Evolutionary Design of Neural Networks

Marko A. Grönroos

Master of Science Thesis

Computer Science

Department of Mathematical Sciences

University of Turku, 1998

TURUN YLIOPISTO
Matemaatisten tieteiden laitos

GRÖNROOS, MARKO: Evolutionary design of neural networks
(Hermoverkkojen rakenteen evolutiivinen optimointi)

Pro gradu -tutkielma, 66 s., 8 liites.
Tietojenkäsittelyoppi
Kesäkuu 1998

Tämä tutkielma käsittelee menetelmiä sopivien hermoverkkorakenteiden löytämiseksi eri oppimisongelmille. Hermoverkot ovat laskennallinen menetelmä, jossa yksinkertaiset, toisiinsa kytketyt laskentaelementit voivat yhdessä muodostaa monimutkaisia funktioita. Elementtien ja kytkentöjen parametrien määrittämiseen on olemassa useita opetusmenetelmiä, mutta ne eivät yleensä osaa määrittää kullekin ongelmalle sopivaa kytkentärakennetta. Tutkielmassa käytetään evolutiivista algoritmia yhdessä neljän eri hermoverkkojen rakenteen geneettisen koodausmenetelmän kanssa. Menetelmät ovat Miller, Todd ja Hedgen suora koodaus, Kitanon graafingenerointikielioppi, Nolfi ja Parisin soluavaruuteen perustuva menetelmä, sekä Cangelosi, Parisi ja Nolfin generatiivinen soluavaruuteen perustuva menetelmä. Koodausmenetelmät määrittävät vain verkon rakenteen; painojen opetus tapahtuu käyttäen RProp-opetusmenetelmää. Evolutiivisen algoritmin tarvitsema kelpoisuusarvo määritellään opetetun hermoverkon laskentavirheenä erillisen arviointiaineiston suhteen.

Tutkielman alkuosassa käydään läpi hermoverkkojen, evolutiivisten algoritmien, sekä hermoverkkojen geneettisten koodausmenetelmien perusteet. Loppuosassa esitellään testiongelmat, sekä tulokset tehdyistä kokeista. Koodausmenetelmien suorituskykyä hyvän verkkorakenteen löytämisessä tarkastellaan kahdeksan eri oppimisongelman suhteen. Ongelmista neljä on keinotekoisia (XOR, koodaus ja kaksi funktion approksimoinnin ongelmaa), kolme on todellisia hahmontunnistusongelmia PROBEN1-ongelmajoukosta (syövän, lasityyppien ja sydänsairauden tunnistus) ja yksi on konkurssien ennustusongelma yritysten tilinpäätöstietojen perusteella. Koodausmenetelmien suorituskykyä mitataan luokitustarkkuudella hahmontunnistusongelmissa, sekä kyvyllä löytää oleelliset muuttujavalinnat keinotekoisissa ongelmissa.

Parhaimmaksi koodausmenetelmäksi hahmontunnistusongelmissa osoittautui graafingenerointikielioppi, vaikkakaan suora koodaus ei ollut oleellisesti huonompi. Muuttujien valinnassa parhaimmaksi osoittautui suora koodaus, vaikkakin Nolfi ja Parisin koodausmenetelmä pääsi melko lähelle sen tulosta.

Abstract

This thesis deals with methods for finding neural network architectures suitable for learning particular problems. We use an evolutionary algorithm with four different genetic encoding methods to search for the suitable architectures. We train the neural network weights with a separate neural learning algorithm. We use eight different learning problems for benchmarking the encoding methods. Four of the problems are artificial (XOR, Encoder and two function approximation problems), three are real-world classification problems from the PROBEN1 benchmarking problem set, and one is a bankruptcy classification problem studied earlier in one of our projects. Our evaluation criteria are classification accuracy and efficiency for using only the relevant variables. The classification results are compared also to those for network architectures found by a systematic search.

Keywords: neural networks, evolutionary algorithms, encoding methods

Contents

Preface

This thesis is the result of a research done at the Department of Computer Science of Åbo Akademi University, within the Countess (Computational Intelligence for Business) project, which is a long-term project done in collaboration with the Department of Information Processing of the Turku University of Economy.

I wish to thank the supervisors of this thesis, Olli Nevalainen and Kaisa Sere. I also wish to thank Kaisa Sere and Barbro Back for providing me the possibility to work in the Countess project. Additionally, I wish to thank the Department of Computer Science of Åbo Akademi for all the computing and other resources required by this work. Finally, I wish to thank Christian Lehtinen, Helmut Meyer, Kari Nilsson, and Mauno Rönkkö (and some others) for their helpful comments regarding this work.

Marko Grönroos
Turku, June 1998

Chapter 1

Introduction

This thesis focuses on comparing methods for the evolutionary design of *artificial neural network* (ANN) architectures. ANNs are nowadays a well-established computational paradigm in the field of *artificial intelligence* (AI). They are usually seen as a method for implementing complex nonlinear mappings (functions) using simple elementary units that are connected together with weighted, adaptable connections. We concentrate on optimizing the connection structure of the networks.

Evolutionary algorithms (EAs), inspired by the principles of biological evolution, are another paradigm in AI that has received much attention lately. EAs, which employ idealized models of genetic code, recombination of genetic information, mutations, and selection, have been noticed to yield very generic and robust ways for finding solutions for computationally difficult search problems. One such problem is the adaptation (training) of ANN architectures and parameters.

The training of ANNs using EAs has been a theme of much attention during the last few years. This interest spawns from many different sources, or points of view. One is that of the AI, where we want to let computers solve problems and learn things just by themselves without using a human programmer. A nearby field, statistical pattern recognition, has given neural learning many strong conceptual and mathematical tools for the analysis of different approaches. Although many quite efficient methods have been developed for the training of the connection weights, no definitive methods exist for the determination of ANN architectures most suitable for particular problems.

Another view is that of *artificial life*, where we try to create (potentially intelligent) artificial lifeforms using the same principles as those used by the nature in our evolution. The results acquired from such experiments are naturally of great interest to the fields of evolutionary and developmental biology, as well as psychology.

The methods we are interested in are those that could be used solely for the architectural design of the networks. We want to do the actual training of the network weights with a traditional (non-evolutionary) neural learning method, as they have been shown to be very efficient for that purpose. The evolutionary algorithm uses the error of such a trained network as a fitness value to guide the evolution. Many evolutionary methods for such architectural design have been developed during the recent years and the results

have been encouraging. However, we can observe that most of the methods have been tested in relation to only a single problem, which has often been a simple toy problem or an artificial life problem. Therefore, we set as our main goal of this work to compare the capabilities of some of these methods by applying them to various (more or less) difficult function approximation and pattern recognition problems. Four of the problems used in this study are artificial problems for which we can easily deduce very suitable neural architectures. They are followed by three real-world classification problems that are well known for benchmarking purposes in the neural learning field. The last problem is the prediction of bankruptcies using a dataset that has been studied within the Countess project earlier (Back, Sere, and Laitinen 1997).

We use two criteria for the evaluation of the performance of the methods; first is the classification accuracy of the best neural network topology found by the evolutionary search. This criterion is most meaningful in the real-world problems. The second criterion is the selection of relevant input variables, which is most meaningful with the artificial problems where we already know the correct answer. We can expect the more direct network encoding methods to have stronger control over the individual neurons, and therefore perform better in this task.

The thesis is structured as follows. An introduction to the relevant topics is given in Chapters 2 to 4. Chapter 2 is a short overview to artificial neural networks and to some of the problems in that field. Chapter 3 gives a similar overview to evolutionary algorithms and Chapter 4 deals with various methods for applying evolutionary algorithms to the design of artificial neural networks. Chapter 5 describes the problems (datasets) used for evaluating the performance of the methods. Chapter 6 gives the first results of some preliminary tests that were made to calibrate the parameters of the evolutionary algorithms and then results of the runs. A discussion of the results is given in Chapter 7. The chapter includes conclusions and gives some future directions for further work. We implemented the algorithms required by the study with the C++ language, using a heavily object-oriented approach. Appendices A, B, and C give a brief description of the ANN program library, the evolutionary algorithm library, and the evolutionary ANN library, respectively.

Chapter 2

Artificial neural networks

The history of artificial neural networks (ANNs) can be traced far back to the times of birth of the digital computers (McCulloch and Pitts 1943; Hebb 1949). There was some very enthusiastic research done on neural networks in the fifties and sixties (Rosenblatt 1958; Minsky and Papert 1969), but then the neural networks were almost forgotten for almost two decades. The boom that begun at the end of the eighties is still going on.

The majority of applications of ANNs are usually divided into three categories: *classification* (*pattern recognition*), where the network tries to classify signal patterns into predefined categories; *prediction*, where the network tries to extrapolate an input series, and *control*, where the network is used to interactively guide some external process or device. The first two are basicly cases of approximation, where we want to approximate some (typically numerical) function.

The introduction below describes a somewhat formal view of neural computation and learning, as the ANN research has in recent years drifted somewhat far away from the biological metaphors.

2.1 Elementary structures

A *neural network* is, as the name implies, a group of *neurons* connected together with (typically one-directional) *connections*. The idealized model of biological neurons used in most ANNs is very simple. A neuron is seen as an elementary computational *unit* that receives *input signals* from other units, sums them up and then sends out an *output signal* according to a simple *threshold function*. The output value ("activation") of jth node of a network is denoted here as y_j. Additionally, when signals are transferred from unit to unit, they are weighted according to connection weights w_{ij}. This is formalized as the *transfer function*

$$y_j = \theta\left(\sum_i w_{ij} y_i\right) \tag{2.1}$$

where the y_i are input signals from source units (see Figure 2.1). By summing up an arbitrary number of these elementary nonlinear functions in an appropriate manner, any

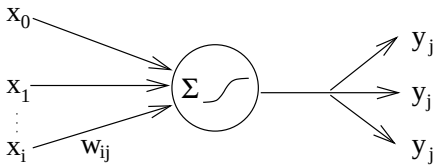

Figure 2.1: An artificial neuron that sums its inputs and squashes the input value with threshold function to form the output value, which can then be used as an input value by some other neurons

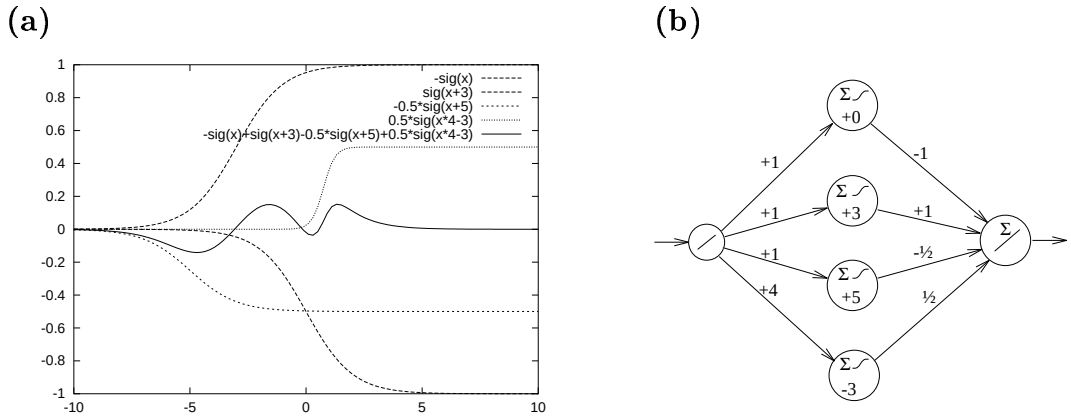

Figure 2.2: An example of forming function $f : \Re^1 \rightarrow \Re^1$ using elementary sigmoid functions (denoted as $sig(x)$) . (**a**) The curves for four sigmoid functions with different parameters (broken lines) and one for the summed result (solid line). (**b**) The corresponding neural network where the parameters are given as weights (shown above the connections) and biases (shown within the neurons). The output units have a linear transfer function (input units do not calculate any function as the network input values are placed as their outputs).

other continuous function can be approximated with arbitrary precision (Lapedes and Farber 1988; Bishop 1995a, Chapter 4). Figure 2.2 gives an example of this potential.

The threshold function (also called *squashing function*) is typically a nonlinear mapping $\theta : \Re \mapsto [0, 1]$. The threshold functions can be roughly divided into *discrete* and *continuous* functions. A discrete function is typically something like

$$\theta(x) = \begin{cases} 0 & , \text{if } x < \beta \\ 1 & , \text{if } x \geq \beta \end{cases} \tag{2.2}$$

where β is a threshold value, or *bias*. The most popular continuous threshold function is the *sigmoid function* (logistic function)

$$\theta\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.3}$$

We can use different threshold functions in the same network. For example, if we wish to have network output values outside the $[0, 1]$ range of the standard sigmoid function, we can use a linear (identity) threshold function $\theta(x) = x$ for the output units.

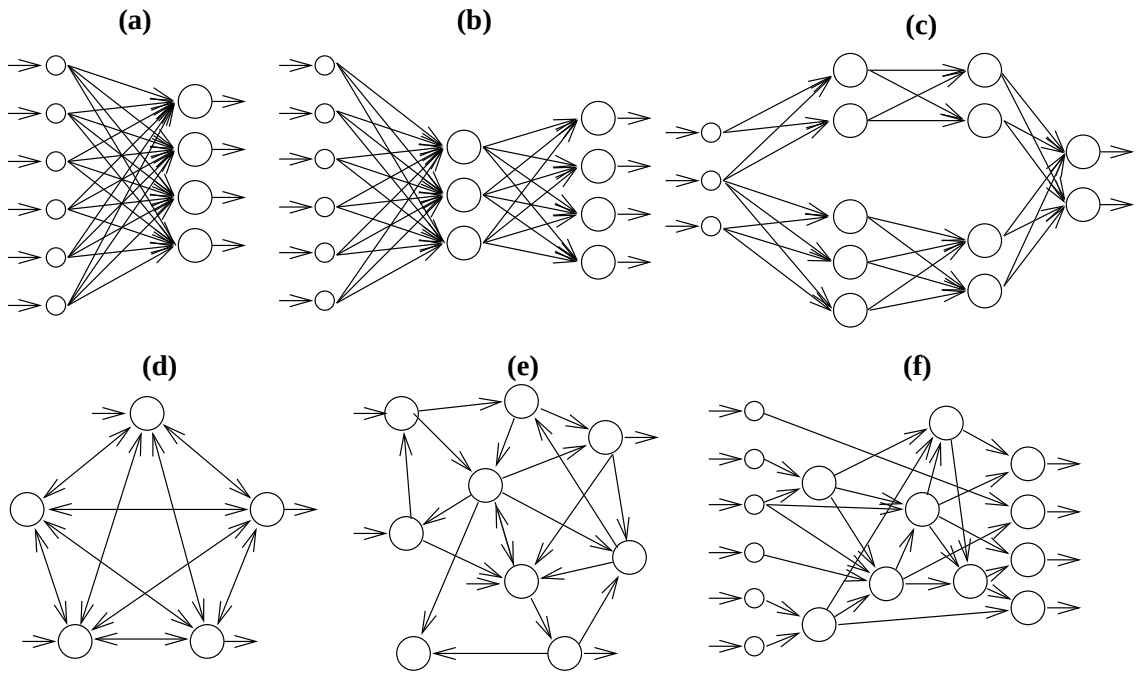

Figure 2.3: Some network topologies. (**a**) A fully connected single-layer perceptron (SLP). (**b**) A fully connected multilayer perceptron (MLP). (**c**) A modular MLP. (**d**) A fully connected recurrent network. (**e**) A sparsely connected recurrent network. (**f**) A feedforward network.

2.2 Network topology

The first question after having these elementary units is how they should be connected to each other. The way how a network is connected is called *network topology* or *architecture*. The network topologies used in this work are *feedforward networks*. They are a generalization of the *multilayer perceptrons* (MLPs) that are the most popular topology category used in neural computation. The single-layer perceptron (SLP), one of the oldest network topologies (Rosenblatt 1962; Bishop 1995a, pp. 98-105), consists of one *layer* of computational units; the input units do not perform any computation (see Figure 2.3a). The input layer is fully connected to the output layer, i.e., every input unit is connected to every output unit.

The *multilayer perceptron* (MLP, see Figure 2.3b-c) has additional layers, which are called *hidden layers*. Although most neural training algorithms are said to train MLPs, they can actually train any networks that have the *feedforward topology* (see Figure 2.3f). This generalization from MLPs is useful for evolutionary neural networks, as can be seen later. A simple definition of the feedforward topology is that any connection w_{ij} (from unit with index number i to unit with index number j) must meet criterion $i < j$. Another way to describe the feedforward generalization of MLPs is that there can be *shortcut connections* between the non-adjacent layers.

If a network does not meet the feedforward criterion above, it has some feedback connections. This topology class is called *recurrent* (see Figure 2.3d-e) and the most com-

mon training algorithms that work with feedforward networks do not work with recurrent networks without modifications.

2.3 Neural learning

There exists various training methods for different kinds of neural networks. We use a variant of the *backpropagation* algorithm (Rumelhart, Hinton, and Williams 1986) in this work. Backpropagation is an efficient, simple, and well-known neural training method. It is a so-called *supervised learning* algorithm, which means that the training is done with a set of *training patterns* (samples) that contain both the inputs and desired outputs. Another class of learning algorithms is *unsupervised learning*, where no corresponding output values are given.

The basic function of the training algorithm is to minimize the error between desired and actual outputs of the network. The error is typically measured as the *mean of squared errors* (MSE) of the outputs:

$$E = \frac{\sum_{p=1}^{N_p} \sum_{j=1}^{N_o} (y_j - o_j)^2}{N_o \cdot N_p} \tag{2.4}$$

where y_j and o_j are the actual and expected outputs, respectively, N_o is the number of elements in output vectors, and N_p is the number of patterns. The use of a continuous threshold function makes it possible for the training algorithm to use derivative of the transfer function to perform *gradient descent search* to find the set of weights that estimates the desired function best. The use of the derivative is the basic idea behind backpropagation, which uses it to get an estimate about the direction to which the weights of the network should be changed. See Appendix A for the details of the backpropagation algorithm and RProp (Riedmiller 1993), its variant.

Crosstalk

One problem with neural learning is *crosstalk* (Jacobs, Jordan, and Barto 1990) which occurs when the network is expected to learn several unrelated problems at the same time. Crosstalk has two modes: *spatial* and *temporal*. The former occurs when a single training pattern has information about two or more problems. The latter occurs when successive training patterns are from different problems. It was noted by Jacobs et al. that modular structure of the network would help solving the problem. According to our view on crosstalk, the different problems would be expected to compete with each other for their representation in the modules during the training, and in that way self-organize to the network topology. One purpose of network topology selection (for example by using evolutionary search) would therefore be to order the network to appropriate modules. The training error is the result of a complex function of the topology, and is affected by numerous factors that do not seem to be separable.

2.3.1 Complexity, overfitting and regularization

We can distinguish three forms of complexity (Duane 1996) in neural learning and pattern recognition:

- Complexity of the problem (the data)
- Complexity of the model (neural network)
- Complexity of the learning/recognition algorithm

The compatibility of the model and problem complexities is a common problem in all data modeling and pattern recognition. If a model has too many degrees of freedom, it can easily *overfit* to the data. This means that even though a model could predict the training data exactly, its ability to generalize can be extremely poor. The problem grows even bigger when noise (which is almost always present in any real-world data) is taken into account. Then the model may start to overfit to the noise, which has brought false complexity to the problem. Of course, if the model complexity is lower than the problem complexity, the model will not be able to capture the function underlying the problem and the result will be over-generalization. It can be thought that the model and problem complexity are bound together by the third form of complexity, that of the learning algorithm, which can be measured in terms of time complexity. The early-stopping method described below is a way to control the model complexity by adjusting the time complexity.

In information theory, the model complexity is discussed for example in terms of the *minimum description length* (MDL) principle (Bishop 1995a, pp. 429-433).

Network topology

Complexity of a neural network is governed by its number of adaptive parameters (weights). Thus, the selection of the network topology directly affects the complexity.

A network with two hidden layers of neurons that is fully connected between the layers can, in theory, represent any mapping that network with some other topology can (Bishop 1995a, pp. 128-129). It can, however, be more difficult to train such a network in practice. Its generalization ability can also be worse than with some other topology.

Several methods have been proposed to optimize the structure. These can be roughly divided into *constructive and destructive methods*. An example of constructive methods is the *cascade correlation* learning algorithm (Fahlman and Lebiere 1991; SNNS 1995). It builds the network from scratch, adding one unit at a time and training the network after each addition. The destructive methods are usually referred as *pruning methods*. They take some network with many interlayer connections which they then cut one by one. Some examples are magnitude based pruning, Optimal Brain Damage (OBD), Optimal Brain Surgeon (OBS) and skeletonization(SNNS 1995).

The evolutionary neural networks (see Chapter 4), that are the subject of this work, are one alternative for this task of controlling the model complexity by adjusting the number of adaptive parameters (weights).

Regularization

Regularization methods that encourage smoother network mappings by adding a *penalty term* Ω to the error function E (Bishop 1995a)

$$\tilde{E} = E + v\Omega \tag{2.5}$$

where v is a regularization coefficient which controls the model complexity. Use of such complexity regulation terms is coherent with the MDL principle. The most popular regularization method is *weight decay* which is usually implemented simply by multiplying all weights of the network by some value $\beta \lessapprox 1$ at the end of each training cycle. Weight decay can also be given as penalty term to the error function

$$\Omega = \frac{1}{2}\sum_i w_i^2 \tag{2.6}$$

where w is a vector containing all weights and biases of a network. This encourages the weight vectors to stay near zero, in the linear part of the sigmoid function, thus reducing the *effective number of parameters* of the model (Bishop 1995b; Sarle 1995; Moody 1994).

