interactive. Many neurons can be activated at the same time.

In competitive learning, however, only one neuron can be active at a time. Here, a set of neurons responds differently to a given set of examples. The neurons learn by competing for the right to respond for a subset of examples: they can be used as feature detectors.

Boltzmann learning uses stochastic learning algorithm. The neurons have two states, they are either on or off. The neurons together constitute a Boltzmann machine, for which we can form an energy function whose value depends on the states and on the synaptic weights between neurons. By flipping the states we change the value of the energy function and hopefully reach an equilibrium state.

In reinforcement learning we strengthen the tendency of system to produce satisfactory state of affairs and weaken the tendency to produce other states. In delayed reinforcement the system is built so that it can interact with its environment and learn to perform a task on the basis of the outcomes of its experience resulting from interaction with the environment. Reinforcement learning strategy has also been used in systems where the network learns to play games by playing against itself.

2.3. Genetic algorithms

The general principle underlying genetic algorithms is that of maintaining a population of possible solutions, called *chromosomes*. In our system a population is a set of L rules. The population undergoes an evolutionary process which imitates the natural biological evolution. In each generation better chromosomes have greater possibilities to reproduce, while worse chromosomes have greater possibilities to die and to be replaced by new individuals. To distinguish between "good" and "bad" chromosomes we need an evaluation function. In our system the evaluation function depends on the size of the network produced and on the ability of the network to learn quickly; our evaluation function is discussed in greater detail later on.

The general structure of a genetic algorithm is shown in Figure 2. There are several parameters to be fixed. First, we have to decide how to represent the set of possible solutions. In "pure" genetic algorithms only bit string representations were allowed, but we allow any representation that makes efficient computation possible. Hence, in the terminology of [14] our system is an "evolution program". Second, we have to choose an initial population. We use initial populations created by random selection. Third, we have to design the genetic operations which alter the composition of offspring during reproduction. The two basic genetic operations are the mutation operation and the crossover operation. Mutation is a unary operation which increases the variability of the population by making pointwise changes in the representation of the individuals. Crossover combines the features of two parents to form two new individuals by swapping corresponding segments of parents' representations. Our system uses the standard genetic operations of galib [4].

(1) $t := 0$;
(2) create the initial population P_0;
(3) evaluate the initial population;
(4) **while not** Termination-condition **do**
(5) $t := t + 1$;
(6) select individuals to be reproduced;
(7) apply genetic operations to create the new population P_t;
(8) evaluate(P_t);
(9) **od**

Figure 2: Genetic algorithm.

In what follows, we suppose that the reader is familiar with the basics of genetic algorithms and related topics as given e.g. in [14, 16].

3. L system directed growth of neural mass

As mentioned, we apply L rules in a matrix, called cell matrix, instead of a string. This is inspired by the fact that the growth of the cell mass is guided by cells close to each other. Each matrix entry may contain a character representing a cell of a certain type. The left hand side of a L rule is an 3×3 matrix (in a biological sense these are neighbouring cells). It is matched against 3×3 submatrices of the cell matrix. A successful match causes changes in the cell matrix; the changes depend on the right hand side of the L rule in question. The right hand side is a cell corresponding to the central cell as shown in Figure 3. Applying an L rule may change the cell type (the character in the matrix entry), create new cells, or delete old cells.

The central entry in a 3×3 matrix is in a special role. A successful match of a rule requires that the central node must match. On the other hand, the other eight entries may contain different kind of wildcard characters not requiring exact match. The system starts with one "seed cell". Other possibility is to use 3×3 matrix also on the right hand side of the L rule instead of one cell so that one L rule can change several cells together at one time. However, this is not implemented in our system.

There are three types of wildcards and four types of matching. If the wildcard is not used the match is exact, i.e. all the values of the cell must be the same. The target cell has to be exactly matched. The wildcard NoCell means that there must

