Evolution of neural net architectures by a hierarchical grammar-based genetic system

Christian Jacob and Jan Rehder
Lehrstuhl für Programmiersprachen
Universität Erlangen-Nürnberg
Martensstraße 3, W-8520 Erlangen, Germany
Email: *jacob@informatik.uni-erlangen.de*

Abstract

We present a hierarchically structured system for the evolution of connectionist systems. Our approach is exemplified by evolution paradigms for neural network topologies and weights. Our descriptions of a network's connectivity are based on context-free grammars which are used to characterize signal flow from input to output neurons. Evolution of a simple control task gives a first impression about the capabilities of this approach.

1 Introduction

The design process for problem dependent neural network models usually consists of the following four stages: selection of a problem domain, selection or development of a suitable network architecture, choosing a learning algorithm for adjusting network specific parameters due to the problem domain, and testing of the network performance according to objective performance measures.

Although this sounds like a recipe for straight forward neural network development, the problem of designing application specific neural networks remains difficult due to the fact that the diverse phases of network design are interdependent and a great deal of experience is needed to choose suitable parameter values (Which architecture? Which connectivity? Which learning rule? ...).

Let us have a brief look at the problem of finding a suitable network architecture for a specific problem domain. This task often depends on the researcher's skill and experience to choose the proper architectural constraints for the net (More than one layer? Feed-forward connections only? Recurrent network?). In many cases, however, suitable net topologies only evolve from the process of supervised or unsupervised learning. With feed-forward networks trained by the error-backpropagation method the number of hidden layers and units may vary according to some performance criterions; with self-organizing networks not only the number of units needed is unknown, but also their connectivity structure. As Miller, Todd and Hegde [?] express it, "the network design stage remains something of a black art". The same is true for the selection of proper neuron activation functionality as well as learning rules for adjusting network weights and connectivity.

However, the "black art of network design" is not as "black" as it seems. Often the network designer has a rough idea about which neural network models could be tried to solve, say, classification, motion control or feature mapping tasks. In most cases there are some "rules of thumb" for special parameter values like the number of hidden units, the learning rate, the neuron activation function etc. So why not use this experience to enhance evolutional development of problem specific neural network models?

Nature solves part of the design problems of natural nervous systems through evolution mechanisms. To put it very simply, the network development process is naturally done in two stages: First, a coarse connectivity structure within specialized networks is evolved; the information about this structuring process is contained in genetic strings and has evolved through genetic mechanisms. Second, fine tuning of the network is done through neurological mechanisms controlled by environmental input signals, which cause the network to "learn" and adjust its performance to specialized tasks. Natural genetic coding, however, is much more complex and hierarchically organized than currently used bit-string codings applied in most of the genetic algorithm systems.

With these ideas in mind we want to develop an evolutional system which supports the different neural network design phases as described above. With "evolutional system" we mean a hybrid system capable of using genetic or evolutional as well as neural mechanisms for optimization, adaptation and learn-

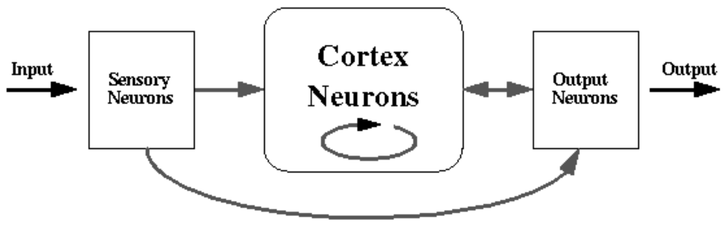


Figure 1: Neural net model

ing.

First of all we are proposing a coarse structure for an evolutional system for problemspecific neural network design. Secondly, we give a short description of our grammar-based approach for evolving neural net connectivity. The outlined system is currently being implemented and will serve as a basis for a hybrid system to get a better understanding when to switch from genetic search and adaptation methods to neurally inspired parameter adaptation and learning as well as to examine the use of higher-order codings and evolutional operators for a broad range of problem domains.