Early stopping

Early stopping (Bishop 1995b; Sarle 1995; Moody 1994) is basicly a way of controlling the model complexity by terminating the training when the model begins to overfit to the data. It is also a way for making the training faster, which is important with evolutionary neural networks (see Chapter 4), when we have to train numerous network topologies during the search to find the best one.

The available training patterns are divided into two parts: a *training set* and a *validation set*. The training set is used for the actual training of the network. Then, at certain intervals, the network is evaluated with the validation set. The purpose of this is to measure the generalization ability of the network. When the validation error begins to rise significantly, it is seen as a sign of overtraining and the training is terminated. The weight state of the network is stored after each validation test. After the training is terminated. the network state with the lowest validation error is restored.

We used the *generalization loss* (GL) termination criterion (Prechelt 1998) in our experiments. GL is defined as

$$GL(t) = 100 \cdot \left(\frac{E_{va}(t)}{E_{opt}(t)} - 1 \right) \tag{2.7}$$

where $E_{va}(t)$ is the validation error at training epoch t and $E_{opt}(t) = min_{t' \leq t}(E_{va}(t'))$ is the minimum validation error so far. Notion GL_α is defined so that the training is terminated after $GL(t) > \alpha$. Prechelt has used parameters around GL_2 and GL_5.

Chapter 3

Evolutionary algorithms

Evolutionary algorithms (EA) are a collection of methodologies inspired by the principles of the biological evolution. They have shown to be very applicable and efficient for certain kinds of computationally difficult search problems.

The idea of *genetic algorithms* (GAs) was first introduced by John Holland in 1960s. He saw the genetic code as a sequence of binary values ("chromosomes") that could be evolved with a computer program. His algorithm is often referred to as the "canonic" genetic algorithm. Other implementations of EAs, dealing with real values and other datatypes, have also been developed.

EAs can, from the point of view of computer science, be thought of as "only" heuristic search methods (strategies). Most traditional heuristic search methods use a single candidate solution, which is changed ("mutated") according to some (often stochastic) rule. If the result is better than the old candidate, it is adopted. In contrast to that, EAs have a *population* of candidate solutions (individuals). The life-cycle of such populations is illustrated in Figure 3.1.

Using the concepts familiar from the biological genetics, *genotype* is the genetic constitution of an individual. The word *genome* is used to refer to the genotype in a more concrete level; in biological organisms, the genome consists of a set of *chromosomes* made of nucleic acids. In GAs, it is typically a string of binary values. What is usually needed, is a mapping from the genotype to a *phenotype*, which is the "decoded", grown[1], individual. Considering this from the other direction, the phenotype can be said to be *encoded* in the genotype.

After the population of individuals has been so decoded, their *fitness* must be evaluated in relation to some sort of *environment*. Since we are evolving solutions to some particular problem in EAs, that problem is our "environment". The phenotypes of the individuals could, for example, be just plain real-valued vectors that are given as a parameter to some function we wish to optimize. In the context of this work, the phenotype consists of a neural network, and possibly some additional parameters. After the evaluation, the next generation of the population is formed typically by selecting two parents for each new

[1]Note that there is typically no interaction between the individual and its environment in EAs during the ontogenesis (growth or decoding process) of the individual.

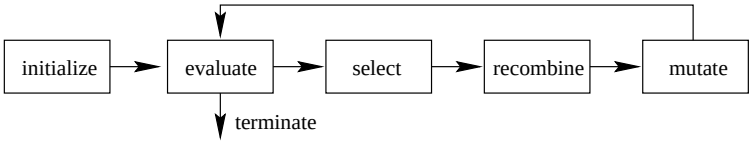

Figure 3.1: Life-cycle of populations in EAs

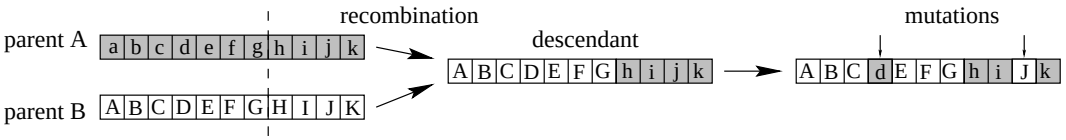

Figure 3.2: Recombination and mutation in EAs

individual, and mating the parental genomes to form the genomes for the offspring. The offspring are finally *mutated* a little. *Recombination* of the parental genomes and mutation of the offspring are illustrated in Figure 3.2.

3.1 Encoding

Possibly the most difficult task in all problem solving is defining the problem in a formal manner, i.e., finding a representation that can be handled easily by the solving paradigm at hand. EAs have been used with boolean, integer, real valued, combinatorial, parse tree, logic expression, graph and many other kinds of search spaces. We do not actually need any mapping from genotype to phenotype, as we can evolve any phenotypic structures directly, if we just implement the basic mutation and recombination operations for them. However, to make our search algorithms more general and easier to interface, we often wish to use as simple representations as possible. The question is also about scalability, as the encoding sometimes includes some sort of compression of data. Such encodings are expected to generate more regular phenotypic patterns, as those observable in biological organisms. Furthermore, using a decoding process also distorts the search space in a manner that may (or may not) help in the search. In too direct an encoding the search might get stuck in a local minima, while appropriately complex encoding might help get over them.

3.1.1 Real values

Binary code

Binary code is a straightforward way to encode real values in genetic algorithms. The binary representation of a real value is a string of binary values $\vec{b} \in \{0,1\}^l$. The string is decoded to a floating-point value $v \in [v_{min}, v_{max}] \subset \Re$ by dividing the value range into 2^l

discrete parts so that

$$v = (v_{max} - v_{min})\frac{\sum_{i=0}^{l-1} b_i 2^i}{2^l} + v_{min} \tag{3.1}$$

Gray code

In this work, we use only the Gray code, which is a common alternative for the plain binary coding. It is also based on binary values, but mutations affect it somewhat differently from normal binary coding. The Gray code is often preferred because it is believed to yield smoother evolutionary landscapes. This follows from the fact that successive codes differ by only one bit from each other, which can be observed in Table 3.1. A Gray-coded value is converted into a standard binary value with

$$b_i = \bigoplus_{j=1}^{i} g_j \tag{3.2}$$

where g is a Gray-coded binary vector and the $\oplus$ denotes the XOR operator.

Integer	Standard	Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Table 3.1: Comparison of binary-coded and Gray-coded integer values.

3.1.2 Integer values

Encoding integer values with binary values is equivalent to encoding real values *if* the range can be divided by 2^n (the mapping is a bijection). If that is not the case, the mapping must overlap in some way. That case is not discussed further here as the ranges needed in this work can easily be set as a power of two.

However, for some purposes (for example, the ANN encoding method by Kitano in Chapter 4), we do not use any encoding at all for integer values. Such values are mutated totally randomly within their domain.

3.2 Fitness

The "fitness" property of individuals is usually measured as the performance of an isolated individual in relation to the target problem ("objective function"). In biological systems, there is no such explicit information available, but the evolution is based on implicit fitness of the individuals and any measurements depend on artificial criteria.

3.2.1 Artificial fitness factors

The use of an explicit fitness value offers a possibility to combine different fitness factors, i.e., objective funtions. The error of a trained neural network is one obvious candidate as the primary fitness factor. Regularization techniques can be used to minimize the model complexity to fit the problem complexity, as was noted in Chapter 2. Similar technique has been used for evolutionary neural networks (Fredriksson 1997) by calculating the fitness (Φ) as a linear combination of different fitness factors

$$\Phi = \sum_i \alpha_i \varphi_i \tag{3.3}$$

The coefficients are normalized so that $\sum_i \alpha_i = 1$. Fredriksson (1997) used RMSE (root of MSE), classification accuracy, number of genes, number of hidden units as a ratio to a maximum value, and number of connections as a ratio to a maximum value. This technique is compatible with the MDL principle (mentioned in Chapter 2) and is identical to that used for regulation in training of neural network weights. Although the technique is interesting, it was not used in the experiments of the present study.

3.2.2 Noisy evaluation

Many of the evolutionary rules change radically when the evaluation contains noise. Evaluation of neural networks contains much noise, so robustness is a very relevant issue. Evolutionary algorithms are generally believed to withstand noise quite well (see the notes about the effect of noise with elitism below).

3.3 Selection

There are many ways to select the parents to be mated. One of the most popular ways is the *fitness-proportional* selection, where the individuals are given selection propabilities proportionally to their fitness value. This method requires some sort of scaling (linear or nonlinear) to bring the fitnesses of the population in some suitable range and distribution (in nonlinear scaling).

3.3.1 (μ, λ)-selection

A simple method for selecting parents is the one where we first order the individuals of the population according to their fitness value. The size of the population is given by the parameter λ. We then select the two parents for each descendant in the next generation from the μ fittest individuals with uniform propability. This method is often referred to as (μ, λ)-selection (Bäck 1996). We allow the selecting of the same individual as both parents in our experiments.

3.3.2 Elitism

Elitism means that we are saving some of the fittest individuals (the elites) intact to the next generation. This prevents us from losing good solutions by accident, as some selection methods might not select some important solutions due to randomness in the selection process. The good solutions might also get mated with incompatible solutions, thus producing fatal offspring (Branke 1995).

Elitism has been critizised for its tendency to converge prematurely, typically to bad local minimas (Bäck 1996, p. 79). It also hinders the self-adaptation of mutation rates (see below). Some studies (Kitano 1990a; Back, Sere, and Laitinen 1997) employing evolutionary ANNs have used elitism, some (Dellaert 1995; Fredriksson 1997; Fullmer and Miikkulainen 1991; Gruau and Whitley 1993) in form of the extremist *steady-state* EA strategy, where only one individual is replaced at a time. The selection strength of the steady-state EA is low, so it should not be so susceptible to premature convergence.

A strange, and possibly dangerous, phenomenon was noticed in some of the early tests done for this study. Consider a fitness function that returns normally distributed random values. With such a function no actual evolution is of course possible. If we use elitism, and evaluate the elites just once in their lifetime, we can observe an *illusion* of evolution, as the highest fitness in the population raises in a logarithmic fashion as more random values are generated. This is demonstrated in Figure 3.3 (which was selected as the most illustrative case from about 10 runs). Curve `2-arg` shows what happens, if we reevaluate the winner's "fitness" once and average the new value with the previous one. Curve `max-avg` shows what happens if the winner is reevaluated in every successive generation by averaging the new measurement with all the previous measurements for that individual. This effect diminishes, if the evolution run is done several times, and the fitness values of the different runs at specific generations are averaged. Without elitism, the average fitness of the "winning" individual would obviously be 0.0, if the population size is one, as it is in the figure.

3.4 Recombination

After some pair has been selected by the selection method for mating, their genomes are recombined to form one or more offspring. This operation has been modeled after the biological processes of crossover (interchanging segments between homologous chromosomes)

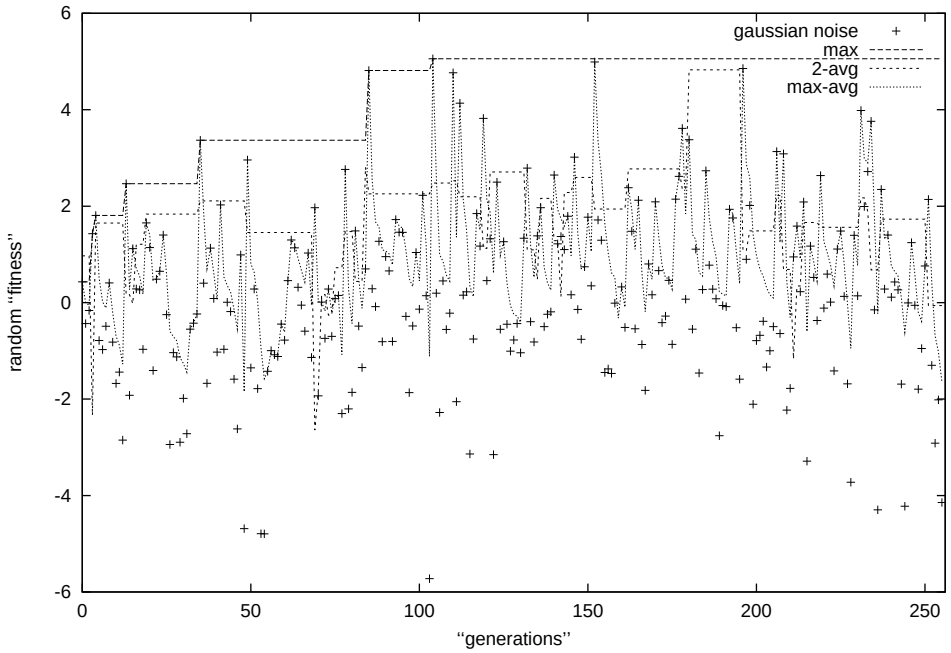

Figure 3.3: Effects of noisy evaluation with elitism. The "fitness" is a normally distributed random value $N(0,1)$.

and independent assortment of the chromosomes during meiosis. There is typically no "meiosis" in EAs, but the recombination occurs during the "fertilization".

The recombination of simple binary genes is illustrated in Figure 3.2. There are various methods to recombine other gene types. For example, the recombination of real-valued genes can be done by averaging, or by taking a random value between the parental values, etc.

We did not, in our experiments, use a recombination operator for the real-valued genes, just for their containers. The recombination is generally handled in a hierarchical manner (see Appendix B), which we do not expect to yield radically different results in relation to the standard (non-tree-structured) recombination procedure.

3.5 Mutations

Each gene type has a set of mutation operators. The standard operator for the binary genes simply flips bits with propability p_{binary}, which lies typically somewhere between 0.001 and 0.1. The operator for uniform (non-binary encoded) integers randomly selects a new value with propability $p_{integer}$. The standard operator for real-valued genes generates new value from

$$x' = x + N(0, \sigma) \tag{3.4}$$

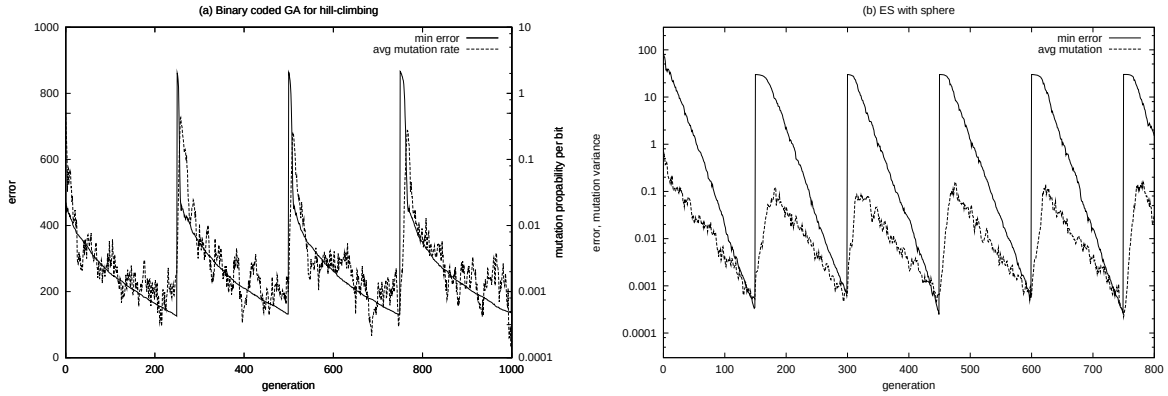

Figure 3.4: An example of the self-adaptation of mutation rates. (**a**) Autoadaptive GA with hill-climbing test-function. (**b**) Autoadaptive Evolution Strategies with the sphere test function.

where σ is the mutation variance for real-valued genes (see below how it is determined) and $N(0,1)$ is a normally distributed random value with average of zero and variance of one.

3.5.1 Self-adaptation of mutation rates

Self-adaptation of mutation rates (Bäck 1996) was used in the experiments of this study. The genomes of the individuals contain genes that control the mutation rates of different gene types. Figure 3.4a illustrates autoadaptive GA for mundane hill-climbing test function $f(\vec{x}) = \sum_{i=1}^{n} |x_i - o|$. The number of elements in the binary-valued vector is $n = 1000$. The objective value o changes every 250 generations (alternating $o = 0$ and $o = 1$). Figure 3.4b illustrates autoadaptative Evolution Strategies (ES, an EA paradigm that uses real-valued genotypes) for the "sphere" test function $f(\vec{x}) = \sum_{i=1}^{n} (x_i - o)^2$ with the objective value changing every 150 generations. The dimension of the real-valued genome is $n = 30$. The landscape of this very simple function contains just one optimum point and the optimal population size would be 2. The strategy parameters for both tests are: $(15, 100)$-selection, and no recombination.

In our implementation, the gene *rateBinary* controls the propability of binary gene mutations (p_{binary}), *rateInt* integer-valued gene mutations ($p_{integer}$), and *varianceFloat* controls the variance of the real-valued gene mutations (σ). We use real-valued representation for the mutability genes. They are mutated with mutation operator

$$\sigma' = e^{N(0,\tau' \cdot \sigma)}, \tag{3.5}$$

where τ' is a mutation "meta-rate" for mutation rate genes. Schwefel (1977; also in Bäck 1996, p. 72) suggests $\tau' \propto 1/\sqrt{2n}$, where n is the number of real-valued genes. We use $\tau' = 1/\sqrt{30}$ and range $\sigma \in [0.001, 1)$ in our experiments for all the three gene types.

We used, in our experiments, only global self-adaptation in respect to the whole genome of each individual. Another option would be to use local self-adaptation for each gene.

3.6 Discussion

Evolutionary algorithms have been found to be powerful tools for various search problems. Their primary strengths are generality and robustness. Analytical studies on their properties have not given many practical answers. The most rewarding studies have been statistical analyses of their performance when applied to various artificial and real-world problems.

3.6.1 Epistasis

Epistasis means, in biological genetics, the direct interaction of different genes so that the expression of one gene depends on the expression of some other genes. In a more general sense, it means functional dependency of genes in relation to the overall fitness. By this definition, most genes are more or less epistatic with each other in most evolutionary search problems.

Kauffman (1993, 1995) has studied the effects of epistasis by using the connectivity parameter of his *NK-networks*[2] to tune the complexity of search landscapes. His results illustrate the problem: the more connected a genetic regulation network is, the more rugged the fitness landscape becomes. The landscape becomes totally random at the point of total connectivity. Nevertheless, some level of epistasis seems to be helpful, if not essential, to evolution in some problem landscapes. This may suggest that the connectivity, or amount of conflicting constraints, may be an important factor in many kinds of evolving systems, including non-biological.

Holland (1975) has given another view ("schema theorem") of epistasis in genetic algorithms. He sees that the optimal solution consists of small subsolutions that compete with each other. The crossover operation combines the fittest subsolutions to form the optimal solution. His theorem pays great attention to the disruptive effects of recombination and mutation to the subsolutions. The theorem implies that the number of genes in the epistatic clusters of each subsolution should be as small as possible, as should be their genomic distance.

[2]The NK-networks are random graphs where the N refers to the number of elements and K to the average number of connections for each element.

Chapter 4

Evolutionary neural networks

In this chapter, we give a short introduction to some of the most well-known approaches for evolving neural network topologies. For more comprehensive overviews on evolutionary neural networks see (Balakrishnan and Honavar 1995; Branke 1995; Kodjabachian and Meyer 1995; Mitchell 1996).

There are currently maybe dozens of different methods for evolving ANNs. We selected the first two methods studied in this work (methods by Miller et al., and Kitano) mostly because they are so well-known and easy to implement. The third method was selected because it had a temptating space-coordinate encoding that was hoped to yield networks that are modular and visually simple. The fourth method was selected because it is a derivative of the previous one, and employs a fractal growth process similar to the method by Kitano.

We also give below the implementation details for each method. The biggest difference between the original methods and our implementations is that two of the original methods did not use a separate neural learning algorithm, but the connection weights were adapted by the GA.

4.1 Direct encoding

Direct encoding methods decode the phenotype directly from the genotype so that each phenotypic feature is encoded by exactly one genotypic code. The only direct encoding method inspected in this study is the one by Miller et al. (1989) below.

4.1.1 Miller, Todd and Hedge

One of the earliest encoding methods is the one proposed by Miller, Todd and Hedge (1989). They encoded an $N \times (N+1)$ matrix in which each element encodes the type of the connection between two neurons in an N-neuron network. They also encoded the type of the bias value, hence the $(N+1)$ above. The value "0" means no connection (or bias) and "L" means learnable connection (or bias). This is illustrated in Figure 4.1. The network topology is restricted to be feedforward by ignoring any feedback connections

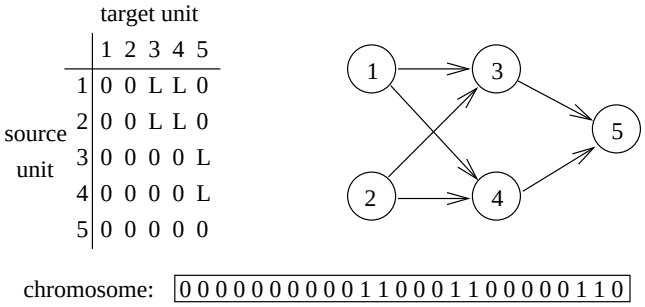

Figure 4.1: Miller-matrix, its genetic representation, and the resulting network topology. The bits in the genome iterate from row 1 and column 1 in the matrix, and continue down the column, and then right to the next column.

and connections to the input layer. Miller et al. used population size of 50, fitness-proportional selection, elitism of one individual, mutations with bitwise rate of 0.005, and recombination with propability of 0.6 in their experiments. Their recombination operator chooses a matrix column randomly, and swaps the respective columns from the parental genomes to create the offspring. The reasoning behind this operation is that each column represents all the incoming connections to a single unit, and this set is supposed to be a functional building block of the network.