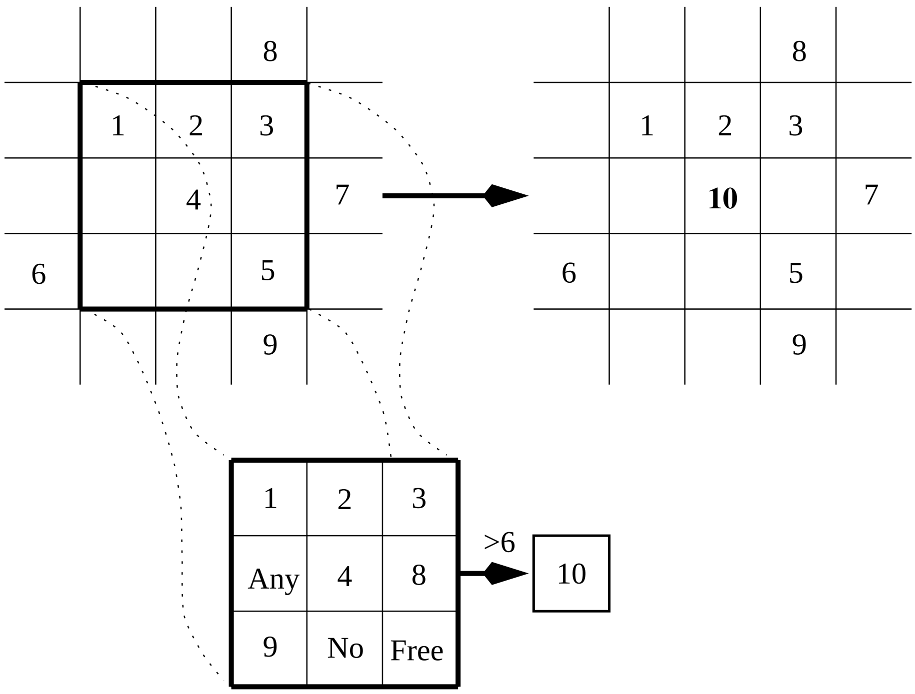

Figure 3: Example of L rule and its application.

not be a cell in the cell matrix. The wildcard FreeCell means that there has to be a cell in the cell matrix but this cell can be of any kind. Last wildcard Anything means that we do not even check if there is the cell in the cell matrix, anything goes. The target cell has to match and eight surrounding cells have to also match according to any wildcards used. Each of the eight cells may have its own wildcard.

A cell has four attributes: a weight, threshold function, scent and target scent. The weight is initial value for the axon of the cell. The threshold function gives the function to be used for the connections of the cell. A cell looks for other cells having the same scent or scent close enough to the target scent of the cell. Scent values are used to build connections between cells. Cells are spatially distributed in a two-dimensional matrix. Hence, there has to be a way to make decisions concerning how connections are created between cells. The scents and the distance between cells are used as criteria for creating connections.

The context consists of nine cells in the 3×3 matrix. The central cell is matched with the target cell of the cell matrix. All the other cells in the context are also matched but not strictly. Wildcards can be used to weaken exact matching.

There are four different types of L rules: And rule, Or rule, GreaterThan rule and LesserThan rule. And rule means that in every single matching the corresponding cell in the context have to match. Or rule means that just one match is enough. In GreaterThan rule there has to be as many matches as accompanied value says and in LesserThan rule there have to be as many or less matches. Figure 3 shows how to apply L rule of type GreaterThan (> 6). Seven cells matchs and two do not, hence we can replace cell number four with cell number ten. Figure 3 does not show the exact contents or attributes of each cell.

Further, our L rules have attribute "age" which bounds the number of times the rule can be fired, i.e. matched. One parallel step can consume only one unit of age: if the same rule fires many times on the same parallel step, it does not consume the age any more than firing the rule only once at that parallel step. The use of age is justified by the fact that the cell mass does not continue to grow forever. Further, it is very hard to justify any human selected number of L rule application times: we leave this decision to genetic algorithm.

The sample rule shown in Figure 3 changes the type of the cell corresponding to the central entry. Notice that rules of this form can also create new entries to the matrix, i.e. they can increase the cell mass.

4. Generation of cell connections

4.1. Axons and Dendrites

After we have grown the neurons we have to create connections between them. Every neuron grows one axon first, after that it grows one or more dendrites. When a dendrite meets an axon, they establish a connection. A neuron grows its axon to its right hand side and its dendrites to the left. This orientation confirms that our network will always be acyclic.

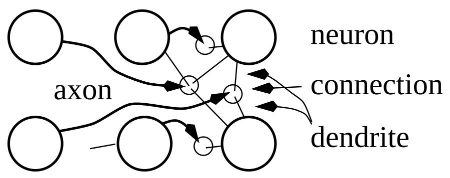

Figure 4: Neurons and connections.

Figure 4 shows an example for our neural network. There are six neurons, which have grown their axons and dendrites. The neurons on the left have no dendrites and the ones on the right have no axons for obvious reasons. The lower of the two neurons in the middle does not have any incoming dendrite connections so we can remove it from the final network or it can become input-cell if we need more than two inputs.

4.2. Growing Axons

A neuron has two codes, which it uses for growing axons. We call them scent numbers. The one is the own scent of the neuron and the other one is the target scent. A neuron grows the axon to such a neuron whose scent is near the scent of the growing neuron. If the difference is small enough or the axon long enough, the axon stops at that position. The growth of the axon is independent of the other axons, so it is not important in what order the neurons grow their axons. The growing can even happen parallel.