2 Related work

Several articles have been published concerning neural network design with the help of genetic algorithms. Most of the articles focus on genetic optimization techniques of neural net connectivity for the specialized class of feed-forward networks trained by error backpropagation algorithms [?], [?], [?]. In [?] and [?] two neural network development systems are described in detail, where genetic operators and search are used to evolve architectures for feed forward networks which are then trained through (modified versions of) backpropagation algorithms and evaluated by testing the resulting networks on predefined test data. In [?] neural learning is replaced by a genetic algorithm, leaving the fine-tuning or optimization of weight values to the genetic operators. Koza and Rice [?] evolve net connectivity and weights simultaneously by using a LISP S-expression coding. Bornholdt and Graudenz [?] let their GA-system operate on linked list data structures coding the net structure as well as the weights.

3 Hierarchical neural network design: A brief overview

We use an approach similar to [?] and [?] to evolve neural net connectivity, single neuron functionality and connection values. The network model used (see figure 1) consists of a predefined set of input and output neurons and a set of cortex neurons to be evolved. The input neurons are connected in feedforward direction only, i.e. inputs can be passed to either hidden neurons or output neurons directly. The cortex neurons are connected either to hidden or output neurons.

The functionality of the hidden neurons is depicted in figure 2: A summation function collects the neuron's incoming signals; this function might be a weighted summation of the inputs o_i, a Sigma-Pi-function or another appropriate input processing function. The summation value is processed by an activation function (linear, sigmoid, radial basis etc.) resulting in an internal activity of the neuron, which is taken by the output function (identity, linear threshold etc.) to compute an externally visible output value o_p that can be passed to other neurons.

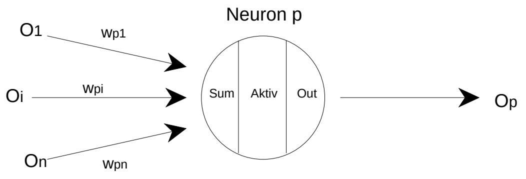


Figure 2: Model of a cortex neuron

The design process for a neural network then has to evolve a connectivity structure for the input, cortex and output neurons, a set of functional parameters defining the functionality of the hidden neurons, and a set of weight values for all connections. For each of these three, partly interdependent phases we

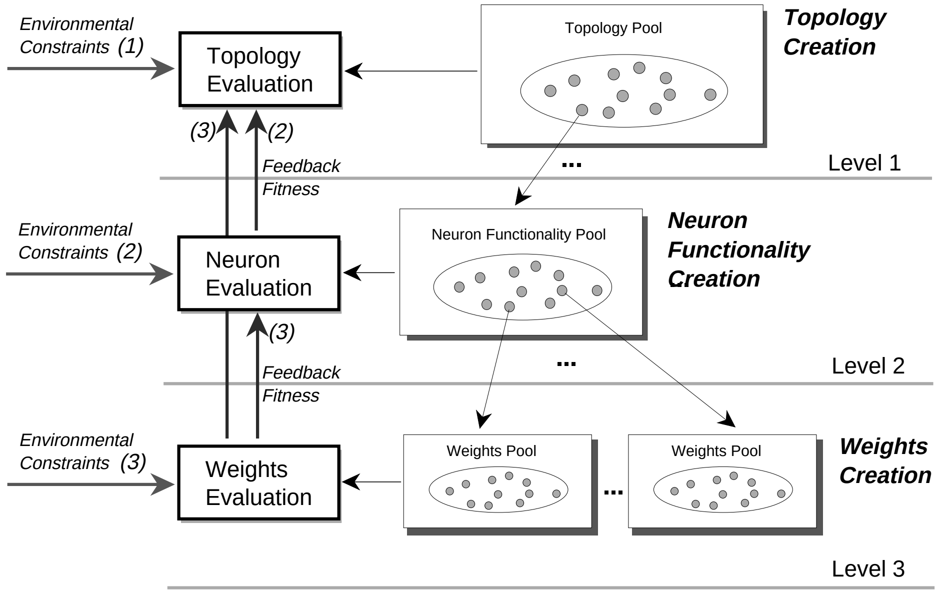


Figure 3: Design Hierarchy

use different string populations and codings. So the string length is usually rather short, and the population sizes can be kept small without giving up too much of a population's diversity. Thus, search spaces for the genetic algorithms are reduced. The results of the structured evoluation and evalutation processes can be interpreted more easily by a human supervisor.