The direct encoding method presented by Miller et al. is just one possible. Other possibilities would be, for example, the encoding of just the existence of input units, or the network topology could be restricted to some handcrafted topology. The former encoding has been used for finding the meaningful input variables (Back, Sere, and Laitinen 1997). Instead of encoding just the existence of the connections, we could also encode their weight values, as well as the bias values, or the transfer functions of the neurons explicitly.

This kind of direct approaches give strong control over the fine details of the network, but they lack scalability. It can result in huge genomes, as the order of required number of genes is usually the number of neurons squared.

Implementation details

Only the connections that are used for an feedforward topology are encoded in the genome. We do not encode the existence of the bias connections. On the other hand, we do encode the existence of the neurons, as we want to be able to control the selection of the input units more effectively. The output units are hard-coded to always exist (although they might not necessarily receive any connections).

Discussion

Analysis of the direct encoding method is not simple. There are two primary questions: first is that how likely it is for a network to be lethal, i.e., the inputs and outputs are not connected together at all? Another question is that a certain network topology can be encoded in many ways, and we would like to know exactly in how many ways, and how

that affects the adaptation. These are both difficult combinatorial problems, and we are not aware of any analysis giving definitive answers to them.

4.2 Our fractal origins

To build complex neural networks, researches have (again) looked at how it has been done in biological systems. The mapping from genotype to phenotype is definitely not direct in biological organisms. We are not built according to direct "blueprints" but according to indirect "recipes" (using the expressions by Dawkins, 1986). The "cooking process" is called the *ontogenesis* of an organism; the long and complex process or splitting and specialization of cells according to the rules encoded in the genome, with the adult individual as the result.

Many of the current evolutionary neural network methodologies have been inspired by the *Lindenmayer-Systems* (L-Systems). The formalism and an application of this idea was introduced by Lindenmayer (1976), originally as a *grammatic* approach for modeling plant morphogenesis. The grammars of an L-system consist of a set of *production rules* that are used to generate a morphological description string. The process of applying the rules is called *string-rewriting*. We start the string-rewriting with an initial string (the *axiom*). Each left-hand side of a rule found in the current string is rewritten (replaced) by the right-hand side. For example, if we first have an initial string "F" and we apply the rule

$$F \rightarrow F\text{-}F{+}{+}F\text{-}F \tag{4.1}$$

we get the string "F-F++F-F". When we apply the rule again, we get "F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F". The rules are usually applied for some predefined number of iterations. Each string which has been generated in such manner must have a sound interpretation to be useful. The symbols $\{F, +, -\}$ can be used to direct the movement of a "turtle" on a surface. The turtle reads the string as a list of commands; "F" means that the turtle should walk forward, "-" means that it should turn left, and "+" right. If we attach a pen to the turtle, we get a drawing that is shown in Figure 4.2a. This is actually a fractal introduced by Koch in 1904. It is not yet very useful for modeling morphogenesis.

If we introduce two new symbols, "[" and "]", we can more easily get "organic"-looking drawings. The first symbol, "[", stores (pushes) the position and direction of the turtle on top of a stack. The second symbol, "]", restores (pops) the topmost stored position from the stack. We can now make grammars such as

$$\begin{aligned} X &\rightarrow F[-[[X] + X] + F[+FX] - X] \\ F &\rightarrow FF. \end{aligned} \tag{4.2}$$

The result of this grammar can be seen in Figure 4.2b. It is clearly very plant-like. The storing-operation simulates the splitting-operation of cells (or plant branches) in biological organisms.

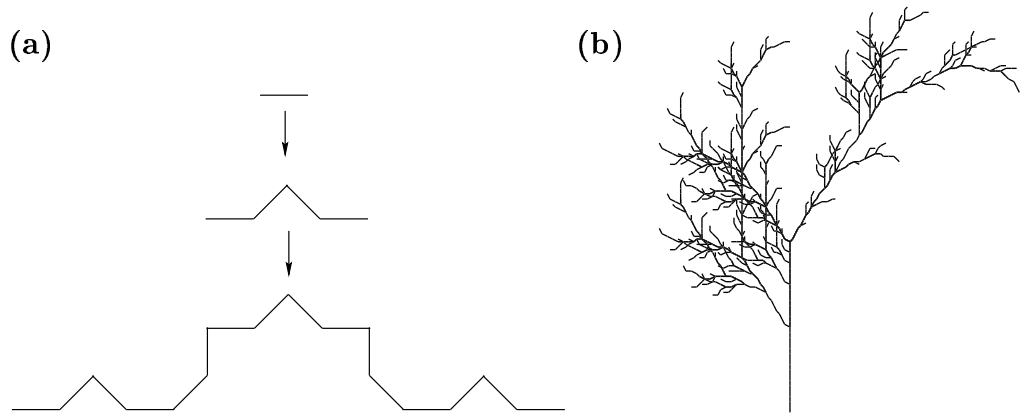

Figure 4.2: (**a**) Grammar 4.1 (Koch-fractal) with two iteration steps. (**b**) Grammar 4.2 drawn with branching angle 33° for 5 iteration steps.

Dawkins (1986) has used a similar model to illustrate the evolution of morphology. He used nine cell division parameters to build graphical 'organisms' called *morphs*. His rules were just division angles and line lengths, resulting in drawings visually similar to L-systems.

Lindenmayer's original purpose for the L-systems was the creation of *forms* of plants, but creating something that looks like a plant doesn't mean that it was constructed in the same way the plant actually grows. Biological trees are grown by dividing cells, not with turtles, stack operations or grammars. The string-rewriting systems are, however, temptative idealizations for these processes.

Biological organisms definitely do not contain production rules; they contain genes which are written in the DNA. The process that actualizes the rules written in DNA involves mechanisms that decode the DNA sequences into proteins. This involves submechanisms like gene activation, transcription of DNA to mRNA and translation of mRNA to proteins. The genes are not decoded simultaneously, but only a fraction of the genes are active at a given time. The mechanism that controls this process is called *genetic regulation*.

String-rewriting systems have been used as an idealization of genetic regulation and cell differentation in many studies dealing with evolutionary neural networks in the last few years. Some approaches have used *genetic regulation networks* for this idealization instead of grammars, some of which are reviewed later.

4.3 Indirect encodings

The trend in encoding the ANN topologies has been towards indirect (or weak) encodings. They are expected to yield better scalability and better development of modular hierarchy.

4.3.1 Kitano

The *graph generation grammar* developed by Hiroaki Kitano (1990b, 1990a) is an early grammar encoding method based on context-free and deterministic L-systems. The grammar contains production rules of the form

$$S \rightarrow \begin{bmatrix} A & B \\ C & D \end{bmatrix} \tag{4.3}$$

where the left-hand-side (LHS) of the production is a symbol and the right-hand-side (RHS) is a 2×2 matrix of symbols from the alphabet $\{A, B, ..., Z, a, b, ..., p, 0, 1\}$. The grammar is divided into two parts: the *variable part*, and the *constant part*. The variable part is the one encoded in the genome and its LHS symbols are from alphabet $\{A, B, ..., Z\}$. The start symbol "S" is guaranteed to exist as the LHS of the first rule in the genome. The constant part contains 16 rules for the left-hand-sides $\{a, b, ..., p\}$, which correspond to the 16 possible 2×2-matrices of ones and zeros on the RHS matrix. The ones and zeros are rewritten with

$$\begin{aligned} 0 &\rightarrow \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \text{ and} \\ 1 &\rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}. \end{aligned} \tag{4.4}$$

It is common during the rewriting process that there exists no rule in the grammar for a certain symbol ($\{A, ..., Z\}$) in the string. Those symbols are interpreted as "dead", and are rewritten using the same rule as for zeros. If there are two rules with the same LHS in the genome, the first one is used. The rewriting is terminated after a predetermined number of iterations. All non-numeric symbols still present in the matrix are replaced with zeros.

The diagonal bit of the final matrix encodes the presense or absense of a specific unit, as it did in the encoding by Miller et al. To make the network topology feedforward, any feedback connections are removed from the connection matrix. The decoding process is illustrated in Figure 4.3.

Kitano used fitness-proportional selection, elitism, single and multiple crossover and mutations. Mutations were done by changing a symbol in the chromosome randomly to some other with an adaptive rate varying from 2% to 30%, depending on the Hamming distance (number of mismatches) between the two parents. High distance resulted in low mutation rate and vice versa (according to Mitchell, 1996).

Kitano compared his method to that by Miller et al. with 4-x-4 and 8-x-8 decoding problems (see Chapter 5), which are traditional testbenches in the study of neural learning. He trained the networks using the standard backpropagation algorithm. The results suggested somewhat faster convergence and better scaling properties for the grammar encoding. He also evaluated the scalability of the method by trying values of 5, 10, 20, and 40 as the number of the variable production rules encoded in the genomes. The longest of the genomes gave the best results at the end of the evolution runs.

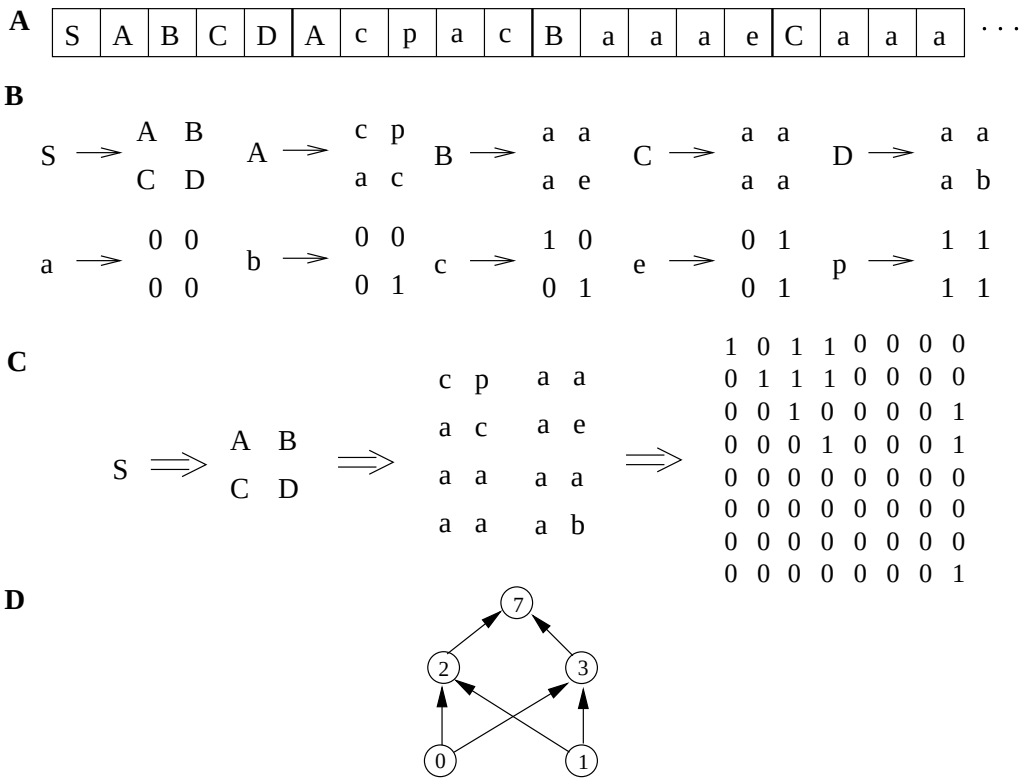

Figure 4.3: Decoding a grammar (example from Kitano, 1990). (**a**) Chromosome that encodes a grammar. (**b**) The production rules. (**c**) Applying the grammar to form a boolean (Miller) matrix. (**d**) The resulting network

Implementation details

We encoded, in our experiments, 64 variable production rules in the genome of each individual, with 26 variable LHS symbols. As with the method by Miller et al., the output units were always enabled in the connection matrix.

Discussion

We are interested in what networks generated from random genomes are like. Let us assume that we have n_V variable symbols (26 by default), n_C constant symbols (16 when the RHS matrix has size 2×2) and n_r rules encoded in the genome. The propability that at least one matching rule will be found is

$$p_{rule} = 1 - (1 - \frac{1}{n_V})^{n_r} \tag{4.5}$$

This gives about 71% For $n_r = 32$ rules and about 92% for $n_r = 64$ rules. Note that the start symbol is guaranteed to exist in the genome, so the above equation is not accurate. The propability that a given symbol is a constant one is clearly $p_C = \frac{n_C}{n_C + n_V}$

iterations	# cells	% fixed	% missing	% total fixed	% zeros	% ones
0	1	0	0	0	0	0
1	2	38.1	0	38.1	0	0
2	4	61.68	5.03	66.71	24.08	19.05
3	8	79.39	2.71	82.1	36.06	25.81
4	16	88.92	1.45	90.37	42.5	31.96
5	32	94.04	0.78	94.82	45.97	35.27
6	64	96.79	0.42	97.22	47.83	37.05

Table 4.1: Expected fixation of elements in the decoding matrix with the encoding method by Kitano et al. These values apply when there are 64 rules encoded in the genome.

and expectation is $e_C = 4p_C$ or about 1.52 constant symbols per rule for our default values (above).

We can now calculate the propability that a certain symbol is fixed ($\{a...p, 0, 1\}$) at ith iteration ($i > 1$) with recursive algorithm

$$\begin{aligned} p_\varphi(i) &:= 1 - (1 - p_\Phi(i-1)) \cdot (1 - p_C) \\ p_\emptyset(i) &:= (1 - p_{rule}) \cdot (1 - p_\Phi(i-1)) \\ p_\Phi(i) &:= p_\varphi(i) + p_\emptyset(i) \\ p_0(i) &:= p_\emptyset(i) + 0.5 \cdot p_\Phi(i-1) \\ p_1(i) &:= 0.5 \cdot p_\varphi(i-1) - \sum_{j=0}^{i-1} p_\emptyset(j), \end{aligned} \tag{4.6}$$

where the propabilities are: the propability p_φ of a fixed symbol, $p_\emptyset$ of a missing rule, p_Φ of total fixed symbols (also from the missing rules), p_0 of zeros, and p_1 of ones, respectively. The initial matrix has

$$p_\varphi(0) = p_\emptyset(0) = p_\Phi(0) = p_0(0) = p_1(0) = 0. \tag{4.7}$$

Also, because the first rule is guaranteed to exist, $p_\emptyset(1) = 0$ at the first iteration. The number of zeros given by these equations is the number of fixed zeros; remember that after the rewriting has finished all non-numeric symbols are replaced with zeros. The propabilities for $n_r = 64$ are given in Table 4.1. This tells us that the initial connectedness is somewhat, but not significantly, lower than with the encoding method by Miller et al. where the propability of ones and zeros is equal. This should reflect to the number of lethal networks in the initial population. However, we must note that the above calculations give just the average propabilities, not their variance. It is easy to see that the grammar encoding produces extremely connected or sparse initial networks much more probably than the direct encoding, as there is high correlation between the neighbouring elements of the connection matrix. The average propabilities given by the calculations should therefore be adjusted by the size of the initial population and the variance.

Siddiqi and Lucas (1998) have also made a comparison between the direct and graph grammar encodings, using the same encoder problem (see Chapter 5) as Kitano used in

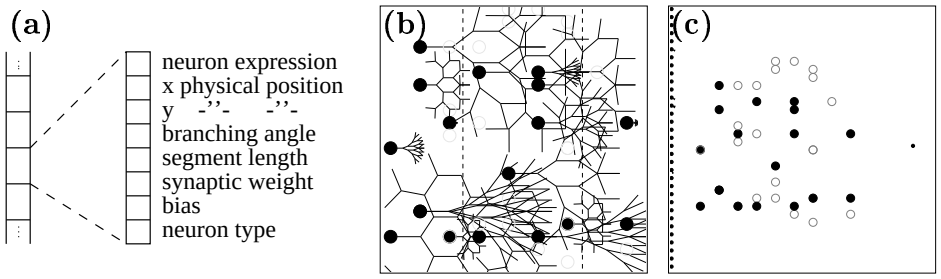

Figure 4.4: Construction of a Nolfi and Parisi network. **(a)** Neuron attributes are read from the genome. **(b)** Neurons are positioned in two-dimensional space and axon trees sprout from the neurons. **(c)** Connections to target neurons are established.

his comparison (1990a). They claimed that the comparison made by Kitano was unfair for the direct encoding because Kitano used propability $p_c = 0.3$ that a given connection would be enabled in the direct encoding. They found that the direct encoding performed much better with higher propabilities, for example with $p_c = 0.7$. We would like to see, after considering this with our analysis above, that Kitano's comparison was not so unfair after all, since the connection propabilities for his encoding were also quite low. Also, this optimal connectivity may depend on the problem, so the better propabilities found by Siddiqi and Lucas may not hold for other problems. It is a very good starting-point in the encoder problem that all the inputs are enabled and connected to all the outputs, which would be achieved with $p_c = 1.0$. That directly gives a working solution that can then be pruned to learn a bit faster. Their notion does, however, bring forth one problem: it is not as easy to adjust the connection propability for the grammar encoding as it is for the direct encoding.

4.3.2 Nolfi and Parisi

In the earliest model presented by Nolfi and Parisi (1992, 1994), the neurons are encoded with coordinates in a two-dimensional space. The mapping from genes to neurons is direct in a sense, but the connections are grown in a special manner. The phases of the decoding process are illustrated in Figure 4.4. Those neurons which fall into the left part of the space are considered as input units and those that fall into the right part are considered as output units. The connections are determined by letting "axon trees" grow forward from neurons. This is illustrated in Figure 4.4b. The trees are basicly L-system fractals generated from the grammar

$$F \rightarrow F[-F][+F] \tag{4.8}$$

with five iterations. The segment length and the branching angle of the fractal are individually encoded for each neuron. A connection is made wherever an axon branch touches another neuron.

The neuron index value i of an input our output unit u_i of the network is determined by the *type* gene. No separate training algorithms was used by Nolfi and Parisi. They

Gene name	gene type	length (bits)	minimum	maximum
expression	boolean	1	-	
coord.x	bit-float	5	0	8
coord.y	bit-float	5	0	20
bias	bit-float	10	-1	1
weight	bit-float	10	-1	1
seglength	bit-float	4	0	1
segangle	bit-float	6	-1	1
type	bit-int	4	0	15

Table 4.2: Genomic representation for Nolfi and Parisi encoding

used the neural network for the study of artificial life, concerning mostly the search of "food" and "water" sources by an autonomous artificial animal. They have also made some observations on the pre-adaptation phenomena exhibited in their model as expected in some punctuated equilibria models of the biological evolution.

Implementation details

The genomic representation used in our implementation is given in Table 4.2. Note that we ignore the weights and biases since we use the neural learning method.

Hidden neurons are indexed according to their x-coordinate. If two neurons have the same x-coordinate, their order in the array of neurons is determined by their order in the genome. Input and output units are indexed according to the *type* gene. For input units the index i is $i = type \ \mathbf{mod} \ N_\iota$ and for output units $i = N - N_o + (type \ \mathbf{mod} \ N_o)$, where N_ι is the number of inputs, N_o the number of outputs, and N the total number of neurons. Using this indexing, several neurons can clearly have the same index. Therefore, the connection of input and output units is handled in a special manner for this encoding method. We add "real" input and output layers where each "real" input unit is connected to every encoded input unit of the corresponding type, and likewise for the output units.

The axons are grown from initial symbol "X" by grammar $\{X \rightarrow F[-X][+X]\}$ for five iterations, and after the last iteration by $\{X \rightarrow F\}$. This is equivalent to the Grammar 4.8, but it goes through each path only once (yes, the earlier one doesn't). A connection is made when the axon tip touches another neuron. We discard feedback connections according to the index values of the neurons. A new (global) parameter was introduced: connectivity radius, r_c, given as the distance from the axon tip to the center of the target neuron. Nolfi et al. apparently used value $r_c = 0.5$, We estimated that value $r_c = 2.0$ would give a reasonable amount of lethal networks for the problems used in this study.

4.3.3 Cangelosi, Parisi and Nolfi

The method by Nolfi and Parisi was further developed by Cangelosi, Parisi and Nolfi (1993). They added cell division and migration rules to *grow* the neuron population

instead of encoding each neuron directly. This method is again much like an L-system. Organization of the genome is illustrated in Figure 4.5.

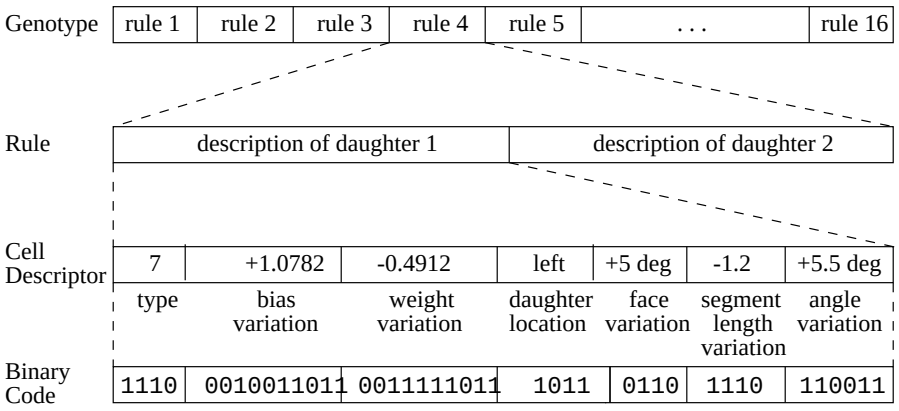

Figure 4.5: Rules contained in the genome

The ontogenesis is started with one "egg" cell with a specific start cell type. The cell is split (rewritten) into two daughter cells with the types and modifications to attributes given by the applied rule. Some cell types may imply programmed cell death. The daughter cells are positioned in the 8 possible locations in the substrate grid around the mother cell. The division is repeated for some number of iterations, after which the cells "maturate" as neurons. The neurons specialize as input or output neurons with the same spatial rule as in Nolfi and Parisi's model and their exact identity is determined by the final cell type. The "face" gene is a new attribute which controls the direction where the axon grows (from the 8 possible directions or "faces").