4.3. Growing Dendrites

When every neuron has grown its axon, we can begin to grow the dendrites. The principle is quite similar to growing axons, but this time every neuron can have more than one dendrite and the scent numbers are not used. A neuron starts to grow a dendrite to the left side of its position. The dendrite tries to find an axon and when it finds such, it makes a connection. The search is begun in small area on the left side of the neuron. If an axon is not found in this area, the area will be expanded.

A neuron can grow more than one dendrite and these dendrites can, of course, connect to different axons, but it is not normally accepted that more than one dendrite of the same neuron can connect to the same axon. The dendrites from different neurons can connect to the same axon. It is possible to bound the number of dendrites connecting to the same axon, how many dendrites one neuron can grow or how long a dendrite can be. By using these parameters we can affect the resulting network. If we want a tight and large network with many connections, we use greater values for these parameters. If we use small values, we get a smaller network with few connections.

4.4. Setting Input and Output Cells

The main idea is that the left side of the cell network becomes the input cells and the right side of the network becomes the output cells. We noticed that it is not the best way to take first cells of the border, because they might be isolated from the others. So we take the first ones, which have a connection to some other cell. In this way we get more probable networks, which can operate at least in some way. The amount of the input and output cells can be computed, since we know the input and output data.

After the input and output cells have been chosen we suppress the cell matrix. There might be some cells that have no inputs or outputs. They can be removed. See also Figure 5. The dashed cells and connections are removed because they are not used in our calculations. We also shorten the chains of cells. In Figure 5 the circulated area containing three connections and two cells is replaced with two connections and one cell. This has its influence on the behaviour of the network. However, because this is done for all the networks, the speed gain is more important than using the exact networks the L rules give.

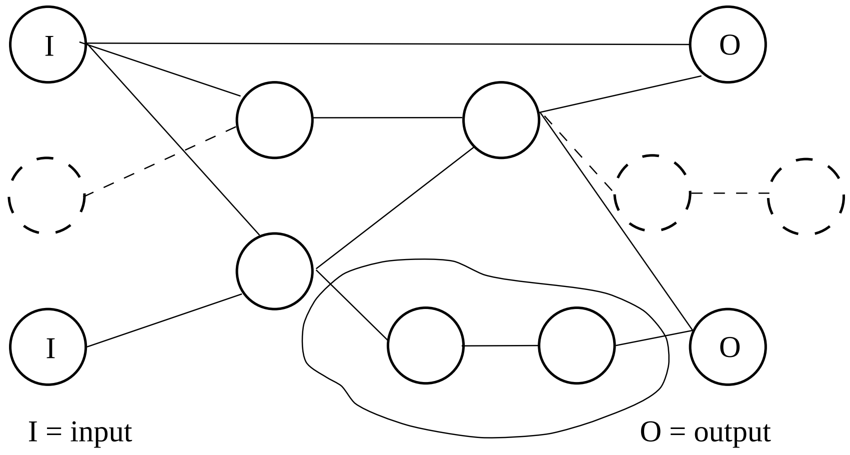

Figure 5: Suppressing network.

5. Learning and evolution

In this section we show how to include the axon information into the neural network models. In section 5.2 a link is established between the *axon weights* and the learning parameters η.

5.1. Axons and structure

The neural models are usually based on the simplification of the biological neural cell [6, 8]. Figure 6 shows a neuron with synapses and input dendrites, the cell body, and axon with synaptic terminals.

Modular neural networks have many properties that are of interest to us [2, 3, 6, 8, 10]. Segev [19] introduces different models for a single neurons. Based on Segev's result we notice that axons can be used to form modularity so that our model resembles the operation of real brain more than the conventional model. Figure 1 shows both models so that the differences become clear. The axons have their own weights and the second layer is divided into separate subsets of neurons. This decreases the number of different connections. Neurons can also specialize; in our algorithm this is modelled so that the cells can have different activation functions.