The evolution process for a neural network could be described as in figure 3. At each level there is a creation module which evolves a pool of competing chromosomes (= strings of coded parameters) and an evaluation module testing chromosome fitness via problemspecific constraints. For each level of pools there are two fitness values to be taken into account for each chromosome: environmental constraints define a "coarse" fitness value, whereas a kind of fitness fine tuning can be achieved by using "feedback" fitness values from levels below. At each level specialized parameter representations ("chromosomes") are used to generate new chromosome populations from previous ones. Strings and respective feedback fitness values are the only interface between the different evolution levels. Figure 4 outlines the coarse structure of an evolution module serving as a building block for the hierarchical evolution system.

A brief example might explain these ideas. Suppose we want to evolve feedforward networks for pattern classification. So the constraints for net topology (level 1) will accept only chromosomes describing feedforward connectivity, and will restrict the number of neurons, the maximal path length from input to output neurons or the connection density. Neuron functionality (level 2) could be constrained to activation functions that settle to zero for large absolute values. As a last constraint the weights (level 3) might be restricted to real values from the interval between zero and one.

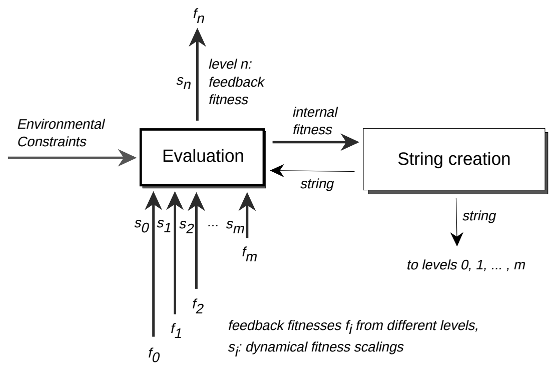

f_n
s_n
level n: feedback fitness
Environmental Constraints
Evaluation
internal fitness
String creation
string
string
to levels 0, 1, ... , m
s_0 s_1 s_2 ... s_m
f_m
f_2
f_1
f_0
feedback fitnesses f_i from different levels,
s_i: dynamical fitness scalings

Figure 4: Structure of a string evolution module

With this structured network design it is easy to alter the sequence of modules and the number of modules used to evolve and optimize problemspecific net architectures. Furthermore, genetic modules may be replaced by, e.g., neurally inspired algorithms: Topology evolution might be done by a self-organizing process controlled by a set of input signals to the network. Neuron functionality can be tuned by a learning rule adapting, say, the threshold value for the

$$\begin{array}{lcl}
Topology & ::= & Path\ PathList^* \\
PathList & ::= & \text{``;''}\ \ Path \\
Path & ::= & InputNeuron\ NeuronList\ OutputNeuron \\
NeuronList & ::= & (CortexNeuron\ \mid\ OutputNeuron)^* \\
InputNeuron & ::= & i, \text{ where } i \in Sensory \\
OutputNeuron & ::= & o, \text{ where } o \in Output \\
CortexNeuron & ::= & k, \text{ where } k \in Cortex
\end{array}$$

Figure 5: Production rules P of grammar G for net topology description

internal activation function. Finally, weight changes may be achieved with error-correcting rules (Hebb, generalized delta etc).

4 Net topology evolution: A grammar-based approach

Concerning the problem domain of net topology evolution we had especially two ideas in mind:

First of all, we want a parameter coding that is easily interpretable by human experts without the need to explain complicated decoding algorithms. This means that the structures we want to evolve should be rather close to (easily understandable) formal descriptions of the problem domain. [1]

Secondly, we wanted our system to behave much like a human expert trying to systematically improve network performance, although the evolution system should be far more efficient and persistent. With these operators the evolution processes should be much more intelligible, especially for people not interested in a deep understanding of evolution system tuning. This is an important prerequisite for an evolution system which shall be used not only for toy problems, but for "real world applications". [2]

In the following sections we will exemplify the basics of our approach to net topology evolution; this is only one level in our evolution system but the same ideas are applicable to the other levels as well. We will describe the coding and the operators used in evolving net topologies for the neural net model depicted in figure 1.

[1]Impressive results have been obtained by Koza [?] with the genetic programming paradigm based on the evolution of LISP-S-expressions. A more general grammar-based approach and some more arguments for the use of higher-order, problem-dependent codings have been presented by Antonisse [?].