Just as Nolfi and Parisi, Cangelosi et al. used the method to evolve control networks for artificial lifeforms.

Implementation details

The rewriting of the cells was iterated for four to six times, depending on the problem. We did not use the *face*, nor the *weight* and *bias* genes. Otherwise, the implementation was just like for the method by Nolfi et al. above.

4.3.4 Other approaches

Boers and Kuiper

Boers and Kuiper (1992) have used a version of L-systems to grow the networks. They used a *context-sensitive* L-system to rewrite neurons and modules of neurons. The neurons are marked with symbols $\{A...H\}$. Modules are denoted with brackets such as "[ABAG]", and they are handled exactly as individual neurons are. Each neuron/module is by default connected to the next adjacent neuron/module. Missing connections are denoted by

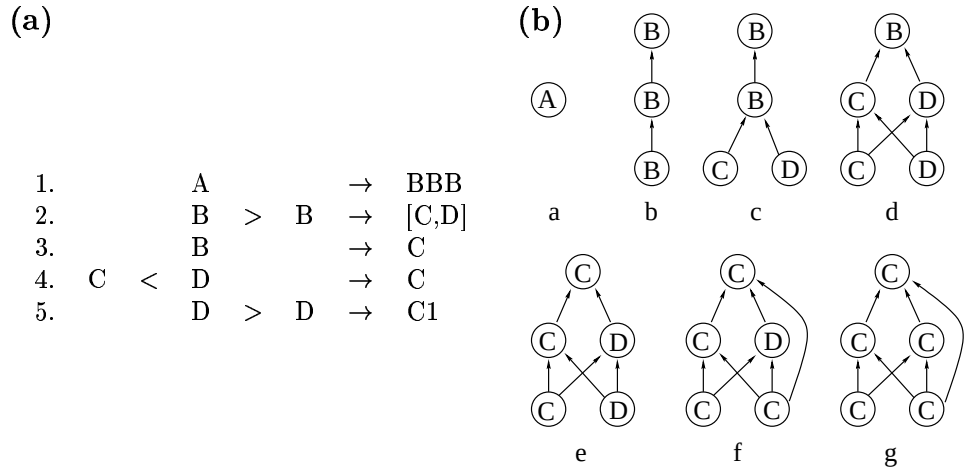

Figure 4.6: (**a**) Sample production rules in the L-system by Boers and Kuiper. (**b**) Rewriting an initial neuron "A" with the the rules, resulting as the network "[C,C1][C,C]C".

comma, such as "A,B". Additional connections are denoted with numbers that tell how many neurons a connection skips; for example, "A3AAAA" describes a chain of neurons, where the first neuron is connected also to the fourth neuron. We apply a rule to a neuron that matches the primary symbol in the left-hand-side of the rule *and* the *context*, if the rule has such. The context is denoted by $<$ and $>$ on the left-hand-side of a rule. To apply a rule with a context, it must also match the input neurons to the primary neuron for the left context ($<$), and the output neurons for the right context ($>$). The rewriting process is illustrated in Figure 4.6.

The production rules are encoded in the genome as bit strings. The strings are read in bit triplets (much like codons in biological chromatides), which are interpreted as symbols in alphabet $\{A...G, 1...5, [,], *\} \cup \{, \}$ (the * is a separator symbol) using a translation table (much like the codons are translated as amino acids). The symbol strings so acquired are interpreted as production rules. To obtain all the rules in the grammar, the triplets are read from the string in three different bit phases and to both directions.

Cangelosi and Elman

The encoding method by Cangelosi and Elman (1995) is continuation of the method by Cangelosi et al. compared in this study. It uses a genetic regulation network (GRN) to control the ontogenetic process. Their GRN consists of 26 regulatory elements (genes), divided into *receptors* of extracellular signals, *structural elements* that control the execution of developmental events, and *regulatory elements* that regulate the gene expression. The cells grow in a 7×20 space equivalent to that in the earlier model. As an example of the control process, element `DUP_Tim` acts as a timing signal for cell division. When that occurs, the placing of the daughter cells is determined by the physical environment around the mother, and by the amount of the `DUP_Tim` and `DUP_Pos` elements.

Axon growth and synaptogenesis are controlled by axonal growth factors, as well as extracellular signals emitted by the surrounding cells.

The GRN used by Cangelosi and Elman is modeled after the biological *operon* model of genetic regulation. It is encoded as a bit string, where each of the 26 genes is represented as an "operon" sequence of 28 bits. The operon is divided into two parts: *regulatory* (16 bits) and *expression* part (12 bits). The regulatory part consists of two regions: *inductor* (8 bits) and *inhibitory* region (8 bits). The expression part is also divided into two regions: *regulation* (8 bits) and *structural* region (4 bits). If a regulation region of an expressed gene matches the inductor region of another gene, the latter gene becomes expressed, unless it is inhibited by a regulation region of another gene matching the inhibitory region. The amount of expression (or "amount of the chemical element") depends on the amount of the inductory regulation element.

Dellaert and Beer

The cellular growth model of Dellaert and Beer (1994) uses a genetic regulation network, inspired by Kauffman's (1993) Random Boolean Networks, to simultaneously build the neural network and the physical morphology of an artificial organism.

They used a regulation network to control the division, differentation and apoptosis (intentional death) of the cells. Their approach is biologically plausible and similar to the method by Cangelosi et al. Instead of growing in a "space", the mother cell divides alternating horizontally and vertically for some iterations, thus forming a $N \times N$ grid. For growing the connections Dellaert and Beer have used a complex and a simplified model. In the complex method the axons grow by "sniffing" for guiding "chemicals" emitted by cells in the grid. Only cells that express appropriate genes can act as a sender or a receiver of a connection. Some cells in the grid contain "Cellular Adhesion Molecules" (CAMs) that allow axonal growth over them. The axon starts from a source cell, sniffs around to find which of the neighbouring cells has most the appropriate CAM, and then grows in that direction. A connection is made when the growing axon encounters a cell emitting *trophic* chemical factor. All the axons are grown simultaneously until the trophic factor is exhausted from the target cells. Connection strengths are determined by the amount of trophic factor present at the target cell after the connection was made. The simplified model connects every unit expressing the *axon* gene to every unit expressing the target gene within a range.

Harp, Samad and Guha

A network produced by the encoding method by Harp, Samad and Guha (1989) consists of a set of *areas*, which are one to three-dimensional blocks of neurons. Connections are encoded as projections between the areas. The genomic structure of the encoding is illustrated in Figure 4.7. Each area is encoded as a binary sequence in the genome, starting from a specific start-marker and ending to an end-marker. Harp et al. did not actually encode the markers, so they are not affected by mutations. Each area has an *area parameter specification* that describes its attributes. Harp et al. used 3 bits (8 values) for each attribute. The area ID serves as a name for the area. It is always 0 for the input area and 7 for the output area. The *total size* encodes the number of cells in

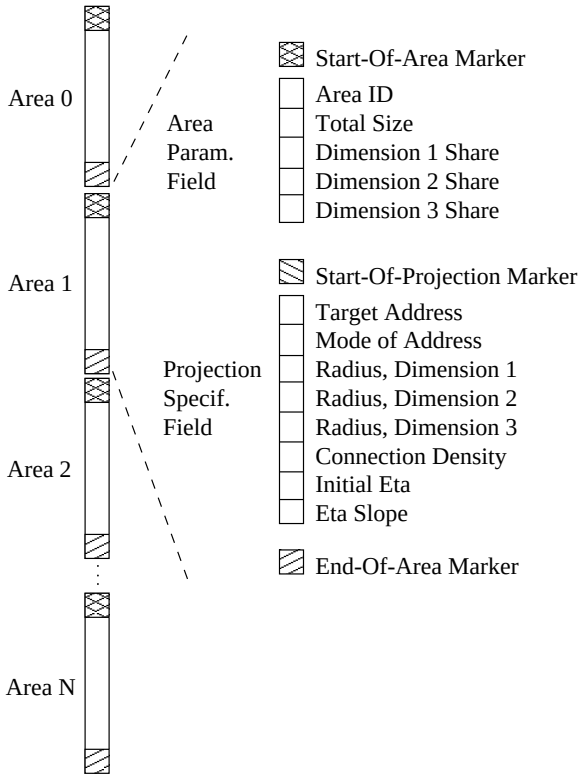

Figure 4.7: Genomic structure in the encoding method by Harp, Samad and Guha.

the area as a power of two (for example, 5 gives 32 cells). The proportions of the three dimensions are encoded as *shares*; the decoding algorithm finds dimensions whose product is equal to the total size, while conforming closely to the indicated shares. The connections between the areas are described with *projection specification fields* within the area gene structure. The target address is the absolute (area ID) or relative (area index forward in the genome) address of the target area, depending on the addressing mode. A binary gene tells which addressing mode to use. The connections are made only partially between the areas, specified by the radius parameters. The connection density parameter specifies the degree of connectivity between 30% and 100%, which adds an undeterministic element to the construction process. Two learning parameters of the backpropagation algorithm, the initial learning speed (eta) and its exponential decay factor, were also encoded by Harp et al.

This encoding method scales well, if mutations that insert and delete area and projection genes are added.

Gruau

Gruau's (1994) *cellular encoding method* uses a *grammar tree* to encode a cellular developmental process to grow neural networks. The decoding starts from a network with a single hidden "cell" that is connected to all input and output neurons of the network. The cell

starts reading the grammar tree from its root. The nodes of the tree are instructions that control how the cell is divided, etc. The child cells of a division differentiate by moving their "read-heads" to different branch of the grammar tree. Typical instructions are

- *sequential division* (`SEQ`), which results in two cells A and B where A receives all the inputs to the original cell, and is connected to B, which is connected to all the outputs of the original cell.
- *parallel division* (`PAR`), where both child cells inherit the input and output connections from the parent cell.
- instruction to *end reading* (`END`), which stops growing the growth for the particular cell.
- (unary) instructions for modifying cell's internal registers, such as bias, etc. The registers include connection registers, which point to specific connections to-and-from the cell.
- (unary) instructions for modifying the weight of a connection pointed by a connection register, or cutting the connection altogether.

The growth stops when all cells have read the `END` instruction.

Gruau implemented the genetic operations (recombination and mutation) in a way similar to the Genetic Programming paradigm developed by Koza (1990, 1992). Instructions (nodes of the tree) are mutated to other instructions with the same arity. The recombination operator replaces a sub-tree from one parent with a sub-tree from the other parent.

Vaario

Vaario's (1993) model was also inspired by Lindermayer's systems. Cells are grown in a two-dimensional space. Sensor neurons are on the other side of the space and actuator neurons on the other. The production rules of the model control various processes, such as cell division, cell death, axon and dendrite growth, etc. Axons and dendrites are grown according to gradients of chemicals emitted by target cells. They bounce against obstacles such as the borders of the substrate and form a connection when they reach another neuron. Connections unable to find a target neuron gradually withdraw.

Vaario has used the model for controlling the behaviour of an artificial lifeform. The model does not use learning during the lifetime of the organism. It uses a symbolic representation for the genome.

Chapter 5

Description of data

We compare the evolutionary neural network methods by their performance with four kinds of problems:

1. Two simple artificial toy problems, the XOR (parity) problem and the N-X-N encoding problem
2. Two more elaborate artificial function approximation problems
3. Three problems from the PROBEN1 (Prechelt 1994) collection of problems for benchmarking in neural network learning
4. The Bankruptcy dataset used by Back et al. (1997)

5.1 Artificial problems

We validated our implementations of the methods by Miller and Kitano with two very traditional test problems; the XOR and Encoder.

5.1.1 `XOR` problem

The XOR (parity) problem is a classical problem to test the capability of the network for performing nonlinear separation, see Table 5.1. It has been shown that this problem cannot be learned by a singel-layer perceptron topology. The minimal network consists of one hidden unit and five connections in all.

5.1.2 `Encoder` problem

The second toy problem we use is the N-X-N encoding problem, which is classical in benchmarking neural leaning algorithms. The task is to input a pattern that has one input as one and others as zero, and receive that same pattern at the output layer (the training set contains all the N permutations). If $X < N$, the network must compress

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 5.1: The XOR problem

(encode) the pattern somehow, and then decode it again at the output layer. For example, a network with 8 input and output units must have at least three hidden units to be able to succeed in this task; training such a network is called 8-3-8 encoding problem. In Kitano's tests (Kitano 1990a), the X was a free variable, as it is determined by the evolutionary search for the optimal network in respect to error after a number of neural training cycles. Kitano tested his graph grammar encoding method with both 4-X-4 and 8-X-8 encoding problems, using the grammatic encoding to construct networks with total maximum of 32 or 64 units. Clearly, no hidden units are required at all, but the connections can be made directly from the input to the output layer. This makes the encoding problem rather mundane.

In contrast to the other problems, we do not use early stopping for these two problems, but use the MSE with the training set as the fitness value.

5.1.3 `Additive` data

The `additive` data (Friedman 1991) has been generated using a function that has a nonlinear additive dependence on the first two variables, a linear dependence on the next three and is independent of the last five (pure noise) variables. We use a function

$$f(x) = \left(0.1e^{4x_1} + \frac{4}{1+e^{-20(x_2-1/2)}}\right) + \left(3x_3 + 2x_4 + x_5\right) + \left(0 \cdot \sum_{i=6}^{10} x_i\right) \tag{5.1}$$

presented by Friedman. The second term is clearly the sigmoid function, so it can be modeled to arbitrary accuracy with just one neuron. The problem data used in our tests contains $N = 50$ ten-dimensional covariate vectors generated in the unit hypercube. We assign the corresponding response values according to

$$y_i = f(x_i) + \varepsilon_i,\ 1 \leq i \leq N, \tag{5.2}$$

with the $\varepsilon_i = N(0,1)$ and $f(x)$ given by (5.1). It should be noted that since the outputs can have values outside range $[0,1]$, the output units of the neural networks can not use the standard sigmoid threshold function, but most preferably a linear one. This holds also for the `Interaction` data below.

5.1.4 `Interaction` data

The `Interaction` data is similar to the `Additive` data, also presented by Friedman (1991). The dataset was generated using a function that has a two-variable nonlinear

depence, a one-variable nonlinear dependence, two linear dependences and is independent of the last five (pure noise) variables.

$$f(x) = \Big(10sin(\pi x_1 x_2) + 20(x_3 - 1/2)^2\Big) + \Big(10x_4 + 5x_5\Big) + \left(0 \cdot \sum_{i=6}^{10} x_i\right). \qquad (5.3)$$

The covariates were generated randomly from a uniform distribution. The responses are assigned using (5.2) with $f(x)$ given by (5.3) and with ε being a standard normal deviate. The dataset used in our tests contains $N = 200$ samples.

5.2 PROBEN1 benchmarking problems

PROBEN1 (Prechelt 1994) is a collection of problems for neural network learning in the realm of pattern classification and function approximation plus a set of rules and conventions for carrying out benchmark tests with these or similar problems. PROBEN1 contains 15 datasets from 12 different domains (see Table 5.2). Three of these problems were used in this study.

Problem	Inputs				Outputs	Samples	Class propabilities
	b	c	m	tot.			
`cancer`	0	9	0	9	2	699	65.5% c_0 (negative)
`glass`	0	9	0	9	6	214	see below
`heart`	18	6	11	35	2	920	45% c_0 (negative)

Table 5.2: Attribute structure of the PROBEN1 problems. The number of input units is given for binary (b) and continuous (c) inputs and binary indicators (m) for missing values. (Continuous means more than two different ordered values)

`Cancer` data[1] deals with the diagnosis of breast cancer and the task is to classify tumors as either benign or malignant based on cell descriptions gathered by microscopic examination. The variables are described in Table 5.3.

`Glass` data deals with classification of glass types. All inputs are continuous and two of them have hardly any correlation with the result. The sizes of the 6 classes are 70, 76, 17, 13, 9 and 29 samples, respectively. The variables are described in Table 5.4.

`Heart` data[2] is used to predict heart disease. The binary decision is based on various personal data. Most of the attributes have missing values, some quite many: for attributes

[1] Original source: University of Wisconsin Hospitals, Madison; Dr. William H.Wolberg

[2] Original source: Hungarian Institute of Cardiology, Budabest; Andras Janosi, M.D., University Hospital, Zurich, Switzerland; William Steinbrunn, M.D., University Hospital, Basel, Switzerland; Matthias Pfisterer, M.D., V.A. Medical Center, Long Beach and Cleveland Clinic Foundation; Robert Detrano, M.D., Ph.D.

10, 12 and 11 there are 309, 486 and 611 values missing, respectively. Most other attributes have around 60 missing values. Additional boolean inputs are used to represent the 'missingness' of these values. The variables are described in Table 5.5.

5.3 Bankruptcy data

The bankruptcy data (`bankrupt`) consists of datasets one, two and three years before the bankruptcy. Each dataset contains training data with 400 cases and a test set with 170 cases. We use only the first of the three datasets in this study. The cases have been collected from 16775 American companies in Compustat Annual Industrial File database from 1985 to 1993. A deeper description of the data has been given by Back, Sere, and Laitinen (1997). Half of the cases are failing companies (class 1) and half non-failing (class 0). The 33 input variables selected by Back et al. (1997) are financial ratios that have been found to be useful in some previous bankruptcy prediction studies. The variables are described in Table 5.6.

The cases within the training data are randomly permutated and then ordered so that failing and non-failing companies alternate in the set. The training data is then divided into a neural training data and a fitness evaluation set in 70 : 30 ratio, as described below, and in Figure 5.1.

5.4 Handling the datasets

5.4.1 Division into subsets

We divide the classification datasets into four pattern sets. First is the actual *training set* that the neural learning algorithm uses to train the connection weights. The second is a *validation set*, used for early stopping during the neural learning. The third is an *evaluation set*. The evolutionary algorithm uses it for evaluating the fitness of the trained networks. We measure the fitness of the trained networks as MSE in respect to the evaluation set. The evaluation of the fitness can not be done with the training set, because we want to measure the generalization ability of the network, not just the learning ability. The fourth set is the *final test set*, which is used in final tests to benchmark the network topologies found by the evolutionary algorithm with the different encoding methods. The set that is used as the evaluation set during the evolution run, is used as the validation set in the final tests.

We formed the sets for the bankruptcy data by dividing the original dataset as illustrated in Figure 5.1. The PROBEN1 datasets are divided in a similar manner; for neural training data 50%, for evalution set 25%, and for final test set 25% of the available patterns. Prechelt (1998) made a search for a good size of the validation set with the PROBEN1 datasets. He received the best results with portions ranging from 20% to 35% of the entire training data. We use 20% of the neural training data as the validation set, and rest as the training set.

PROBEN1 contains three different permutations of each dataset. The sets can be partitioned in six additional combinations. We use the first permutation and partitioning, which is indicated in the dataset name with, for example, `cancer1a`.

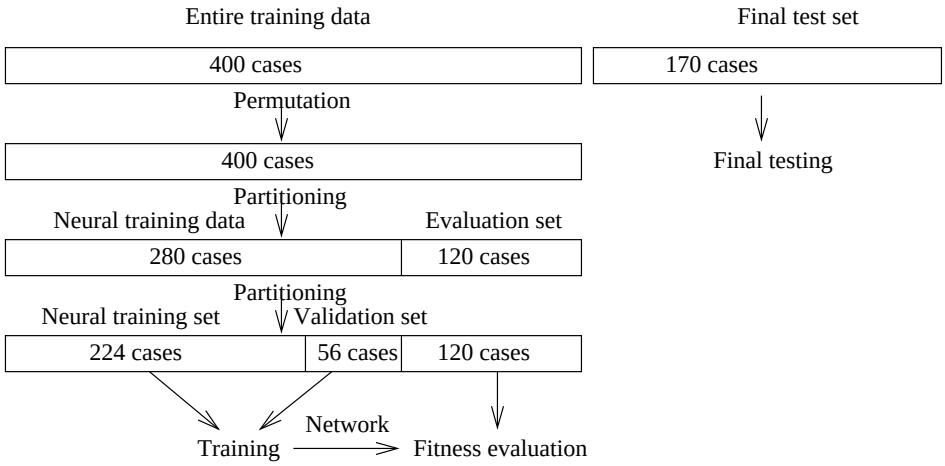

Figure 5.1: Division of the datasets. The numbers are for the bankruptcy data.

5.4.2 Equalization

The input vectors of the bankruptcy data were equalized using two methods: *histogram equalization* and *gaussian equalization* (see Appendix A). In both cases each of the vector components is equalized separately. The test sets were equalized using the equalization parameters from the corresponding training data. Note that this equalization applies only to the bankruptcy data; the datasets in PROBEN1 were equalized by Prechelt (1994) with linear (gaussian) equalization.

Missing values

There is no clearly best way to inform (most) artificial neural network models that some of the input vector components are missing. From the PROBEN1 datasets, such anomalies have either been removed or are indicated by a separate binary inputs. About 10% of the cases in the bankruptcy dataset contain missing values. After the equalization, we set the missing values to the average of the equalized sets: 0.5 for the histogram normalization and 0.0 for the gaussian normalization.

Output vectors

We scaled the the output vectors of the bankruptcy data so that $0.1 = class_0$ and $0.9 = class_1$. The reason behind this is that, if the output layer of the neural network uses the sigmoid function, its derivative in the limits 0 and 1 would be zero, which might cause problems in the training process.