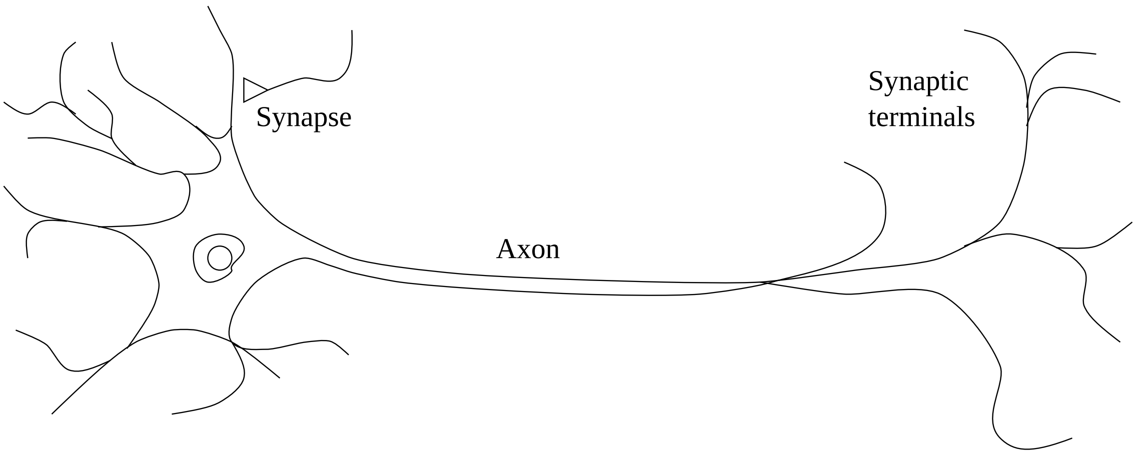

Figure 6: Typical neuron.

The dotted lines at the layers separate "levels" that are fully connected. It is worth mentioning that in the structured network there are usually much fewer connections than in the conventional networks.

If the networks use back propagation and if the structure is applicable to problem at the hand, the learning of the network will become much faster than and superior to that in the conventional networks [3]. But why to use separate weights for the axons? If the dendrite connections also have their own weights, then the weights from layer one to layer two can have any value. The explanation for this is given later in subsection 5.2 by establishing a link between the learning rate parameter(s) η and the axon weights.

5.2. The learning algorithm

The learning algorithm for our model is based directly on the normal back propagation algorithm (cf. [6, 8]). However, a few things may be pointed out.

The *levels* are sets of nodes, so that the levels can be collected to two-pairs $G_1, \dots, G_n$, the input and output levels that are fully connected. The first pairs contain the input nodes of the whole network and the last pairs the output nodes of the whole network. We do not implicitly form the levels in the algorithms, but it is convenient to use them in the forward pass.

Consider a network $G_i = (I_i, O_i)$ where $|I_i| = n$ and $|O_i| = m$. Hence, we have axon weights $|A| = n$, activation functions $|f| = m$, connection matrix (dendrite weights) $N \in \mathbb{R}^{m \times n}$ and the signals $|s| = n$ waiting for the processing in the input nodes. We have dropped the index i, so that we can use, say A_i, to mean the axon weight of node i. The sum $v_i = \sum_j N_{ij} s_j$ is the internal activity level of the node i. The inputs are added directly to the internal activity level so the output O will give out

$$f(Ns) \cdot A = f(v) \cdot A = f_i(v_i)A_i \quad (1 \leq i \leq m), \tag{1}$$

where the $(\cdot)$ operation means Hadamard product. The output of the node i to the dendrite connection is $f_i(v_i)A_i$. Once we have calculated the output O_j, we can form the inputs for the next level G_{j+1} and then the outputs of this level and so on until we have calculated the final outputs.

Let the final output (or result) be y and desired result d. Now we have error $e = d - y$ and we calculate the gradients against $E = \sum e_i^2/2$. In backward phase we cannot use the levels as in forward phase, because in the inner nodes the error is calculated from output nodes that might not be on the same level. However, we can calculate and store $v_i = \sum_j N_{ij} A_j s_j$ for later use. The correction for the connections leading to output nodes i of the whole network is

$$\Delta N_{ij} = \frac{\partial E}{\partial N_{ij}} = \delta_i A_j s_j, \tag{2}$$

where the $\delta_i = e_i(-1)f_i'(v_i)$. The correction for the links leading to inner nodes i for the whole network looks as (2), but this time we have $\delta_i = f_i'(v_i) \sum_k \delta_k N_{ki} A_i$. Here the δ-sum corresponds to the error e_i at the output nodes.

From the equation (2) it follows that we use almost normal back propagation, if we think that the axon weights A_i are learning rate parameters η_i. Haykin [6] has noted that the performance of the network can be improved by giving individual parameters η_i for the layers or for the cells. Our system justifies and determines the approriate parameters for the user through evolution, and the parameters (axons) affect also the gradients more naturally than the normal algorithm does.

We do not have to choose those parameters ourselves but we leave readily the decision to the genetic algorithm and L rules. According to Haykin [6] we can calculate and use the gradients also for learning parameters (each node has its own parameter), and the network will perform better and learn faster. The genetic algorithm should find good axon weights for the network, but it has to learn itself how the genes affect the axon weights through L rules. This may take a lot of time.

5.3. Evolution

To determine the appropriate structure, we use evolution through L systems, since there is no direct method to calculate the goodness of a given structure. We have used standard genetic operations and algorithms provided by galib [4].