[2]It seems to be a general problem of genetic or evolutionary systems that control parameter tuning remains a very difficult problem. This is especially true for the domain of neural network optimization. But this only means that the "black art problem" only has been shifted from one domain to another.

4.1 Grammar-based genetic parameter coding

In the current version of our network design system we use a contextfree grammar to describe and evolve strings that represent net connectivity structures. [3]

Our (contextfree) grammar $G := (N, T, S, P)$ is charcterized by

- non-terminals $N = \{$ *Topology*, *Path*, *PathList*, *NeuronList*, *InputNeuron*, *CortexNeuron*, *OutputNeuron* $\}$,
- terminals $T = \{\text{``;''}\} \cup Sensory \cup Cortex \cup Output$,
- a startsymbol $S = Topology$ and
- production rules P as specified in figure 5. [4]

We refer to n input neurons, m output neurons and a previously undefined number of cortex neurons by the following sets of symbols:

$$\begin{array}{rcl}
Sensory & := & \{i_1, i_2, ..., i_n\} \\
Output & := & \{o_1, o_2, ..., o_m\} \\
Cortex & := & \{1, 2, 3, ...\}
\end{array}$$

Note that the numbers n and m of sensory and output neurons, respectively, are predefined due to the problem domain, whereas the number of cortex neurons has to be evolved.

[3]A first advantage of the grammar approach is that the rules of the grammar can be used to generate strings which then automatically belong to the language $\mathcal{L}(G)$ of the grammar. Within the current prototype implementation strings are not modified by operators referring directly to the given grammar G; this will be done in future versions of the evolution system. Closure with respect to $\mathcal{L}(G)$ is only assured by appropriate definitions of the string operators.

[4]The non-terminals on the left side of the "::="-sign can be replaced by the strings on the right side. The string creation process begins with the startsymbol. A string t is defined to be in the language $\mathcal{L}(G)$ of G if t can be created from startsymbol S by a finite number of applications of the production rules P, and if t only consists of terminal symbols from T.

Each string generated by grammar G produces a list of paths from input to output neurons. As we will see even loops or recurrent connections can be modelled by this grammar. The following example string is produced by the grammar:

$$w = i_1 1223 o_1 \;;\; i_1 3 o_1 1 o_2 \;;\; i_2 21 o_2$$

The input neuron set is $Sensory := \{i_1, i_2\}$, the cortex neuron set is $Cortex := \{1,2,3\}$, and the set of output neurons is $Output := \{o_1, o_2\}$. Three paths describe the connection structure from input neurons over cortex neurons to output neurons (see figure 6):

$$\begin{aligned} p_1 &= i_1 \rightarrow 1 \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow o_1 \\ p_2 &= i_1 \rightarrow 3 \rightarrow o_1 \rightarrow 1 \rightarrow o_2 \\ p_3 &= i_2 \rightarrow 2 \rightarrow 1 \rightarrow o_2 \end{aligned}$$

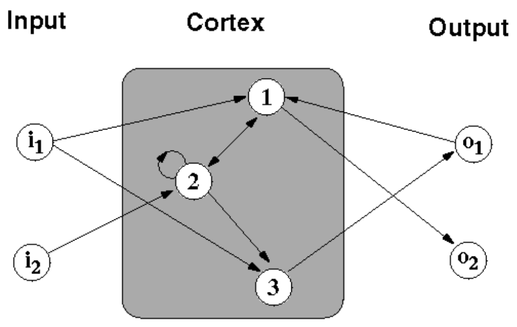


Figure 6: Example network produced by grammar G

As each path starts with an input and ends with an output neuron it is guaranteed that input signals eventually reach some of the output neurons and that there are no useless neurons which do not lie on any path from input to output neurons. Thus, all cortex neurons take part in the calculation of the output signals. If cortex neuron 3 had no output connection then it would be useless as it would only receive signals but never pass its own signals to other neurons; this effect is automatically prevented by the grammar.

The path descriptions can be decoded easily into a network (like in figure 6) by mapping the paths into an adjacency matrix; with this matrix at hand duplicate parts of the path description are eliminated (as e.g. the edge from neuron 1 to neuron o_2 in path p_2 and p_3). As we will see in the following sections, redundancy of the connectivity coding is essential for the evolution process.

4.2 Mutation on grammar-based chromosomes

Mutation on the grammar-based chromosomes should assure that we do not leave the language $\mathcal{L}(G)$ of the grammar G, i.e. we want closed operators on the strings.