Variable	Description	Range
I_1	Clump Thickness	1 - 10
I_2	Uniformity of Cell Size	1 - 10
I_3	Uniformity of Cell Shape	1 - 10
I_4	Marginal Adhesion	1 - 10
I_5	Single Epithelial Cell Size	1 - 10
I_6	Bare Nuclei	1 - 10
I_7	Bland Chromatin	1 - 10
I_8	Normal Nucleoli	1 - 10
I_9	Mitoses	1 - 10
O_1, O_2	Class (O_1: benign, O_2: malignant)	

Table 5.3: Variables in the `cancer` data. The range refers to the range in the original data.

Variable	Chemical	Min	Max	Mean	SD	Correlation with class
I_1	RI	1.5112	1.5339	1.5184	0.0030	-0.1642
I_2	Na	10.73	17.38	13.4079	0.8166	0.5030
I_3	Mg	0	4.49	2.6845	1.4424	-0.7447
I_4	Al	0.29	3.5	1.4449	0.4993	0.5988
I_5	Si	69.81	75.41	72.6509	0.7745	0.1515
I_6	K	0	6.21	0.4971	0.6522	-0.0100
I_7	Ca	5.43	16.19	8.9570	1.4232	0.0007
I_8	Ba	0	3.15	0.1750	0.4972	0.5751
I_9	Fe	0	0.51	0.0570	0.0974	-0.1879

Output	Class name
O_1	Building windows, float processed
O_2	Building windows, non-float processed
O_3	Vehicle windows, float processed
O_4	Containers
O_5	Tableware
O_6	Headlamps

Table 5.4: Input and output variables in the `glass` data. The ranges and statistics refer to the original data.

Field	Input variable(s)	Description
0: age		
	I_1	1 continuous 28...77 $\rightarrow$ 0...1
1: sex		
	I_2	1 binary (0=female, 1=male)
2: chest pain type		
	$I_{3,4,5,6}$	4 nominal (3:typical angina, 4:atypical angina, 5:non-anginal pain, 6:asymptomatic)
	I_7	1 binary (0=attrpresent, 1=attrmissing)
3: resting blood pressure		
	I_8	1 continuous 80...200
	I_9	1 binary (0=attrpresent, 1=attrmissing)
4: serum cholestoral in mg/dl		
	I_{10}	1 continuous 85...603
	I_{11}	1 binary (0=attrpresent, 1=attrmissing)
5: fasting blood sugar		
	$I_{12,13}$	2 nominal (12: $> 120mg/dl$, 13: $\leq 120mg/dl$)
	I_{14}	1 binary (0=attrpresent, 1=attrmissing)
6: resting electrocardiographic results		
	$I_{15,16,17}$	3 nominal (15:normal, 16:ST-T wave abnormality, 17: left ventricular hypertrophy)
	I_{18}	1 binary (0=attrpresent, 1=attrmissing)
7: maximum heart rate achieved		
	I_{19}	1 continuous 60...202
	I_{20}	1 binary (0=attrpresent, 1=attrmissing)
8: exercise induced angina		
	$I_{21,22}$	2 nominal (21:yes, 22:no)
	I_{23}	1 binary (0=attrpresent, 1=attrmissing)
9: ST depression induced by exercise relative to rest		
	I_{24}	1 continuous -2.6...6.2
	I_{25}	1 binary (0=attrpresent, 1=attrmissing)
10: the slope of the peak exercise ST segment		
	$I_{26,27,28}$	3 nominal (26:upsloping, 27:flat, 28:downsloping)
	I_{29}	1 binary (0=attrpresent, 1=attrmissing)
11: number of major vessels (0-3) colored by flourosopy		
	I_{30}	1 continuous 0...3
	I_{31}	1 binary (0=attrpresent, 1=attrmissing)
12: thal		
	$I_{32,33,34}$	3 nominal (32:normal, 33:fixed defect, 34:reversable defect)
	I_{35}	1 binary (0=attrpresent, 1=attrmissing)

Table 5.5: Variables in the `heart` data. The *field* index refers to the field in the original data, while the *input variable* index refers to the preprocessed data.

Variable	Ratio	Type
1	Cash/Current Liabilities	L
2	Cash Flow/Current Liabilities	L
3	Cash Flow/Total Assets	L
4	Cash Flow/Total Debt	L
5	Cash/Net Sales	L
6	Cash/Total Assets	L
7	Current Assets/Current Liabilities	L
8	Current Assets/Net Sales	L
9	Current Assets/Total Assets	L
10	Current Liabilities/Equity	L
11	Ebit/Total Interest Payments	L
12	Equity / Net Sales	S
13	Inventory/Net Sales	L
14	Long Term Debt/Equity	S
15	Long Term Debt/Net Capital Employed	S
16	Market Value of Equity/Book Value of Debt	S
17	Net Income/Total Assets	P
18	Net Quick Assets/Inventory	L
19	Net Sales / Total Assets	P
20	Net Worth/Total Liabilities	S
21	Operating Income/Total Assets	P
22	Quick Assets/Current Liabilities	L
23	Quick Assets/Net Sales	L
24	Quick Assets/Total Assets	L
25	Rate of Return to Common Stock	P
26	Retained Earnings/Total Assets	P
27	Total Debt/Equity	S
28	Total Debt/Total Assets	S
29	Total Liabilities/Net Capital Employed	S
30	Working Capital/Net Sales	L
31	Working Capital/Net Worth	L
32	Working Capital/Total Assets	L
33	Earnings Before Interest and Taxes / Total Assets	P

Table 5.6: Financial ratios in the `bankrupt` dataset. The *type* tells that the ratio is a L=liquidity, P=profitability, or S=solidity indicator.

Chapter 6

Experiments and results

In this chapter, we first show the results for some preliminary calibration tests, and then discuss the results for the encoding methods.

6.1 Noisy fitness

We first analyzed the amount of noise in neural training due to randomness in initial connection weights. This was done to calibrate the GA for the development of neural network architectures. We then searched for good GA parameters on simple search problems while varying the amount of noise. The parameters that were optimal for the amount of noise in neural learning problems were adopted for later use.

It may seem a bit strange to add noise (random initialization of weights) to the evaluation of the individuals and then fuss about the problems it brings. The random initialization is, however, essential for neural learning algorithms. First of all, it is done to avoid problems due to symmetries in the network (Bishop 1995a, pp. 260-262). But, that doesn't yet explain why we use *different* random initialization every time we train the network. First of all, as mentioned in Chapter 2, the training error is the result of a complex function of topology, especially with deterministic learning algorithms; small changes in the topology can cause unpredictable changes in the training error. This means that there is "noise" anyways, and initializing the weights of the sligthly different networks randomly doesn't make the noise much worse. The second very important reason is that the random initialization gives the EA a statistically more valid view to the goodness of a topology than a single starting-point would give. This is especially important for the elites (and also for accidentally identical individuals), which "should not get a free lunch" just if a certain initialization happens to favor them. Instead, we want to evaluate them multiple times to make sure they really are good. The third reason is a technical one: determining which connections of similar networks are the "same" is usually highly ambiguous.

Archit.	Training set				Validation set				Final test set			
	classif. err.%		MSE		classif. err.%		MSE		classif. err.%		MSE	
	mean	σ	mean	σ	mean	σ	mean	σ	mean	σ	mean	σ
Prechelt	-	-	9.25	1.07	-	-	13.22	1.32	19.89	2.27	14.33	1.26
35-8-2 S	12.42	1.23	10.33	0.71	17.93	1.26	13.53	0.43	19.53	1.29	14.58	0.47
35-4-2 S	12.92	1.19	10.59	0.65	17.84	1.28	13.54	0.40	19.74	1.25	14.56	0.42
35-2-2	16.18	6.55	15.63	10.25	20.53	5.39	18.54	10.31	21.99	5.07	19.73	10.41

Table 6.1: Comparison of three hand-picked topologies with 1000 training-testing-runs with differing random network initialization. Validation error and MSE are given for each set. The first one, 35-8-2, is the same used by Prechelt in his sample tests (see the first row).

6.1.1 Amount of noise in training

The amount of noise in the fitness was estimated by training and testing the `heart1a` data 1000 times with three fixed neural topologies that were somewhat different from each other. The runs were done using RProp (presented in Appendix A) with $\Delta_{max} = 50.0$, $\Delta_{min} = 1e^{-6}$, $\eta^- = 0.5$ and $\eta^+ = 1.2$, standard sigmoid transfer function for hidden units, linear transfer function for outputs and GL_5 termination. Weights were initialized randomly to range $[-0.5, 0.5]$. The small, although statistically significant differences in results are probably due to minor differences in training parameters, such as the used sigmoid function, termination method, etc. A numeric comparison of the results is given in Table 6.1. This tells us that the signal-to-noise ratio might be somewhere between 1:1 and 1:100.

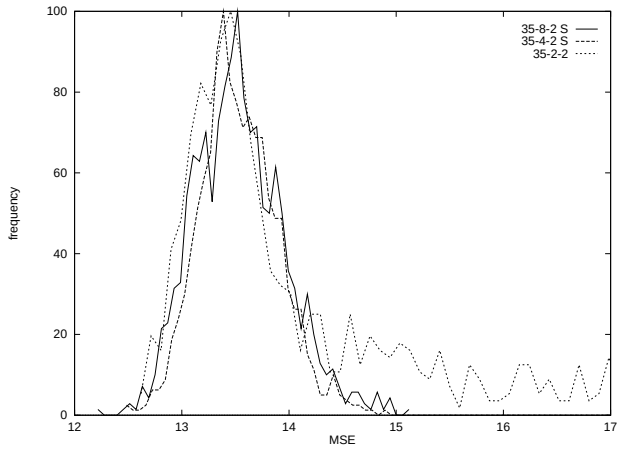

Figure 6.1: Distribution of validation MSEs for the `heart1a` data. The histograms were constructed according to the range of each test case separately, so their scaling varies. The resolution (number of slots) of the histogram was 1000, 100 and 50 for the sets 35-8-2S, 35-4-2S and 35-2-2, respectively. The frequency was scaled so that the maximum frequency for each histogram is 100. Note that the average MSE (given in Table 6.1) for the 35-2-2 network is still outside the MSE range of the graph, so that curve continues to the right.

A histogram reporting the distribution of validation MSEs is shown in Figure 6.1. The distribution of the smallest topology is particularly interesting, since it seems that it might actually yield better networks than the bigger ones, although its results are on

average much worse. Since we do not have any way of exploiting this behaviour, we ignore it and look only at the average values. Some discussion on this is given in Chapter 7.

It should be noted that these are just some hand-picked samples from the huge space of topologies and they may not be very representative. Taking this into account, these results can give only a very rough estimate.

The above results must also be kept in mind when analyzing the evolution graphs in Section 6.2., as the graphs show the MSE of the fittest individual of each generation. If one encoding method produces networks with a larger learning variance than the others, it may look better, although it is not.

6.1.2 Selection parameters

To calibrate the μ parameter (the number of potential parents) of the (μ, λ)-selection, we used a fixed population size of 20 individuals and evolved the GA with the sphere test function[1]. It is somewhat questionable how well this simple test function fits here, but we hope to get at least some direction from it.

Effects of noise on the sphere test function are visualized in Figure 6.2A-C. It seems that the optimum would be somewhere between $\mu = \{3..10\}$. Assuming that this scales to other population sizes linearly, we would have $\mu \approx 0.2 \cdot \lambda$. As any real search landscape can be expected to be much rougher, we should propably use somewhat weaker selection if we hope to find better solutions. Then again, many people have used even stronger selection for evolving neural networks. We noticed during the early runs on the artificial problems that, if the selection was too weak, the GA had great difficulties in getting rid of the fatal networks and good networks quite often vanished from the population. Some non-even selectivity between the parent candidates might have helped this problem, but we think that such weighting is equivalent to tightening the μ selection parameter. Therefore, we compromise to a smaller value of $\mu = 0.16 \cdot \lambda$ and hope that such a rough estimate suffices.

These results do not tell much about whether or not we should use elitism (non-re-evaluating elitism). As we were afraid it might cause some trouble such as that described in chapter 3, we were rather wary about using it. However, it was noticed in the early runs especially with the Kitano method that the propability of having a surviving individual in the starting population was somewhat low. If one good individual was found, it was usually lost (possibly forever) after the first reproduction if no elitism was used. Therefore it was decided to use weak elitism of one elite individual.

[1]Notice that the optimal number of parents for sphere function when there is no noise is $\mu = 1$! More accurately, the optimal selection strategy is $(1 + 1)$ (elitist population with one parent and one offspring, see Chapter 3 for further explanation of this selection mechanism).

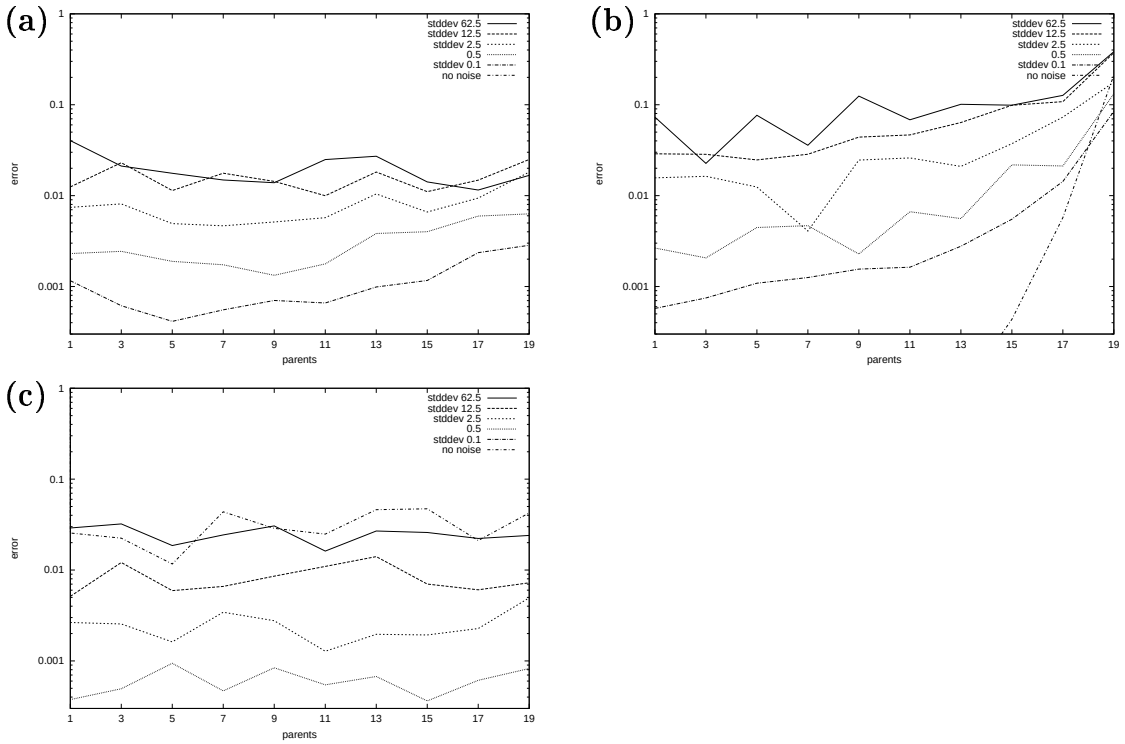

Figure 6.2: Effects of μ to the adaptation speed with the sphere test function (2-dimensional, 16-bit gray-encoded) with different levels of artificial normally distributed noise. **(a)** gives the results for $(\mu, 20)$ and **(b)** for elitist $(\mu + 20)$ selection with μ varying from 1 to 19 (with stepsize of 2) for 100 generations. **(c)** has also an elitist $(\mu + 20)$ selection, but we keep the number or evaluations constant (2000) so that the number of generations is $2000/(\lambda - \mu + 1)$ (we add $+1$ because of the re-evaluation of the king). The "no noise" curve is below the lower limit for A and C.

6.2 Evolving networks

Network topologies were adapted by evolutionary search for every problem dataset by all of the four encoding methods. Individual runs took between about 15 minutes and 70 hours on 167-296MHz Ultra Sparc workstations, depending on the problem and the encoding method. The runs with the encoding methods by Nolfi and Parisi, and Cangelosi et al. took so much time that we made only one run with each of the classification problems.

The parameters for the experiments are described below.

6.2.1 Neural learning parameters

Neural training was done using RProp during the evolutionary search and using back-propagation with momentum in final testing. The learning parameters α and η for back-propagation were set to fixed values 0.2 and 0.3, respectively. Weights were initialized to range $[-0.5, 0.5]$. GL_2 early stopping was used during evolution, and GL_5 in final testing.

6.2.2 Genetic algorithm parameters

Parameters common to all runs are given in Table 6.2. The global coefficients for mutation rates and variance are multipliers for the autoadapted rates. For example, the range for the actual p_{binary} is $0.01 \cdot [0.01, 1.0) = [0.0001, 0.01)$.

For the XOR and Encoder problems, the number of training cycles was set to a relatively small value of 50, because we wanted to avoid the RProp learning the problems too well, i.e., reaching the calculation precision limit, and therefore leaving no decent gradient for the GA to evolve on. For these two problems, we didn't need early stopping. The number of potential neurons was a power of two, and same for all the encoding methods. It was reasonably higher than the total number of inputs and outputs to leave room for hidden units. These problem-specific parameters are given in Table 6.3.

6.2.3 Results

The best network topologies of the final populations of each evolution run were trained and tested for 30 times with different connection weight initializations. The validation set used for early stopping in these final tests is the same set that was used for the fitness evaluation in the evolutionary learning phase. After training, the networks were tested with the final test sets. The results are given in Table 6.4.

The selections of the input variables, observable also in the network pictures in Figures 6.4 to 6.18, are summarized in Table 6.5.

6.2.4 Individual runs

The performance curves for the individual evolution runs are given in Figures 6.3 to 6.17. The plotted value is the MSE of the *fittest* individual of the population for each generation. The fittest networks at the final generation (100) of each evolution run are shown in Figures 6.4 to 6.18. The network drawing algorithm that we used in producing the network pictures is explained in Appendix A. For the methods by Nolfi and Parisi, and Cangelosi et al., also the internal networks are shown. The evolution log graphs and network pictures are accompanied by a short discussion of the results. The discussions combine the results from the tables and figures below.

Parameter	Value	Description
m_{binary}	0.01	Coefficient for the mutation propability of binary genes
$m_{integer}$	0.01	Coefficient for the mutation propability of integer genes
m_{real}	0.1	Coefficient for the mutation propability of real-valued genes
m_{σ}	0.1	Coefficient for the mutation variance of real-valued genes
$min(p_{mut})$	0.01	Lower bound for mutation rates.
$gens$	100	Number of generations evolved
λ	50	Population size
μ	8	Number of potential parents
ϵ	1	Number of elites in the population
p_{recomb}	0.5	Recombination frequency per top-level genetic container
$evaluations$	1	Number of training-testing evaluations for the networks
$cycles$	3000	Maximum number of training cycles
$terminator$	GL_2	The early stopping method
$term-part$	0.25	Portion of training samples used for early stopping
l_{strip}	10	Interval of validation tests for early stopping

Table 6.2: Generic parameters for the EA.

Problem	Neurons	Types	Training	Early stopping
`XOR`	16	2	50	no
`Encoder`	32	8	50	no
`Additive`	32	16	3000	yes
`Interaction`	32	16	3000	yes
`cancer1a`	32	16	3000	yes
`glass1a`	32	16	3000	yes
`heart1a`	64	64	3000	yes
`bankrupt`	64	64	3000	yes

Table 6.3: Problem-specific parameters. *Neurons* indicates to total number of genetically encoded neurons, including the input and output units. *Types* indicates the number of neuron types for the methods by Nolfi et al. and Cangelosi et al. *Training* is the maximum number of learning cycles with neural learning, unless it is stopped earlier by early stopping.