The main algorithm is shown in Figure 7. After some initializations we let the evolution perform a predetermined number of generations for the population. In

```
Initialize random genes for the population
while generations left do
    Make some evolutionary operations for the population
    for each gene in population do
        Form the L rules from the gene and create neural mass
        Generate the cell connections
        Train the new network
        Calculate the score based on the training results
    od
od
```

Figure 7: The main algorithm.

each generation some genetic operations are done for the population after which for each individual in the population, we calculate the score. To obtain the score, we form the L rules, generate the connections and train the network corresponding to the individual at hand in the population. This setting allows an easy way to distribute calculations to many processors.

The score is a combination of the learning speed of the network (epochs used), the change of the error (end error minus start error), the overall error at the end of the learning process and the number of neurons used in the network (i.e. the size of the network). These criteria can be weighted to make them work well together. We will also add and test the number of connections with the criteria mentioned above.

A individual contains only the information to form L rules. The number of generations needed is highly problem specific, just like the weights for the criteria.

6. Implementation

The genetic algorithms are well suited for distributed parallel implementations and in this framework they will perform specially well. We use pvm-library [17] to distribute the calculations of each gene in the population. The pvm-distribution enables the network of heterogeneous workstations to look as a multiprocessor computer.

The network implies rather large latency but the forming of L rules, the growth of the network, the generation of connections and the learning of one individual will take much more time than the traffic caused by sending one gene (at most a few kilos of integers) over the network. A few seconds or fractions of a second for the network traffic is little as only the learning phase in the calculation of the score may take hours depending on the problem at hand.

Hence, we use master-slave model where the master performs all the genetic

```
Initialize random genes for the population
while generations left do
    Make some evolutionary operations for the population
    while not done do
        if some gene without score and free slave then
            Send gene to the slave
        end
        if slave with gene and the slave is ready to send score then
            Receive the score from the slave
        end
        if all genes have their score then
            done is set true
        end
    od
od
```

Figure 8: The master.

operations and counts the number of generations and where the slaves form the L rules, grow up and generate the network, learn the network and finally evaluate the score to be returned to the master. Figure 8 shows the master algorithm.

The slave is essentially the innermost part of the for-loop of the algorithm in Figure 7 so we do not show it here. Apart from that, it will have the tasks of receiving the gene, sending the pvm-note "score ready" and the actual score sending.

With the score we actually send also the error, the number of cells used in the learning phase and the matrices (connections, axon weigths, i/o-types and activation function types). We want the master to keep records on the overall progress. It records from every population the average scores, errors and sizes of the valid networks. The valid networks are those that have at least 2 cells (networks containing only one cell cannot be used). The galib can also give a lot of different statistics concerning genes and population.

Also each new best scoring network is recorded into a separate file. The first networks in that file are not very good ones but they show how the system evolve through the time. The last networks might have rather similar scores but the networks can be totally different. Having few networks from which to select gives alternatives we are looking for (cf. section 7).

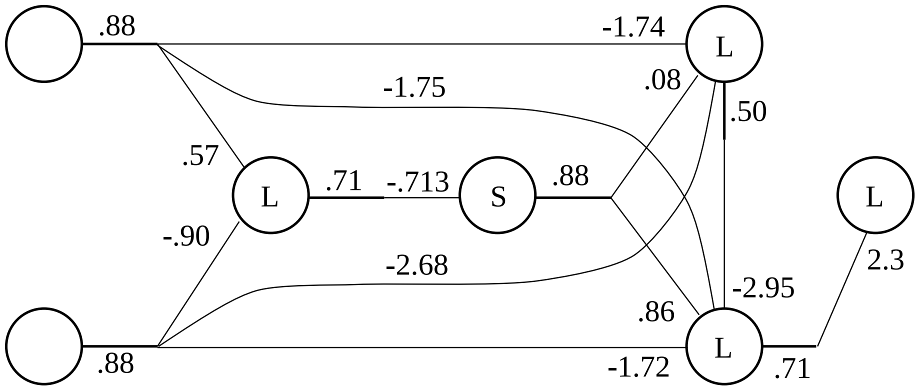

Figure 9: Seven cells solving the xor problem.

7. Experiments

To obtain good results one has to find first appropriate parameters for the system (see appendix for detailed description of parameters). It is achieved by testing the system and looking at the outputs of the system. It may take a few days but after that the system starts to work. Much of the time is consumed by waiting to see at least few generations and their results. When the system is somehow tuned, it can be used with some problems. We used the well-known xor problem.