Let $w = p_1; p_2; ...; p_n$ be a path concatenation, and let w' be the path resulting from applying the following mutation operators.

1. Create a new path p and insert p into w at position k:
$$w' = p_1; p_2; ...; p_{k-1}; p; p_k; ...; p_n$$
2. Remove a path p_k from w:
$$w' = p_1; p_2; ...; p_{k-1}; p_{k+1}; ...; p_n$$
3. Select a path $p_k = i_k c_{k1} c_{k2} ... c_{km} o_k$ from w and insert a new neuron $c \in Cortex \cup Output$ anywhere between i_k and o_k. Neuron c can be one of the neurons still available or can be an additional cortex neuron:
$$p'_k = i_k c_{k1} ... c_{kl-1} c c_{kl} ... c_{km} o_k$$
$$w' = p_1; p_2; ...; p_{k-1}; p'_k; p_{k+1}; ...; p_n$$
4. Select a path $p_k = i_k c_{k1} ... c_{kl-1} c_{kl} c_{kl+1} ... c_{km} o_k$ from w and remove a neuron c_{kl}:
$$p'_k = i_k c_{k1} ... c_{kl-1} c_{kl+1} ... c_{km} o_k$$
$$w' = p_1; p_2; ...; p_{k-1}; p'_k; p_{k+1}; ...; p_n$$

It is easy to see that we do not leave the grammar language $\mathcal{L}(G)$ when we apply these operators. In our implementation we use these mutation operators in the following way:

- Operators (1) and (2) are responsible for global search in the path description domain. That is why they are used with rather low probability rates. [5]
- An "edge-add" operator selects a path from the chromosome, duplicates the path, introduces and deletes neurons through operators (3) and (4) on the duplication string, and then inserts the mutated path into the chromosome. This introduces duplicate connectivity descriptions but extends the topological structure very smoothly because

[5]The effects of operators (1) and (2) are comparable to the mutation operators used for genetic algorithms on bit-strings.

it integrates existing single neurons into existing paths; due to redundant coding this does not disturb the overall connection structures very much but may just introduce a signal path necessary to solve an input-output mapping. Degree of redundancy is restricted by limiting the path lengths as well as the number of paths per string.

- To remove single edges we use operator (4).

4.3 Crossover on grammar-based chromosomes

The crossover operator picks out two chromosomes $chrom_1$ and $chrom_2$ and selects two crossover points k_1, k_2 and l_1, l_2 within each chromosome, respectively. The crossover points must lie between the paths, i.e. at the locations of the path separators ";". Then the path lists of $chrom_1$ and $chrom_2$ between the crossover points – the chromosomes are treated as ring structures – are exchanged resulting in new chromosomes $chrom'_1$ and $chrom'_2$: [6]

$$\begin{aligned} chrom_1 &= p_1; \ldots; p_{k_1-1}; \underline{p_{k_1}; \ldots; p_{k_2}}; p_{k_2+1} \ldots; p_n \\ chrom_2 &= q_1; \ldots; q_{l_1-1}; \underline{q_{l_1}; \ldots; q_{l_2}}; q_{l_2+1} \ldots; q_m \end{aligned}$$

$$\begin{aligned} chrom'_1 &= p_1; \ldots; p_{k_1-1}; \underline{q_{l_1}; \ldots; q_{l_2}}; p_{k_2+1} \ldots; p_n \\ chrom'_2 &= q_1; \ldots; q_{l_1-1}; \underline{p_{k_1}; \ldots; p_{k_2}}; q_{l_2+1} \ldots; q_m \end{aligned}$$

5 Some implementation details

5.1 Simulation system

Currently we have implemented levels 1 and 3 of the design hierarchy (see fig. 3). [7] The topology and weights creation modules can be distributed over a network of workstations communicating via socket interfaces. The topology module creates path descriptions as referred to in the last section and sends these strings in the compressed form of connectivity matrices to an array of subprocesses. These weights creation processes work independently from each other; they evaluate each topology string they have been sent by generating a pool of weights settings for the received topology. Each weights setting together with the topology then describes a fixed network structure which now can be evaluated for a predefined test environment (e.g. a pattern classification or parameter control task). For a fixed number of generations a genetic algorithm [8] for weights evolution then tries to find an optimal weights setting within the given environment. Finally, a weights string evolves which lets the network solve its task in an optimal way, and a fitness value serving as a performance measure for this network structure is returned to the topology module. These fitness values then control the evolution process at level 1.