Problem	Method	Run	Train set				Validation set				Test set			
			clsf. err.%		MSE		clsf. err.%		MSE		clsf. err.%		MSE	
			mean	σ	mean	σ	mean	σ	mean	σ	mean	σ	mean	σ
cancer1a	miller	1	3.49	0.45	2.93	0.40	1.94	0.45	1.84	0.39	2.30	0.97	2.15	0.58
		2	3.07	0.38	2.88	0.83	1.66	0.30	1.74	0.78	2.18	0.68	2.16	0.90
		3	3.54	0.50	3.01	0.26	2.76	0.58	2.06	0.15	2.01	0.68	1.88	0.41
		4	3.98	0.77	3.18	0.54	1.98	0.32	1.94	0.35	2.16	0.94	2.08	0.66
	kitano	1	2.56	0.39	2.15	0.26	3.03	0.78	2.32	0.34	**1.99**	0.56	1.53	0.25
		2	2.74	0.27	2.20	0.24	3.24	0.77	2.32	0.26	**1.99**	0.60	1.58	0.23
		3	2.60	0.32	2.22	0.25	2.78	0.63	2.25	0.29	2.82	1.17	2.44	0.50
		4	2.58	0.39	2.18	0.32	2.95	0.78	2.24	0.29	2.28	0.64	1.70	0.31
	nolfi	1	3.80	0.41	3.44	0.22	1.73	0.42	1.76	0.23	2.32	0.54	2.06	0.29
	cangelosi	1	3.51	0.21	3.21	0.09	2.86	0.15	2.61	0.15	2.70	0.66	2.36	0.52
	prechelt	S	2.77	0.39	2.53	0.20	3.62	0.74	2.57	0.25	2.20	0.55	1.78	0.38
		NS	2.61	0.70	2.36	0.68	3.07	0.73	2.36	0.37	2.11	1.21	1.64	0.81
glass1a	miller	1	31.68	8.57	8.75	2.31	42.47	6.91	10.83	2.29	40.38	5.37	11.90	3.03
		2	36.60	8.71	8.54	1.66	39.63	7.71	9.70	1.33	48.05	5.27	11.77	2.30
		3	35.08	9.14	9.70	3.91	42.47	7.97	11.38	4.44	41.38	7.20	11.61	4.12
		4	33.33	8.54	7.99	1.26	38.21	6.47	9.59	0.75	38.18	9.13	9.94	0.85
	kitano	1	30.31	7.54	7.14	1.15	34.81	3.31	9.17	0.32	33.96	5.44	9.33	0.50
		2	29.50	7.54	6.98	1.10	35.12	2.64	9.28	0.26	33.96	5.49	9.47	0.42
		3	28.13	5.87	7.06	0.99	34.81	2.72	9.41	0.26	**32.08**	5.25	9.64	0.66
		4	31.18	5.35	7.28	0.93	34.38	3.57	8.87	0.25	33.84	4.18	9.32	0.37
	nolfi	1	37.85	14.63	10.93	4.66	43.40	12.24	12.10	4.56	49.69	9.37	14.71	5.90
	cangelosi	1	62.68	14.03	16.28	7.51	59.81	15.90	15.62	6.80	63.46	14.89	16.08	6.37
	prechelt	S	33.36	6.25	7.92	0.81	36.11	2.65	9.54	0.30	33.14	5.20	10.16	0.64
		NS	32.93	5.56	7.82	1.03	37.10	3.41	9.41	0.74	34.47	6.83	9.86	1.14
heart1a	miller	1	15.11	0.97	11.23	0.54	17.26	0.94	12.33	0.27	21.32	1.16	15.03	0.42
		2	15.28	0.82	11.92	0.84	18.64	1.14	13.33	0.84	20.57	1.02	15.26	0.91
		3	15.42	1.45	11.49	1.43	17.14	1.27	12.67	1.17	20.32	1.25	14.99	1.18
		4	17.76	5.90	14.58	5.79	18.84	5.52	15.19	5.34	22.36	4.09	17.18	5.06
	kitano	1	14.34	1.37	11.28	0.90	17.01	1.16	13.51	0.65	20.33	1.20	14.54	0.64
		2	14.86	0.31	11.55	0.38	17.49	0.49	13.35	0.37	**19.97**	0.43	14.37	0.33
		3	14.76	3.68	11.70	3.95	18.57	3.81	14.08	3.70	21.30	2.88	15.29	3.59
		4	13.25	0.97	10.23	0.58	19.91	1.20	13.32	0.30	20.51	1.12	14.80	0.39
	nolfi	1	22.70	9.64	18.48	10.83	24.52	9.40	19.63	10.72	26.84	7.10	21.78	9.28
	cangelosi	1	32.41	12.22	24.72	9.97	33.80	12.63	25.52	9.92	34.16	10.06	26.10	9.35
	prechelt	S	12.27	0.94	10.32	0.53	17.65	1.25	13.53	0.37	*19.41*	1.18	14.61	0.39
		NS	12.62	0.92	10.13	0.56	18.80	0.97	13.36	0.37	19.93	0.98	14.41	0.35
bankrupt	miller	1	19.66	2.21	14.97	1.43	23.28	2.41	16.03	0.93	23.24	2.29	17.24	1.21
		2	18.23	2.55	14.45	1.83	22.46	2.32	15.35	1.10	22.43	1.77	16.99	1.16
		3	19.34	1.43	15.09	1.09	23.66	1.74	15.70	0.42	23.24	1.77	16.75	0.52
		4	21.78	4.78	16.57	2.71	25.27	3.89	16.94	2.69	23.24	4.48	18.52	2.64
	kitano	1	18.10	1.43	13.98	0.91	24.28	1.66	16.35	0.44	22.94	1.33	16.69	0.42
		2	19.37	4.94	14.84	2.13	24.73	4.26	16.79	1.72	25.27	4.35	18.15	1.52
		3	18.82	2.04	14.28	1.34	23.98	2.13	16.79	1.23	23.61	2.57	17.40	1.09
		4	18.02	1.57	14.34	0.92	23.83	1.33	16.18	0.73	**19.82**	1.51	16.11	0.41
	nolfi	1	20.58	4.69	15.39	2.56	22.84	3.52	16.38	2.65	24.57	3.64	18.58	2.55
	cangelosi	1	22.14	3.67	16.22	1.85	23.46	2.56	16.22	1.52	23.94	3.25	17.13	1.59
	manual	S	19.50	4.06	14.71	2.51	25.90	1.95	18.19	1.75	26.08	3.32	19.35	2.51
		NS	20.76	3.67	15.36	1.97	25.32	2.57	17.69	1.62	24.63	2.86	17.49	1.96

Table 6.4: Classification test runs with the best neural topologies found by the evolutionary search runs. Results obtained from test runs with the best topologies found by Pretchelt, and with a similar (manual) search for the `bankrupt` data, are given for comparison ("S" denotes shortcuts and "NS" no shortcuts). Notice that the MSEs are given multiplied by 100, as they were given in the tests done by Prechelt.

Method	Run	Problem: Encoder	Additive	Interaction
Miller	1	●●●●●●●●	●●●●●○○○○○	●●●●●●○○○●
	2	●●●●●●●●	●●●●●○○○○●	●●●●●●○○●○
	3	●●●●●●●●	●●●●●○○○○●	●●●●●○○○●○
	4	●●●●●●●●	●●●●○○○○○○	●●●●●○○○○●
Kitano	1	●●●●●●●●	●●●●●●●●○○	●●●●●●●●○○
	2	●●●●●●●●	●●●●●●●●○○	●●●●●●●●●
	3	●●●●●●●●	●●●●●●●●○○	●●●●●●●●○○
	4	●●●●●●●●	●●●●●●●●○○	●●●●●●●●○○
Nolfi	1	●○●●●●○○	●●●●●○○○●○	●●●●●○○○○○
	2	●●●○○●●●	●●○●●○○○○○	●●●●●○○○○○
	3	●●●●●●●●	●●●●●○●○○○	●●●●●○○○○○
	4	●●●●●●●●	●●○●○○○○○○	●●●●●○○○○○
Cangelosi	1	●●●○○○○○	●●○●○○○○○○	○●○●●○○○●○
	2	○●●○○○○●	●●○●●○○○○○	●●○●●○○○○○
	3	●○○●●●○○	●●○●○○○○○○	●●○●●○○○○○
	4	○○●○○●○●	●●○●○○○○○○	●●●●●○○○○○

Method	Run	Problem: Cancer	Heart
Miller	1	●○●○●●●●○	●●○●●○○○○●●○●●●○○○●●●○○●○○●●○●●●○○●
	2	●○●○●●○●●	●●○○●●○○○●●○○●○○○○○○●○○●○●●○○●○●○●○
	3	●●●○●●●○●	●●○●●○●●○○●●●●○○○○○○●●○○○●○●●●●○●○○○
	4	●○○●○●●●○	●●○●●○●●●●●●○○○○●●●○○●○○●○●○●●●●○●●○
Kitano	1	●●●●●●●●	●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●○●○
	2	●●●●●●●●	●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●○○○
	3	●○●●●●●●○	●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●
	4	●●●●●●●●	●●●●●●●●●●●●●●●●●○○○○○○○○●●●●●●●●●●●
Nolfi	1	●○○○○●○●○	●●○●○●○○○○●●○○○○●○○○○○○○○●●●○○○●○○●
Cangelosi	1	●○○○○●●○○	●●○●○●○○○○○○●○○○○○○○○○○○○○○○○○○○○○

Method	Run	Problem: Glass	Bankrupt
Miller	1	○●●○●●●●○	○○●○○●●●○●●●●○○●○●○○●○●●○●○○●●○○○
	2	○○●○●●○●○	○●○●○●●●○●●●○○○●○●●○●○○●○○○●○●○●○
	3	●○●●○●●●○	○●●●●●○●○●●●●○●●●○○○○●○○○●○●●○○●○
	4	○●●●●○●●●	○○○●○○○●●○○●●○●●●●●●○●○●○●○○○○○●○
Kitano	1	●●●●●●●●	●●●●●●●●●●●●●○●●●●●●●●●●●●●●●●●●●
	2	●●●●●●●●	○○○○○○○○○○○○○○○○●●●●○○○●○○○○●●●●○
	3	●●●●●●●●	●●●●●●●●○○○○●●●●●●●●○●○○○○○○●●●●●
	4	○○●●●●●●	●●●●●●●●○○○○○○○○●●●●●●●●○○○○○○○○●
Nolfi	1	○○●●●●●○	○○○●○●○○○○○○●○○●○○●●○○○○○○○●○●○○
Cangelosi	1	○○●○○○○●○	●○○●○○○○○○○●○○○●○○○○○○○○○○○○○○○○○

Table 6.5: Summary of the input selection by different methods for different problems. The use of a certain input is denoted with ●, and leaving out with ○. The inputs are ordered in increasing order from left to right. These selections can also be observed in the pictures of the best networks below in Figures 6.4 to 6.18.

(This page intentionally left blank)

XOR

All methods found at least one feasible solution (i.e., a solution where the network has at least minimal structure required to learn the problem). The methods by Miller and Kitano found a feasible solution in the initial population. The method by Nolfi and Parisi found two feasible solutions. The method by Cangelosi et al. found none according to the evolution graph, but the network from run 4 is supposed to be a feasible solution. There were some peaks in the evolution graph for that run, which might imply the the winner networks were able to learn the problem only sporadically. It may be that the short training time was not enough for the small networks produced by the method, while the bigger networks generated by the other methods learned in a more stabile manner.

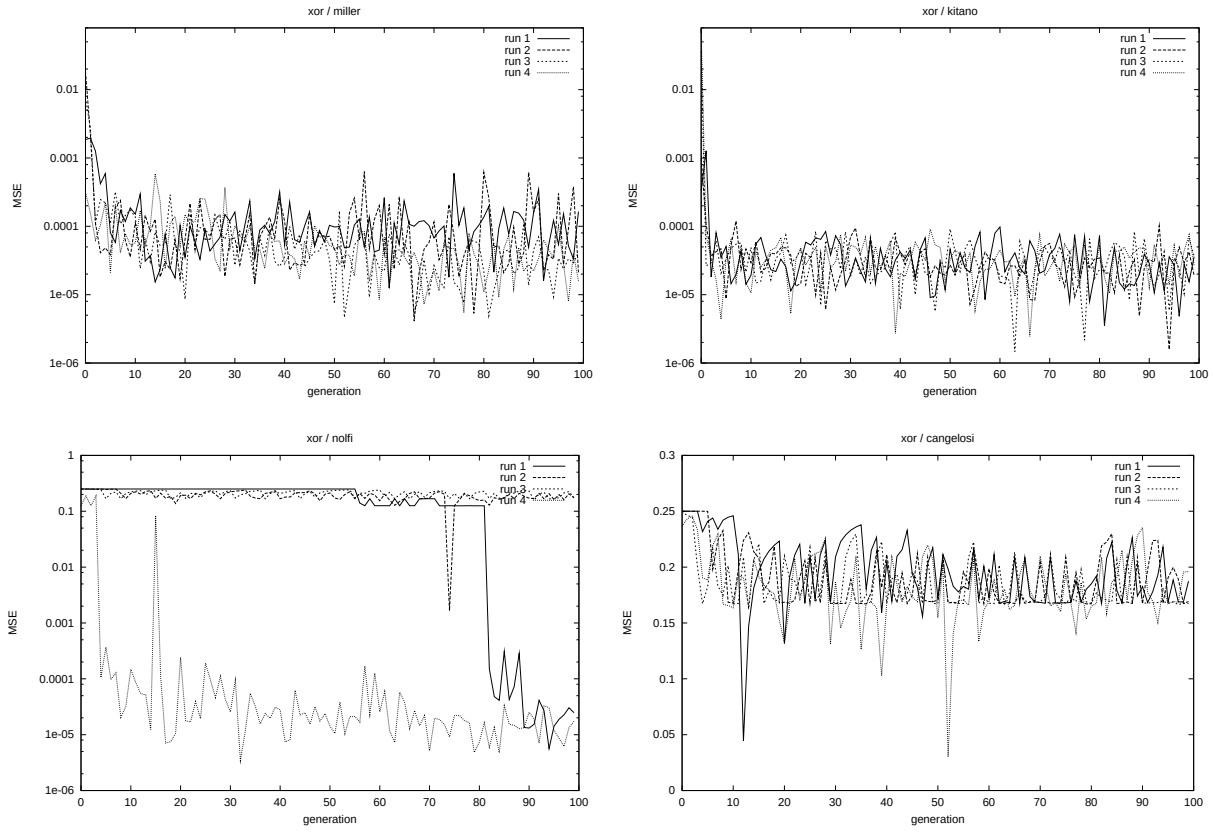

Figure 6.3: Evolution logs for the `xor` problem.

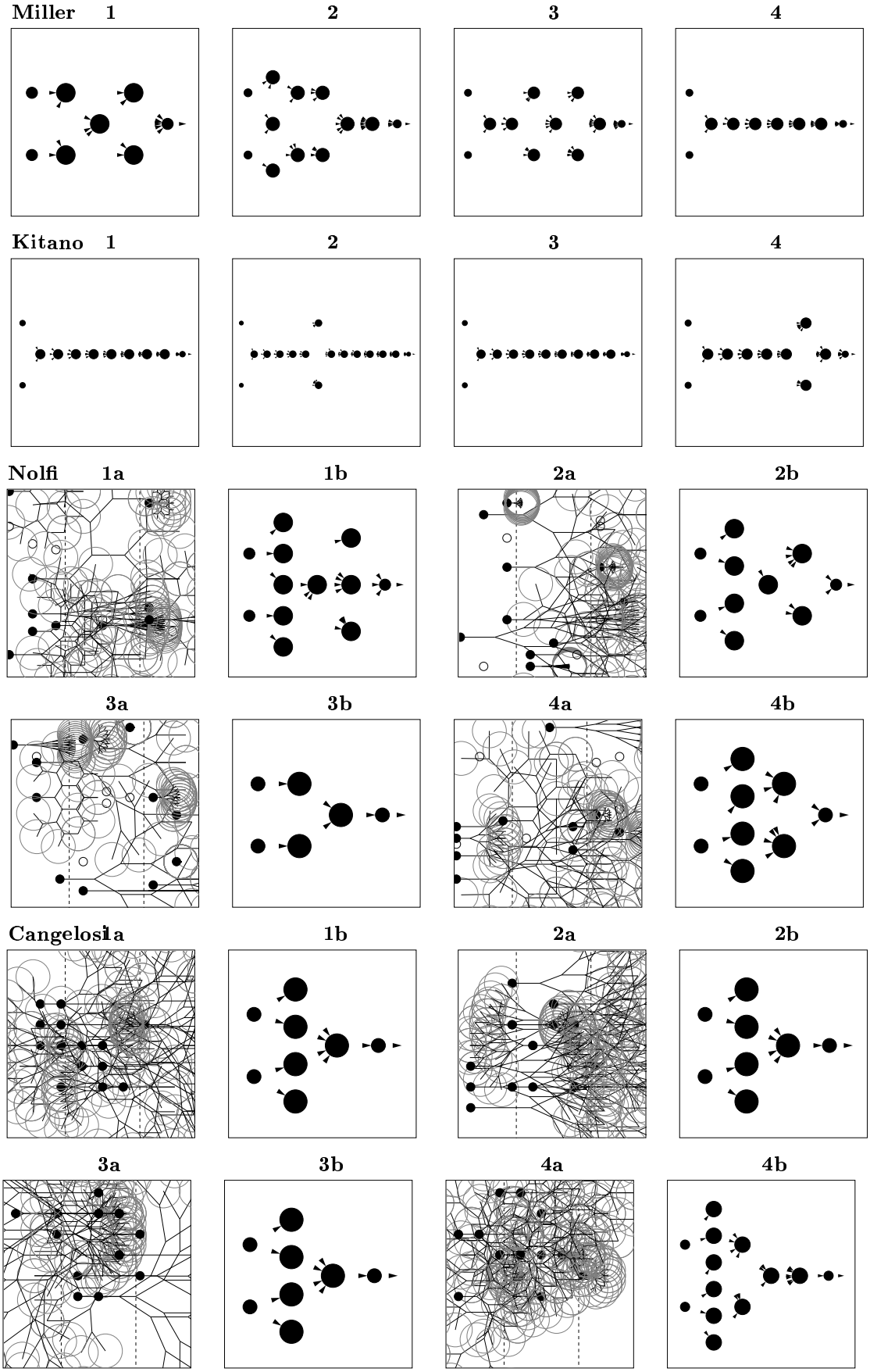

Figure 6.4: Networks for the XOR problem.

Encoder

The methods by Miller et al. and Kitano produced feasible solutions in all runs. The solutions found by Kitano's encoding were almost same; no hidden units and all inputs connected to all outputs. One solution was a bit more sparse, but that run seems to have performed slightly worse on average, if we look at the evolution curve. The encoding by Miller et al. left some unnecessary hidden units, but it had also connected all inputs directly to their corresponding outputs. When we compare the connectivities of the solutions and the average performances of the runs, we can see that the ("unnecessary") extra connections may have helped in neural learning. The evolution graphs below are somewhat similar (on average) to those reported by Kitano (1990a). Three runs with the method by Kitano achieved a very good performance right in the first generation. This may imply that feasible solutions with all inputs enabled were present in the initial populations.

The methods by Nolfi and Parisi and Cangelosi et al. performed worse than the two methods above. Notice that runs 3 and 4 with the method by Nolfi and Parisi were done with 64 potential neurons. Those runs yielded even better solutions than the runs with the Kitano's method. The method by Cangelosi et al. did not find any feasible solutions.

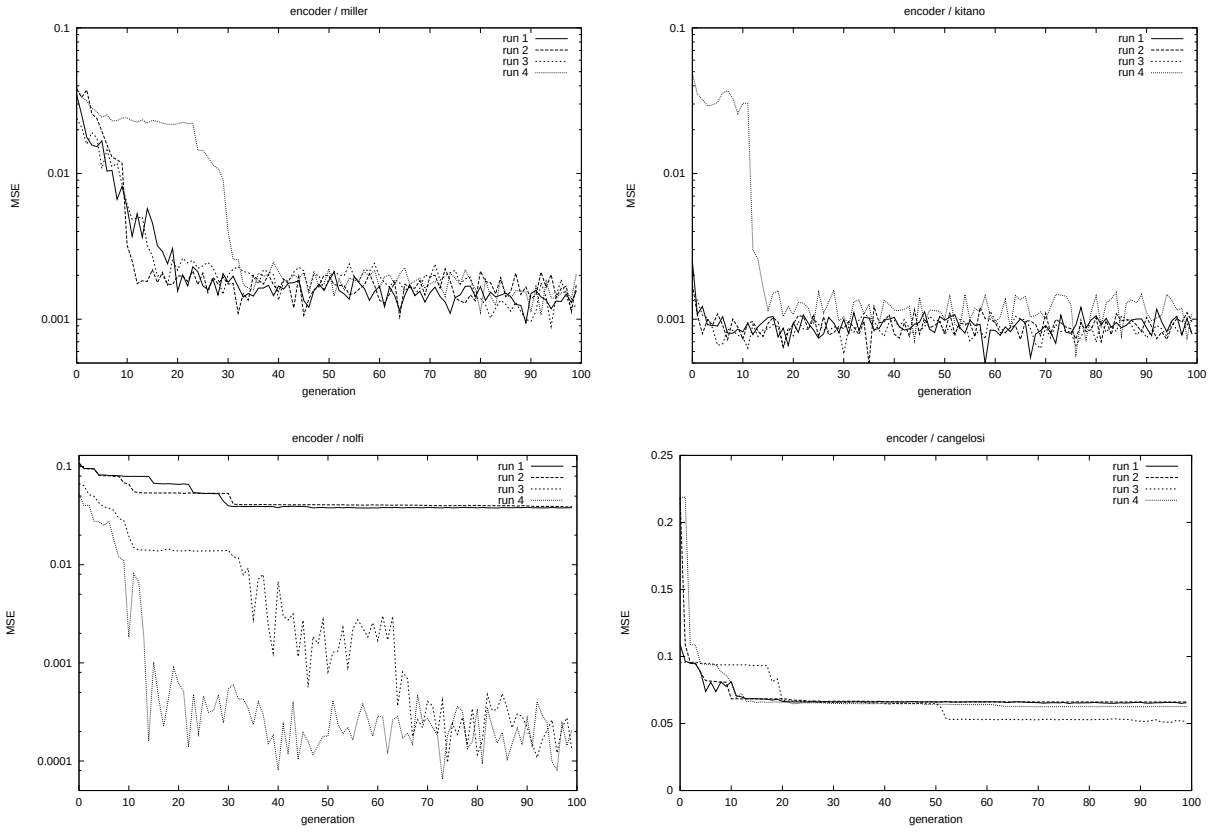

Figure 6.5: Evolution logs for the `encoder` problem.