Figures 9–10 show some structures found for the xor problem. On the left there are input nodes and at right there is the output node. Each network was taught with 18 example-answer pairs until the error was less than one percent or the network had used all of its epochs (one epoch consists of those 18 pairs). The results are shown in the figures, where we denote S for sigmoid, T for tanh, s for step and L for linear activation function. Also rounded axon and dendrite weights are shown. The axons are drawn with a thicker line. We also tested the known structure for xor problem given in [6, 8], which needed, on many trials, several thousands epochs and which even did not converge every time. The long training times for xor problem with standard solution have been noticed also in f. ex. [8]. Hence our system has found very interesting networks and has proved to work out (almost) as expected. The main problem is the running time: we needed time from one to two weeks with 5 Sun Workstations and two Linux PC.

Figure 9 shows a network consisting of 7 cells. This particular example needed 80 epochs to converge. This networks contains only sigmoid and linear activation functions. It is noteworthy that sigmoid function is needed only once. In the same test the system found other networks with a similar graph: the axons were different and usually the activation functions were a combination of sigmoid and tanh functions without step or linear functions. The convergences happened in about 130–750

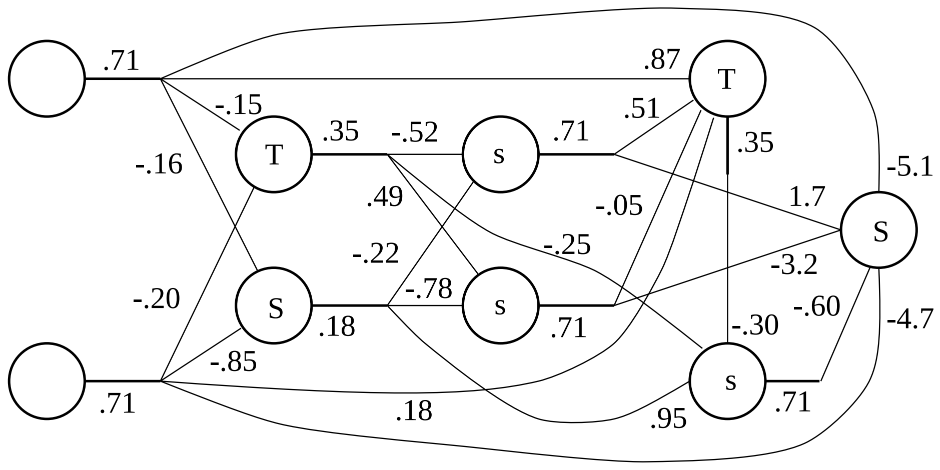

Figure 10: Nine cells solving the xor problem.

epochs with average about 425 epochs.

The algorithm also found many 8 cell networks. They showed up similarly as the 7 cell networks: they had the same underlying graph and only the axons and the activation functions differed. However, the axons and the activation functions did not differ very much. These networks were reasonably fast (convergences happened in $200 - 800$ epochs).

Figure 10 shows a network consisting of 9 cells. It needed only 138 epochs before convergence (in this particular case, not overall). One should note that three cells has step function as their activation function.

We also found networks of sizes from 15 (average 300 epochs) up 22 (one with 51 epochs) that were able to learn efficiently the xor problem. These networks were found during normal system test runs. The networks structures found and believed to be efficient should be tested separately by several successive runs to have a view of the overall performance.

At the same time we noted the need to make the evolution score dependent on the total number of multiply-operations used in the learning phase. That may describe the speed of learning most effectively. Networks involve rather often only tanh and sigmoid activation function (at the same time), which is not surprising. In some cases also step-functions has shown but linear activation functions did not show up very often.

We remarked during tests that axon weights are not adjusted as strongly as we supposed. To increase this “prelearning” effect we propose that the dendrite weigths are not initialized randomly. We rather run the system a number of generations with deterministically initialized dendrite weight $3 - 10$ epochs to find suitable axon weights first and then continue normally allowing more epochs (for example maximum 3500 epochs).

8. Conclusion

We have considered a new method to find efficient neural network structures. Our method simulates the evolution and thereby uses genetic algorithms. The genes involved in the individuals are encoded into the L rules. It is possible that we have stored too much information in our individuals and we are planning to simplify them.

After the L rules are formed, we start to operate on 2-dimensional cell matrix with L rules. Each L rule is matched in turn to whole matrix and if at least one matching occurs the age of the rule is increased. When the rule is old enough to die, we stop to use that L rule. The age and other parameters of L rules are produced by the genetic algorithm.

The cell matrix is formed with L rules or more exactly, with L rule directed growth. After producing enough cell mass (i.e. no more applicable L rules can be found), we grow the axons from the cells and then form some dendrite connections. Now the network topology is in principle ready to be taught. Practically, however, we want to suppress the network. We remove useless cells and connections.

In our neural model the signal goes from the neuron first through the axon, after which it goes through dendrites to the other neurons. The neuron having signals from many sources first sums them up and then applies activation function, which also can be one of several types (given by the L rules).