[6]Without loss of generality: $k_1 \leq k_2$, $l_1 \leq l_2$ and $d_1 := (k_2 - k_1) < (l_2 - l_1) =: d_2$

[7]The functional properties of the processing elements (neurons) remain fixed during the evolution processes.

5.2 First simulation results

For our first test experiment we defined a control task similar to the well-known pole-balancing task. A small ball thrown on a seesaw at random position and with random initial speed has to be balanced to the seesaw's centre. The control task is said to be successfully solved whenever the ball comes to rest near the seesaw hinge (see fig. ??).

The seesaw is controlled by networks as depicted in fig. **??** consisting of three input neurons and a single output neuron. The suitable number of cortex neurons has to be evolved. The input neurons take the current seesaw angle, the velocity of the ball and the position of the ball, accordingly. The output neuron controls the seesaw's delta angle.

5.2.1 A simple test experiment

Evaluation of a population of networks – all with the same topology structure – is performed as follows:

1. *Evaluate each weight setting:*

   Select initial position and speed of the ball.

   Let the network control the ball for a fixed number of cycles.

   To measure the network's performance distinguish three cases:

   (a) *The ball is tossed from the seesaw.*

   (b) *The ball is still on the seesaw but does not come to rest.*

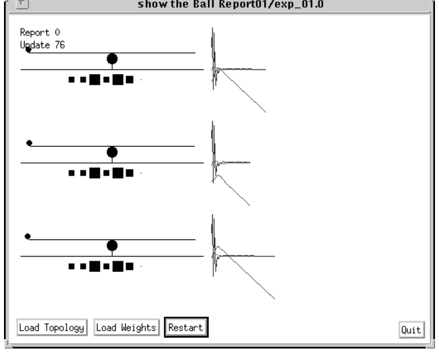
   (c) *The network succeeds to bring the ball to rest; the ball's distance to the seesaw's centre is taken into account.*

2. *Perform selection and other GA-operators for a predefined number of generations on the weights setting population.*

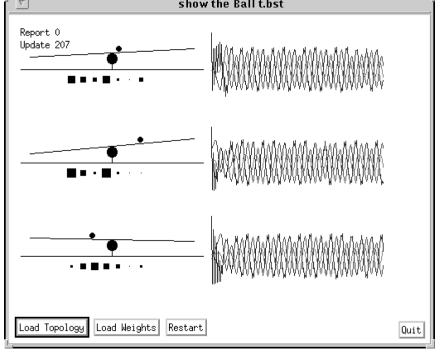
3. *Re-evaluate the best evolved weights settings:*

   In order to calculate a normalized fitness value, select initial test positions and speeds of the ball for a

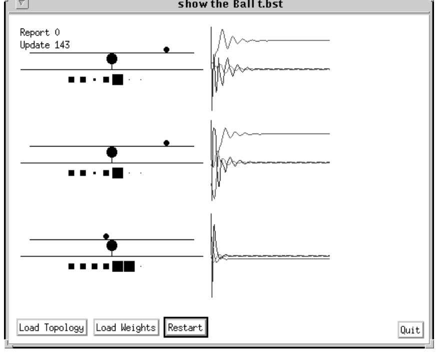
[8]We use binary as well as floating point representations for the weight values.



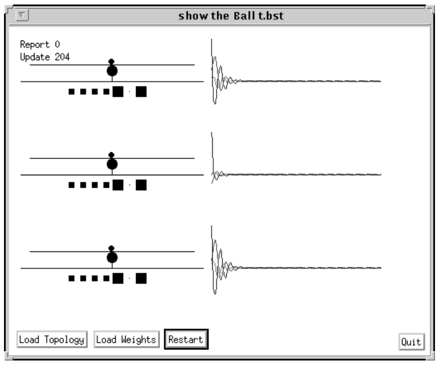
a.