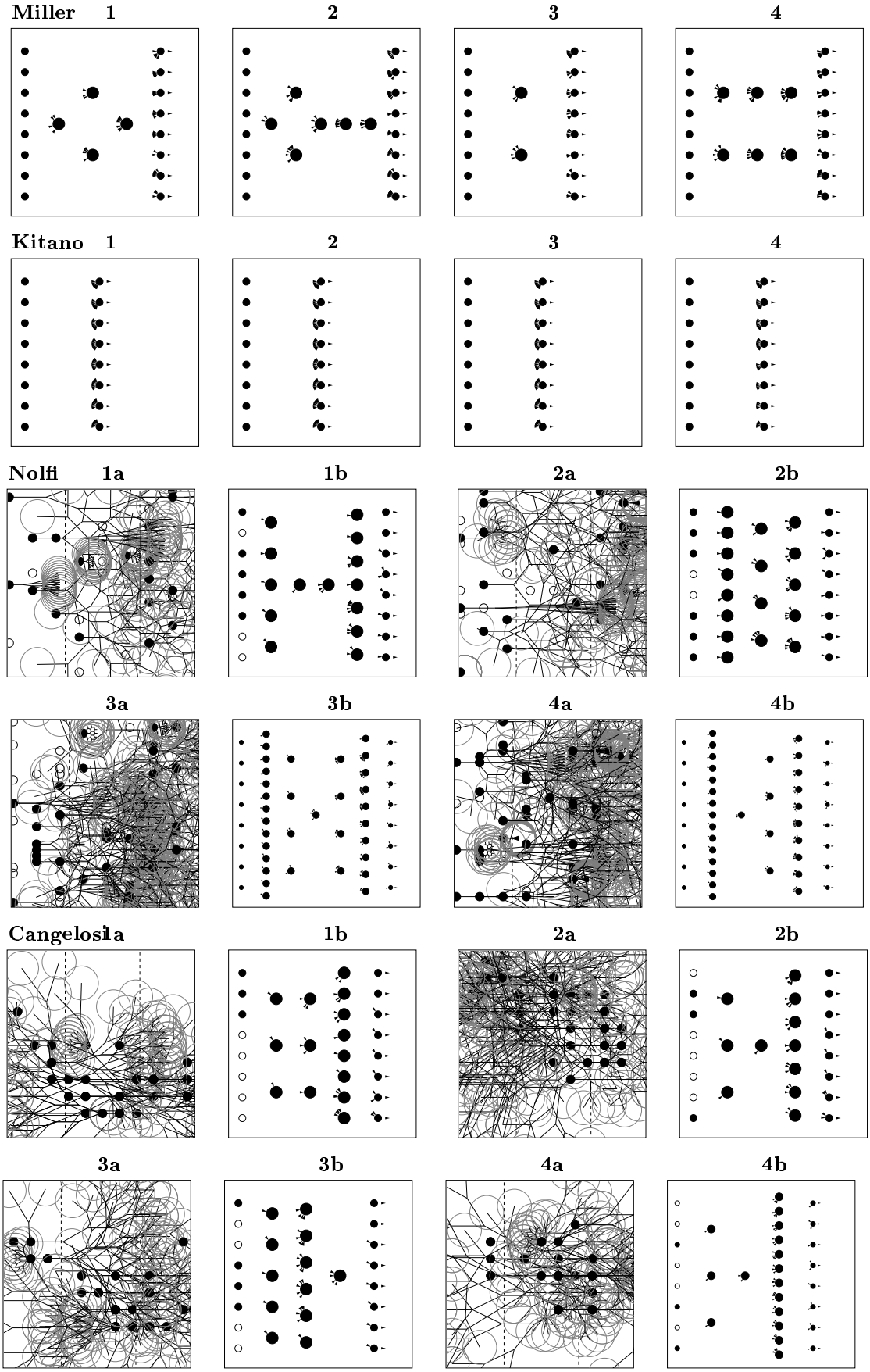

Figure 6.6: Networks for the `Encoder` problem.

Additive

The method by Miller et al. found the minimal selection of inputs in three runs. The method by Kitano found a feasible solution in only one of the four runs. It enabled always the eight first inputs, which is propably because the blocks in the connectivity matrix usually have size of a power of two. The method by Nolfi and Parisi performed also quite well, yielding two feasible solutions. The method by Cangelosi et al. performed clearly worst: no feasible solution was found in any of the runs. Notice that inputs 3, 4, and 5 require just a straight connection from inputs to the output, input 2 requires just one hidden unit, and input 1 requires rather many hidden units. Some connectivity pattern like this might be seen by visual inspection of the networks given by the method by Nolfi and Parisi, and also from the networks 2 and 3 by the method by Cangelosi et al. The connectivity in the networks given by the method by Miller et al. was too high to see any patterns of this sort.

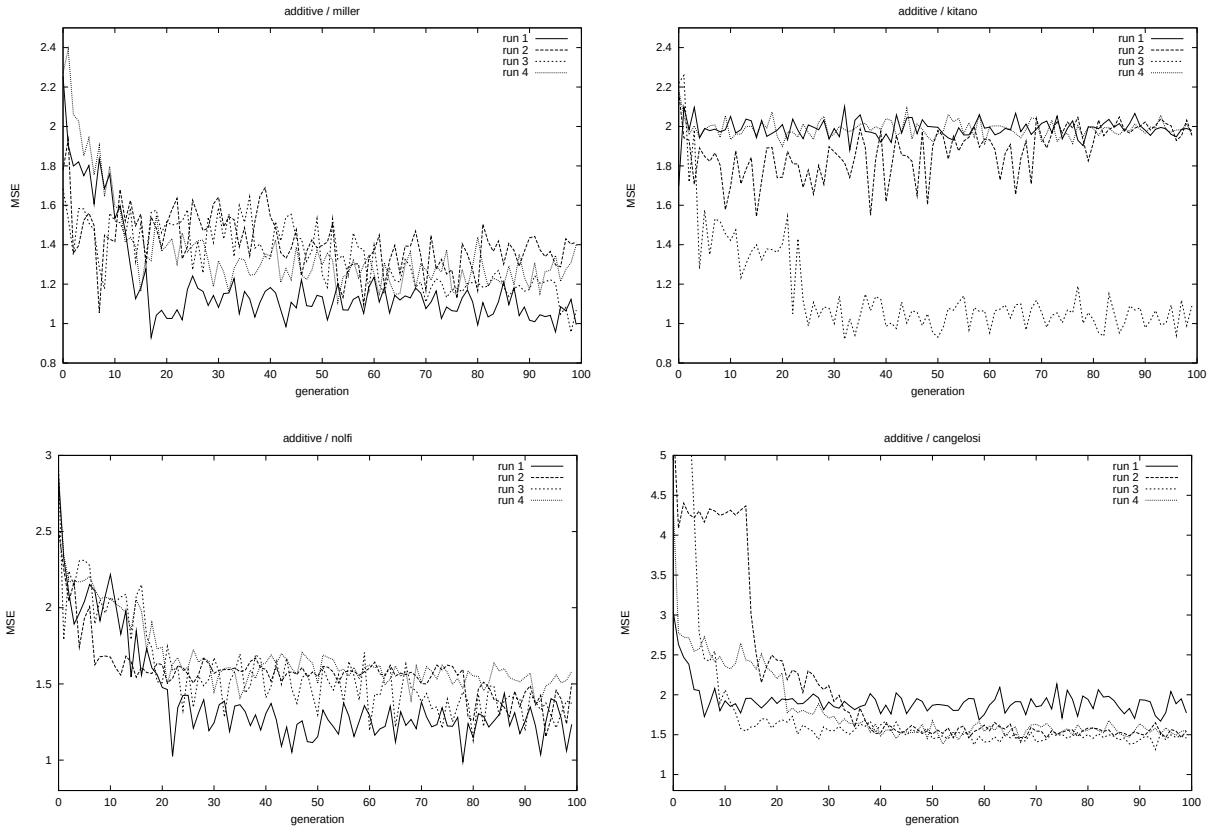

Figure 6.7: Evolution logs for the `additive` problem.

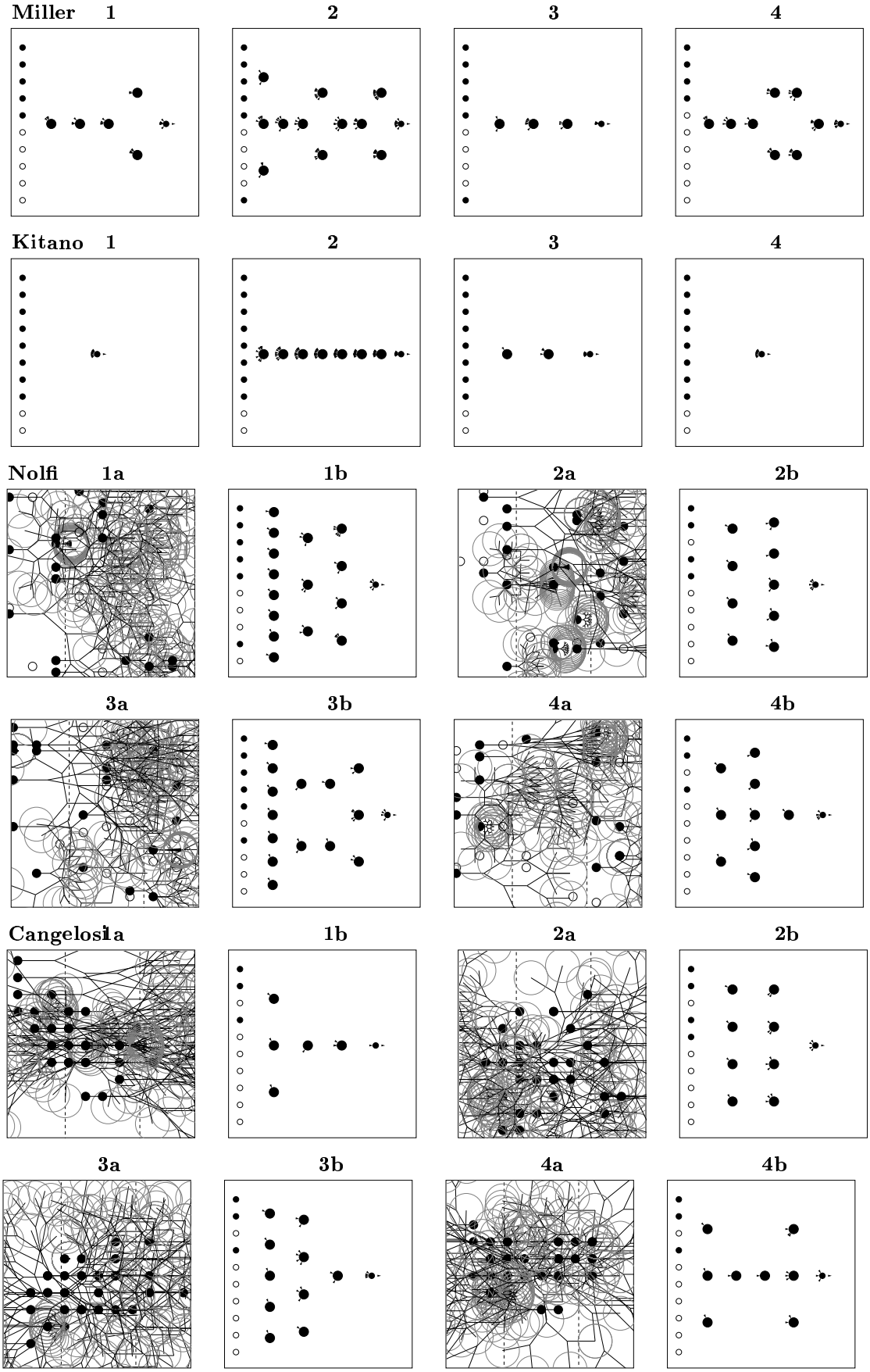

Figure 6.8: Networks for the `Additive` problem.

Interaction

The methods by Miller et al, Kitano, and Nolfi and Parisi found feasible solutions in all runs. The method by Nolfi and Parisi found the exactly correct input selection in all four runs. The method by Cangelosi et al. found only one feasible solution. It is difficult so see any strong connectivity pattern relevant to the problem data by visual inspection.

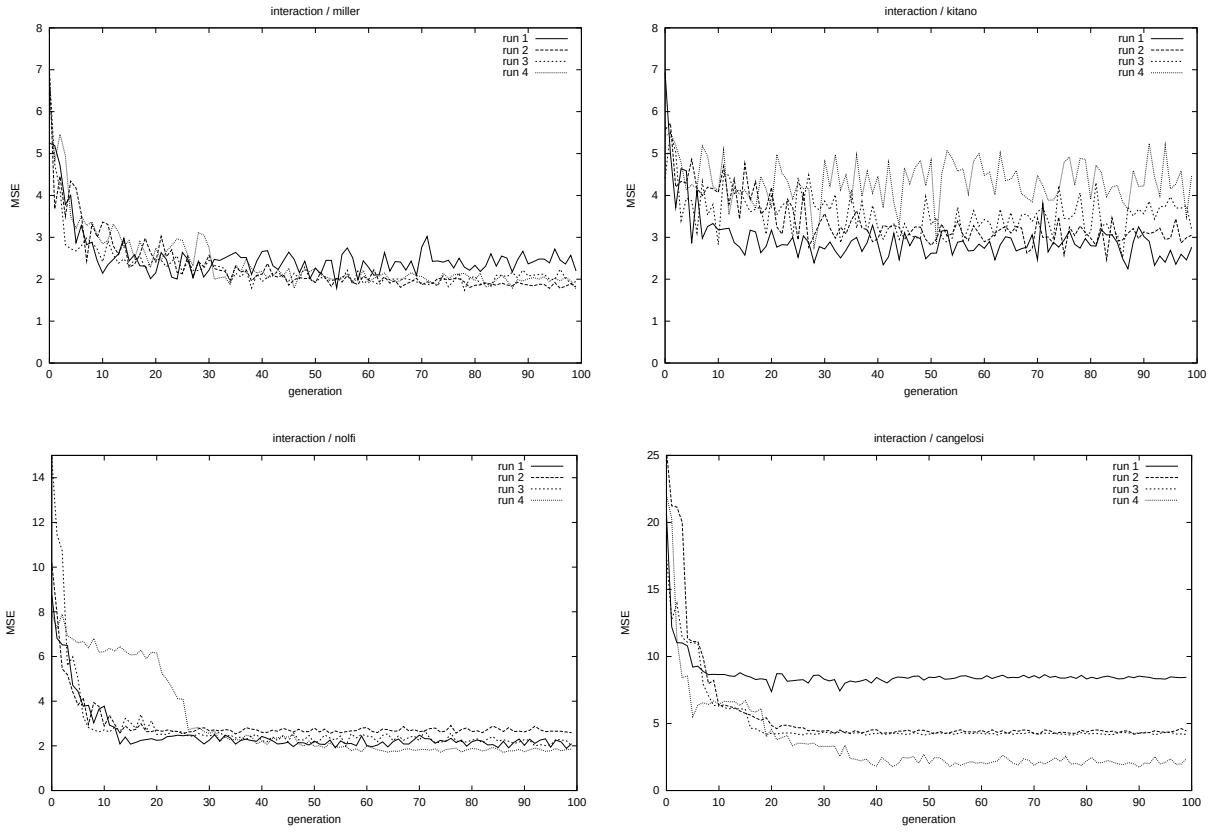

Figure 6.9: Evolution logs for the `interaction` problem.

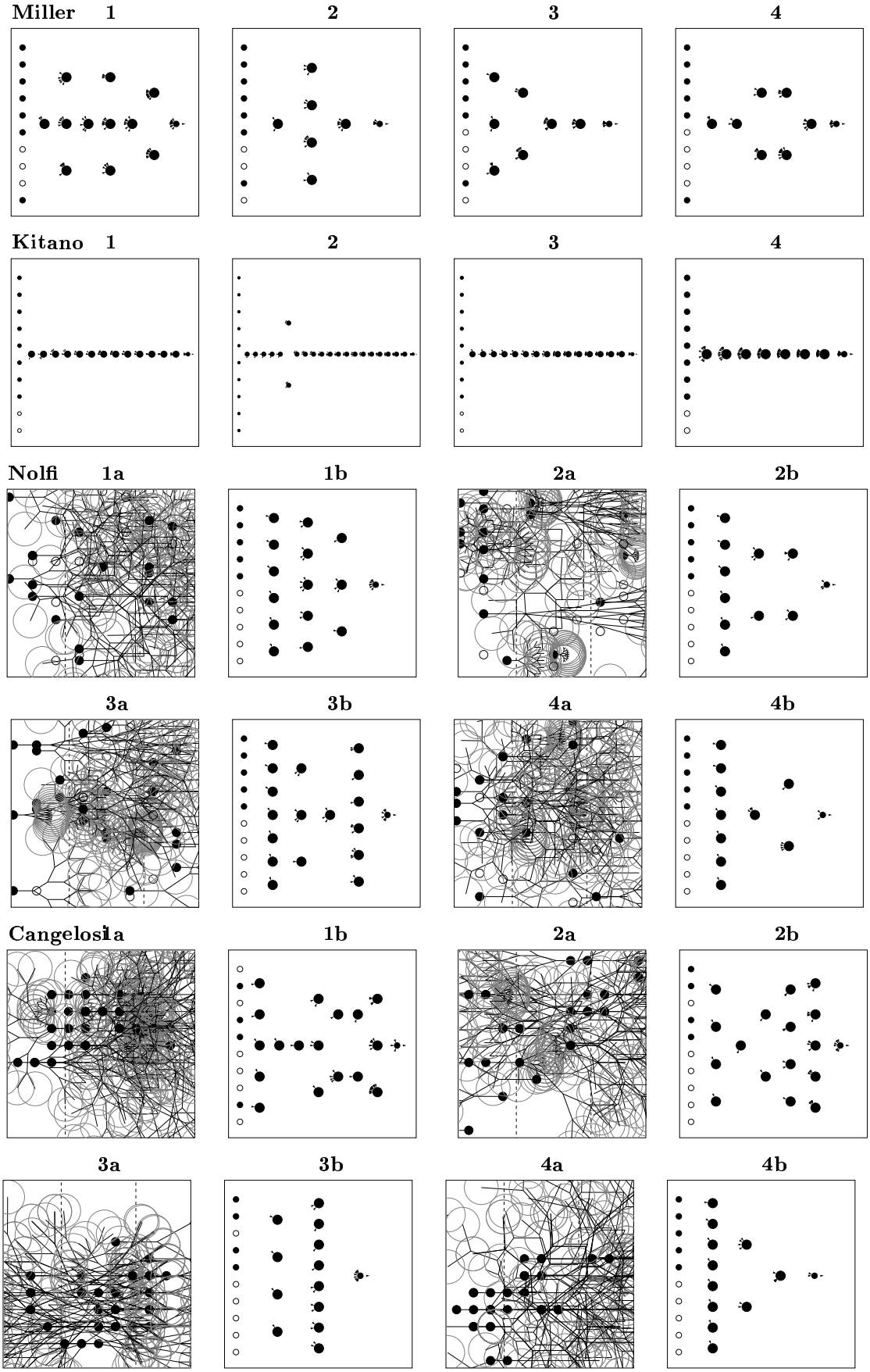

Figure 6.10: Networks for the `Interaction` problem.

Cancer

The method by Miller et al. performed best, but the method by Nolfi and Parisi was also quite good. There was not much evolution in the runs with the methods by Miller et al. and Kitano, and rather good solutions were found right in the initial population. Kitano's method produced the networks with the lowest classification error, although the networks generated by the method by Miller et al. were not much worse.

Inputs 1 (clump thickness) and 6 (amount of bare nuclei) were enabled in all solutions.

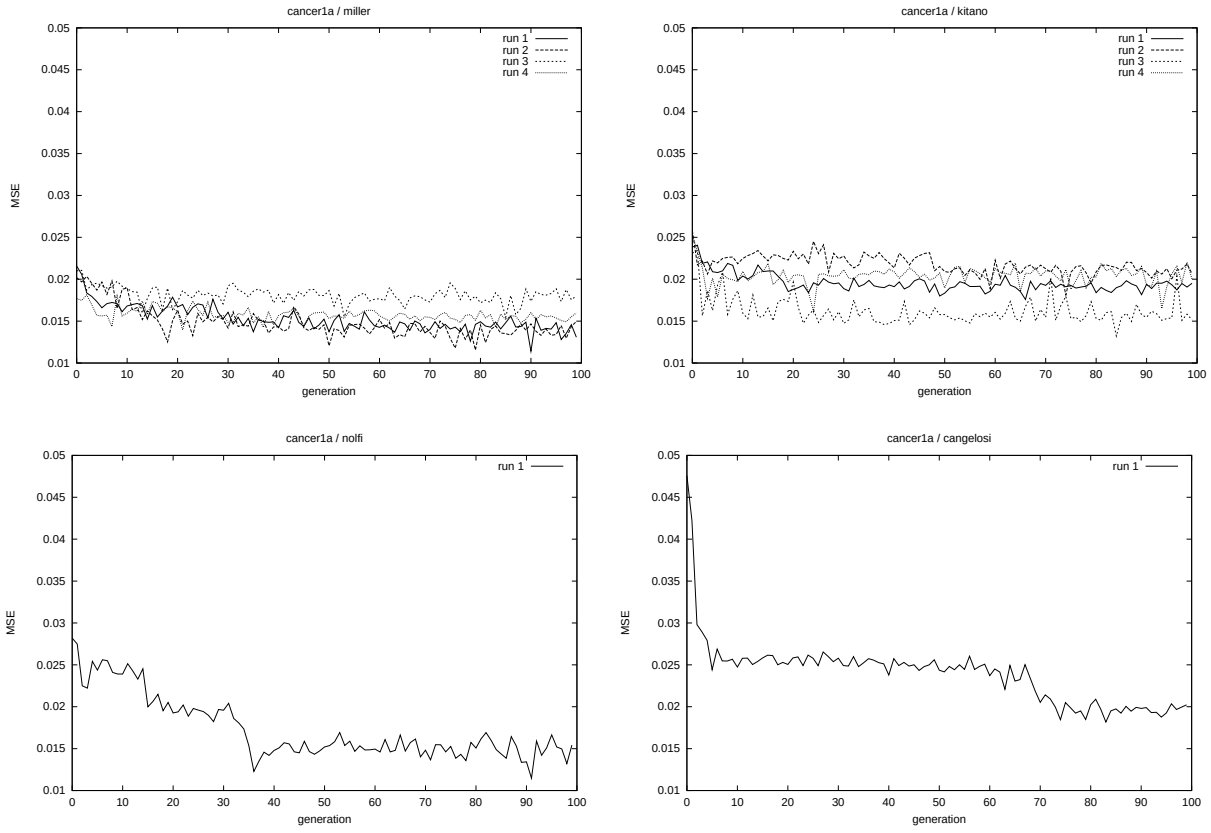

Figure 6.11: Evolution logs for the `cancer1a` problem.