The axons represent the prelearned things: information that is knowledge by birth. The dendrite connections are used to represent experience. A link is shown between learning parameters used in back propagation and the axon weights.

Our experiments show that this kind of system can be used to find very efficient structures for neural networks. Our prototype system is still rather slow and it consumes much computational capacity. However, other similar systems suffer also from these kinds of problems. These problems are mostly technical and we are working to improve the efficiency of our system.

We have used genetic algorithms, but maybe evolutionary algorithms (EA) [1] would be more useful in this kind of problem. It is very difficult to set all the parameters for genetic algorithm and learning process. If we use EA, we can leave many decisions to the algorithm. On the other hand our problem will easily become too large to solve if we leave all parameters to the EA.

Acknowledgment

Authors are grateful to Prof. Martti Juhola for his comments.

References

[1] Thomas Bäck, Evolutionary Algorithms: Principles and Algorithms, *SNAC—School on Natural Computation, Working Material*, Turku Centre for Computer Science, TUCS General Publication, No. 6, August 1997.

[2] Richard K. Belew, Interposing an ontogenic model between genetic algorithms and neural networks, In: Stephen J. Hanson, Jack D. Cowan and C. Lee Giles (eds.), *Advances in Neural Information Processing Systems* 5, Morgan Kaufmann, San Mateo, CA, 1993, 99-106.

[3] Egbert J.W. Boers and Herman Kuiper, *Biological metaphors and the design of modular artificial neural networks*, Master's thesis, Dep. CS and Exp. and theor. psych., Leiden University, Netherlands.

[4] galib, software available in http://lancet.mit.edu/ga/.

[5] Frederic Gruau, Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm. Ph.D. Thesis, Ecole Normale Supérieure de Lyon, France, 1994.

[6] Simon Haykin, *Neural Networks, A Comprehensive Foundation*, Macmillan College Publishing Company, 1994.

[7] Gabor T. Herman and Grzegorz Rozenberg, *Developmental Systems and Languages*, North-Holland, 1975.

[8] John Hertz, Anders Krogh and Richard G. Palmer, *Introduction to the theory of neural computation*, Lecture Notes Volume I, Santa Fe Institute Studies in the Sciences of Complexity, Addison-Wesley, March 1992.

[9] Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell (eds.), *Principles of Neural Science*, Third Edition, Elsevier, 1991.

[10] Hiroaki Kitano, Designing neural networks using genetic algorithms with graph generation system, *Complex Systems* 4, 1990, 461–476.

[11] Aristid Lindenmayer, Mathematical models for cellular interactions in development, Parts I and II, *Journal of Theoretical Biology* 18, 1968, 280-315.

[12] S.M. Lucas, Evolving neural based learning behaviours with set-based chromosomes, In: *Proc. of European Symposium on Artificial Neural Networks (ESANN '96)*, 1996, 291 – 296.

[13] Warren S. McCulloch and Walter Pitts, A logical calculus of the ideas immanent in nervous activity, *Bulletin of Mathematical Biophysics*, 5, 1943, 115-133.

[14] Zbigniew Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, 1992.

[15] Marvin Minsky, *Computation: Finite and Infinite Machines*, Prentice Hall, 1967.

[16] Melanie Mitchell, *An Introduction to Genetic Algorithms*, The MIT Press, 1996.

[17] pvm, software available in http://www.epm.ornl.gov/pvm/pvm_home.html

[18] Grzegorz Rozenberg and Arto Salomaa, *The Mathematical Theory of L Systems*, Academic Press, 1980.

[19] Idan Segev, Single neurone models: oversimple, complex and reduced, *TINS* 15, 11, 1992, 414–421.

Appendix - system parameters

The whole system has many parameters which affect its behaviour dramatically. The parameters are divided into a groups. One group contains the parameters that are fixed into the code as constants and to change them one have to compile the code. The other three groups are read from the files as the system starts. These files are conf_gen.dat, conf_fix.dat and conf_run.dat.

The first set is the parameter file controlling the genetic algorithm. The galib has many things that can be parametrized. We use the next ones:

```
GEN_POP_SIZE        100
GEN_GENERATIONS     1000
GEN_SCORE_FILE      bog.dat
GEN_FLUSH_FREC      3
GEN_CROSSOVERS      0.9
GEN_MUTATIONS       0.015
GEN_FILE_CHANGE     250
```

The meaning of GEN_POP_SIZE, GEN_GENERATIONS, GEN_CROSSOVERS and GEN_MUTATIONS is obvious. The GEN_FLUSH_FREC is the frequency in generations how often to write the some statistics into the file GEN_SCORE_FILE. Parameter GEN_FILE_CHANGE is not used with master-slave -model. It is used if the calculations are done on single machine without pvm and the number is the generation number when to change the conf_run.dat to some other file with the same parameters (with different values). This way we can change the behaviour of the system at run time, if it seems to be worthwhile.