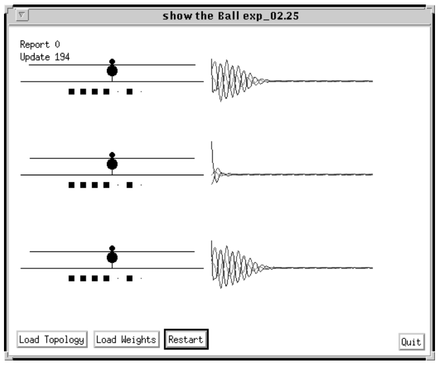
b.



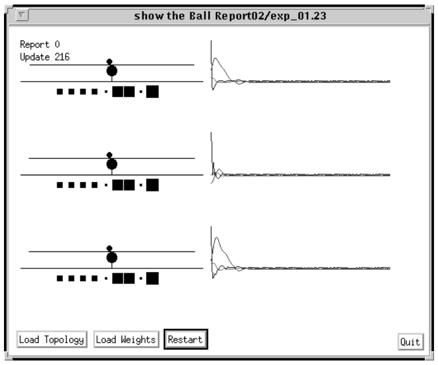
c.



d.



e.



f.

Figure 7: Performance of evolved control networks

fixed number of times. Let the network control the ball for a fixed number of test cycles and measure its performance.

Return the mean of these fitnesses as the final fitness value for the champion weight setting and the current topology. [9]

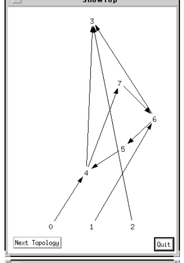
5.2.2 Evolved networks

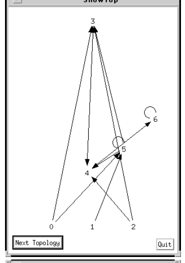
Figure **??** gives a brief overview of control networks evolved. Each picture shows a seesaw (with its control net depicted below) for three different initial settings for the position and speed of the ball. The graphs on the right side plot the ball's position and speed and the seesaw's angle for every update cycle. Interestingly enough, the evolution process has developed very different strategies to control the ball: The worst strategy is not to react at all; finally the ball falls off the seesaw (a). Other networks try to keep the ball on the seesaw by periodically balancing (b); mostly the ball never comes to rest. Very smooth reaction of the seesaw sometimes succeeds in stopping the ball from rolling, however, the ball rests far from the seesaw's centre (c). The best strategies force the ball to the centre by periodically balancing (d,e) – some networks need a long time for that – or by immediately slowing down the ball and carefully pushing it to the centre (f).

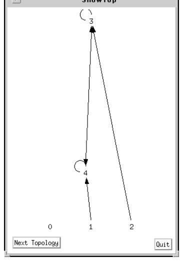
[9]This ensures that the returned fitness values for the different topologies are comparable, if for the calculation of the final fitness value the same test intervals and cycles are used.

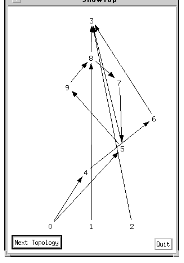
6 Conclusion and further research

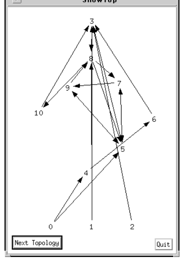
Up to now we only have some first and very simple simulation results which are promising, however. The hierachical approach has to be tested for more complex problems. Furthermore, it has to be investigated in how much the network design hierarchy helps in evolving proper net architectures according to a number of constraints and performance measures. Last but not least, the system is still missing a "grammar frontend" which generically produces string representations and genetic operators from a given grammar specification.

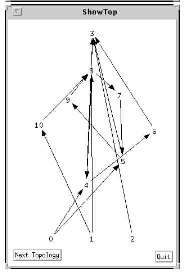

ShowTop
3
7
6
5
4
0
1
2
Next Topology
Quit

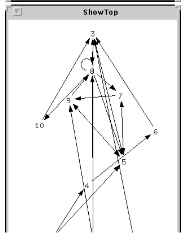

ShowTop
3
6
5
4
0
1
2
Next Topology
Quit


ShowTop
3
4
0
1
2
Next Topology
Quit


ShowTop
3
8
9
7
6
5
4
0
1
2
Next Topology
Quit


ShowTop
3
8
9
7
10
6
5
4
0
1
2
Next Topology
Quit


ShowTop
3
8
9
7
10
6
5
4
0
1
2
Next Topology
Quit


ShowTop
3
8
9
7
10
6
5
4