The second set of parameters are fixed parameters, i.e. they are not meant to be changeable during the run. The parameters are:

```
NUMBER_OF_INPUTS    16
NUMBER_OF_OUTPUTS   3
NN_EX               /home/ohdake/tyisah/gennet/data2/esim.mat
NN_AN               /home/ohdake/tyisah/gennet/data2/ans.mat
STATS_FILE          /home/ohdake/tyisah/gennet/data2/stats.dat
COLLECT_STATS       1
BEST_NET_FILE       /home/ohdake/tyisah/gennet/data2/best_nets.dat
```

By NUMBER_OF_INPUTS and NUMBER_OF_OUTPUTS we fix the size of the network. The number of inputs means the maximum number of input nodes the network can have. It is also the number of columns the example file contains (dimension of one example). Similarly, the number of outputs is the number of outputs the network must have and at the same time the number of columns the answer file contains (dimension of one answer to an example). The rows of the example and the answer files correspond to each other.

That we want every time the number of outputs to be precisely used has nothing astonishing. However, the number of inputs does not have to be fixed in the network itself. There are situations when the network can learn the problem without using all its inputs [3]. Also if the network needs more inputs than the example file has, the additional inputs can be interpreted as treshold inputs giving always -1 signal.

The `COLLECT_STATS` parameter tells if the statistics are to be collected. If it is one, then the statistics are collected to `STATS_FILE`. The `NEST_NET_FILE` is always in use.

Another group of parameters are varying (run) parameters. At the moment the system reads this file only once. Yet it is easy to change the system so that it reads the parameter file every time it processes a gene and this would enable us to change the values at run time by writing directly new values to the file. These parameters are

```
NN_USE_INIT_VALS    1
NN_INIT_VALS        0.5
NN_MAX_EPOCHS       2000
NN_ALFA             0.5
NN_ERROR            0.01
NN_ME_ERROR         0.0001
NN_ME_ERROR_T       25
NN_DEATH            750
NN_SHOW_PROG        0
NN_LARGE_SIZE       45
NN_LARGE_EPOCHS     2
SC_SIZE_W           0.2
SC_ERROR_W          2500
SC_IDEAL_SIZE       15
SC_LITTLE_NET       95
```

The parameters with prefix `NN` are used with back propagation algorithm and the parameters with prefix `SC` are used in the function that calculates the actual score to be given for the genetic algorithm.

The first two parameters are related to the use of initial values. If we want to use initial values before starting to learn the network, the values are given randomly from the interval $[-0.5, 0.5]$. `NN_MAX_EPOCHS` counts the number of times we teach the example-answer pairs (whole file) to the neural network. This varies a lot with different problems. NN_ERROR is the error level of the mean square error of the epoch that we consider as small enough.

Parameters NN_ME_ERROR and ME_ERROR_T are used to stop the learning phase of the networks that have converged to some value. If the difference between two consecutive epochs is smaller NN_ME_ERROR we decrease ME_ERROR_T by one. When it is zero, we stop the learning phase. In the tests we realized rather quickly that many networks are such that there is no need to continue the learning phase to NN_MAX_EPOCHS. Parameter NN_DEATH is also used to quit the learning phase before NN_MAX_EPOCHS: it is decreased by one every time the error of the previous epoch is smaller than the new one. When it reaches zero we stop the learning phase.

NN_SHOW_PROG should be zero (false) when used with pvm-system. If it is one (true), then the learning algorithm shows the progress of error of the networks, not every epochs, but some. NN_LARGE_SIZE is the size of the networks in cells that we consider too large to be practical and these networks are learned only NN_LARGE_EPOCHS epochs. "Too large" is very dependent on the problem at hand and it should be considered carefully. Too large networks are slow to teach and they are also rather often non-stable.

The SC_LITTLE_NET is the number to be added to the networks that have less cells than the number of the input and output nodes is. It can be said that those networks are not good ones, because it is unlike that there will be intermediate nodes and hence the question of structure is in almost every case ruled out. If the network size is less than SC_IDEAL_SIZE we add same amount to the size of the network.

Size is used in score so that its square is multiplied with the number of the epochs used (if the network did not learn the problem then the NN_MAX_EPOCHS is used) and then it is weighted with SC_SIZE_W parameter and decreased from the score. SC_ERROR_W weights the inverse of the error and this is added to the score.

Size is squared because we want networks that learn and operate fast. The total number of multiply-operations is of order of square of the size multiplied with the number of the epochs used in the operation (feed forward) phase.