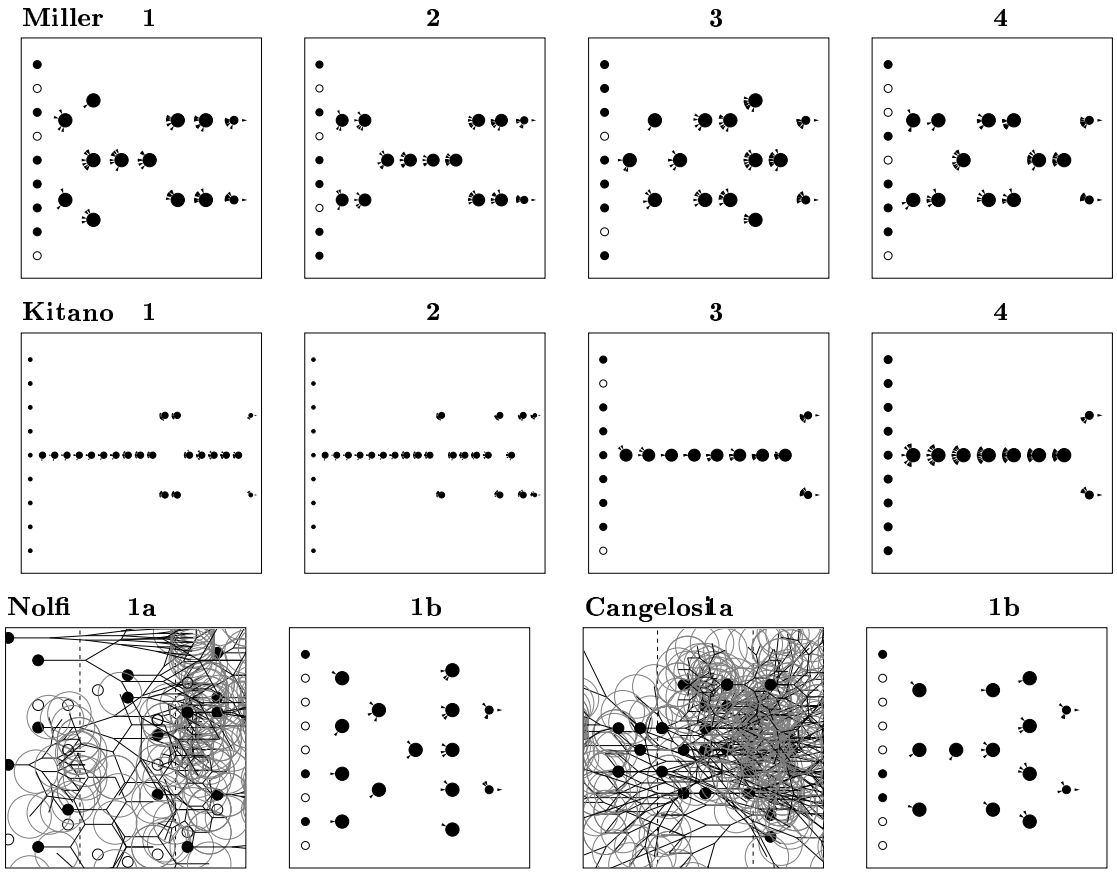

Figure 6.12: Networks for the `Cancer` problem.

Glass

The results for this problem were intriguing. At first sight, the method by Nolfi and Parisi performed clearly best. However, the averages given in Table 6.4 indicate that the best solution found by that method performed quite badly. We can also observe that the variance for the performance of that solution was quite high. What we observe here in the graphs is propably the noise effect described in Section 6.1.1. Inspecting all the runs, we can note a tendency that the solutions that have a weaker average give higher variance and perform better according to the evolution graphs.

Kitano's method produced very large and highly connected networks. It also produced the network with the lowest classification error. The solution found by the method by Nolfi and Parisi was somewhat smaller than the solutions found by the methods by Miller et al. and Kitano.

The inputs 3 (magnesium) and 8 (barium) were enabled in all runs by all methods. They were the only inputs enabled in all the runs done with the method by Miller et al, and the method by Cangelosi et al. selected exactly these.

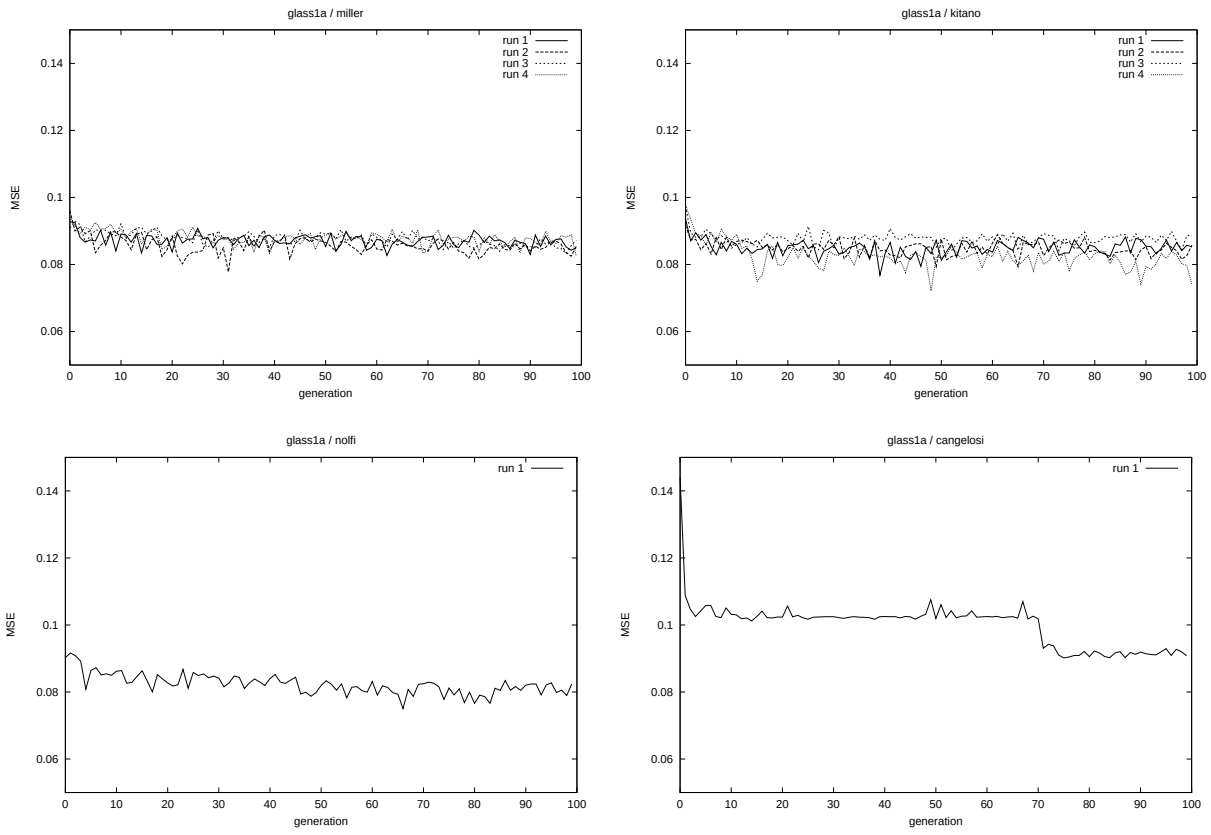

Figure 6.13: Evolution logs for the `glass1a` problem.

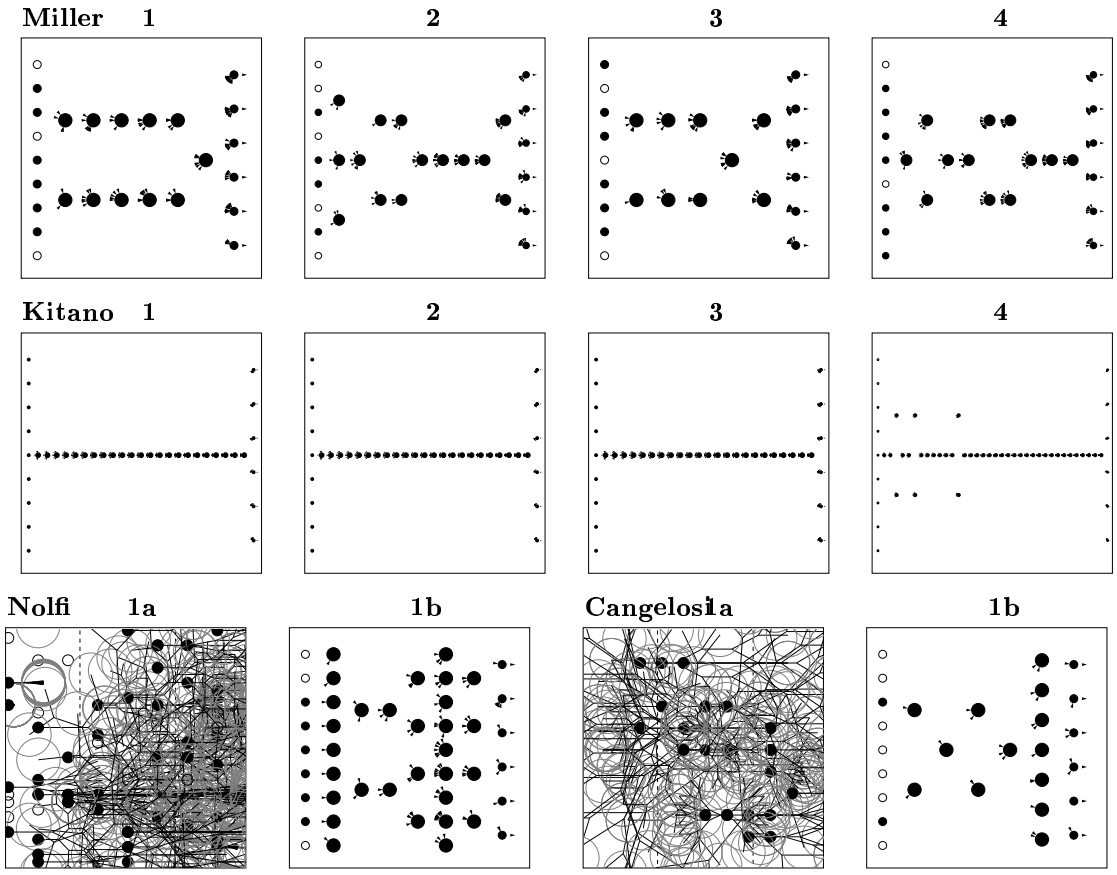

Figure 6.14: Networks for the `Glass` problem.

Heart

The results for the Heart problem are similar to the Glass problem. The graphs show that the method by Miller et al. performed best, much better than the method by Kitano, as did the method by Nolfi and Parisi. However, the results shown in Table 6.4 tell again a different story; Kitano's encoding yielded the solution with the lowest classification error, although the results for the method by Miller et al. were not far behind.

Inputs 1 (age) and 2 (sex) were enabled in all runs. Inputs 4 (atypical angina), 5 (non-anginal chest pain), 27 (flat peak in exercise ST segment), 30 (number of major vessels colored by fluoroscopy), and 32 (normal thal) were selected quite often. The selection of the input 11 (attribute present indicator) can be ignored since input 10 was not selected so often.

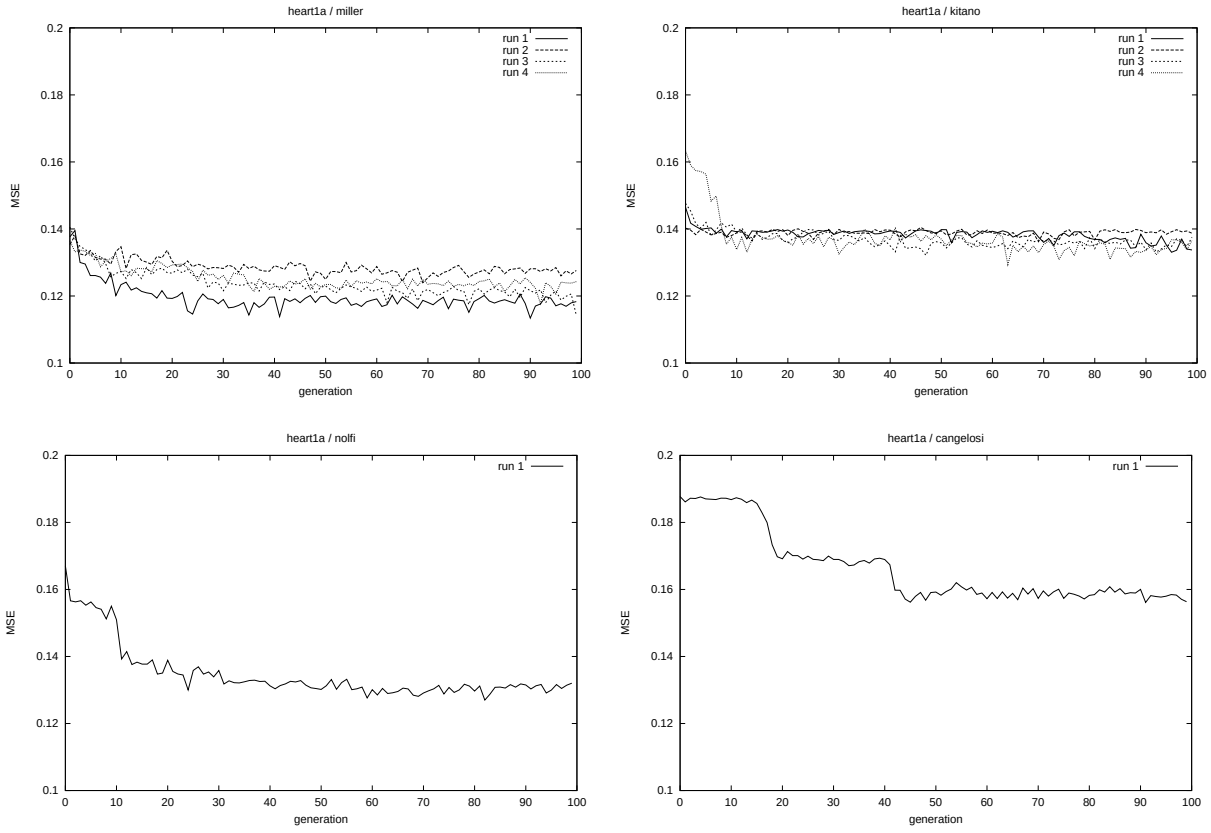

Figure 6.15: Evolution logs for the `heart1a` problem.

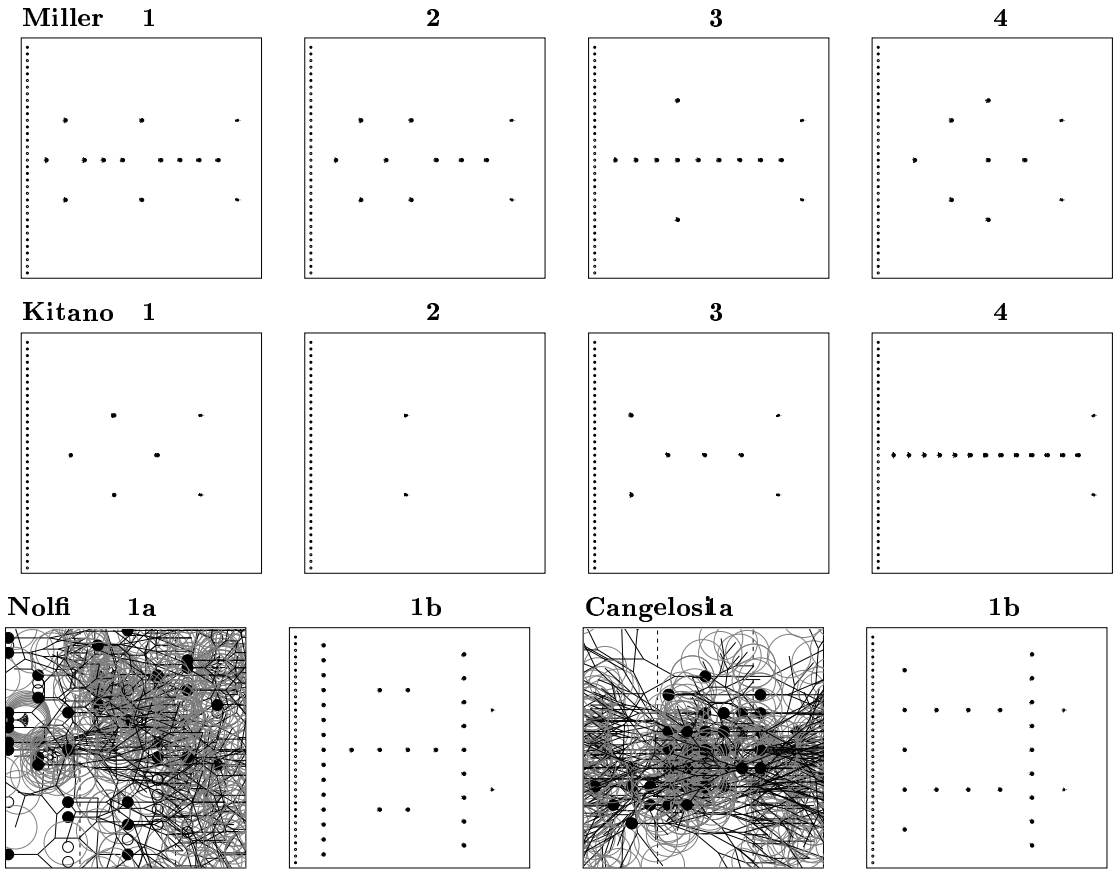

Figure 6.16: Networks for the `Heart` problem.

Bankrupt

The best network evolved with Kitano's encoding had the lowest classification error, while the method by Miller et al. performed about equally well on average. The evolution log graphs are again misleading in this respect. Notice that the performance with the test set was actually lower than with the validation/evaluation set. This result is hard to explain.

There were no inputs selected in all the runs. Variables 8 (Current Assets / Net Sales), 12 (Equity / Net Sales), and 16 (Market Value of Equity / Book Value of Debt) were selected in all runs with the method by Miller et al., but the best performer used only one of them. This variation might imply that the solution is quite multimodal in respect of the input selection. This is understandable, since all of the financial ratios have been found usedful in some studies.

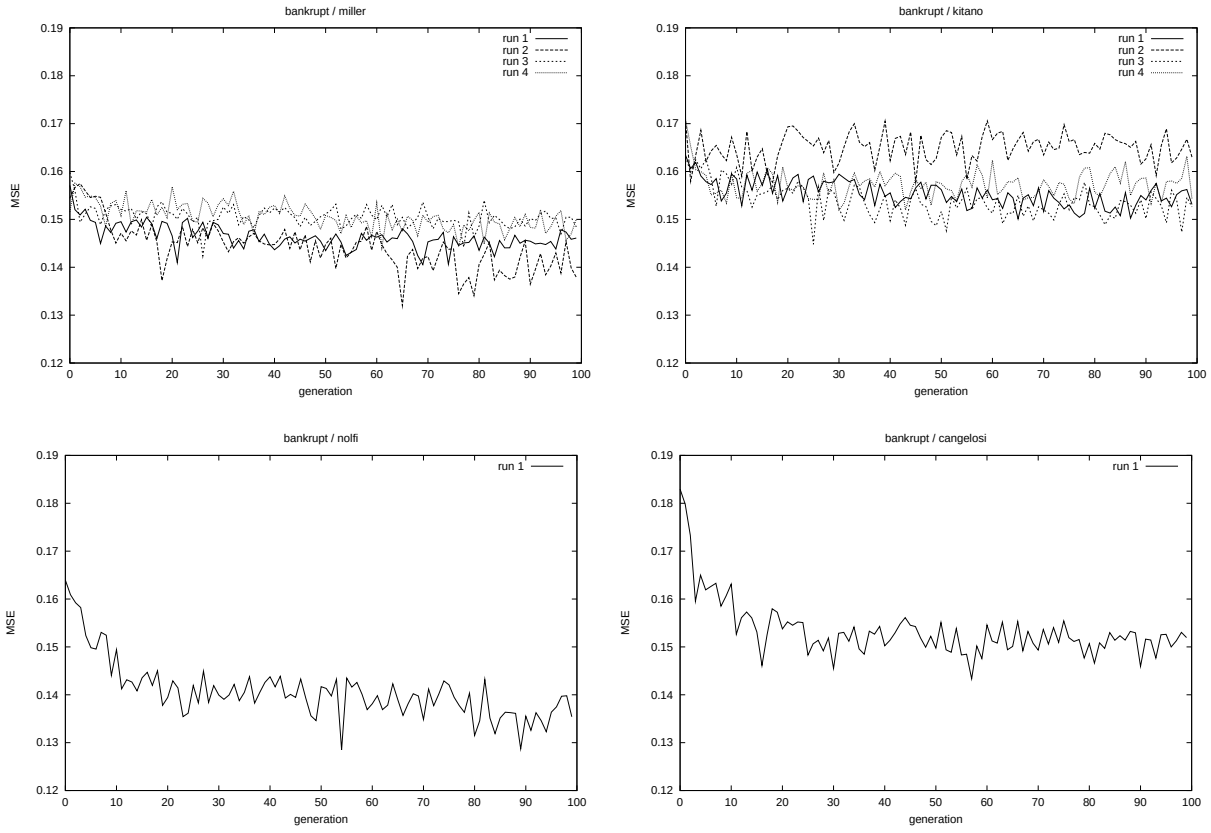

Figure 6.17: Evolution logs for the `bankrupt` problem.

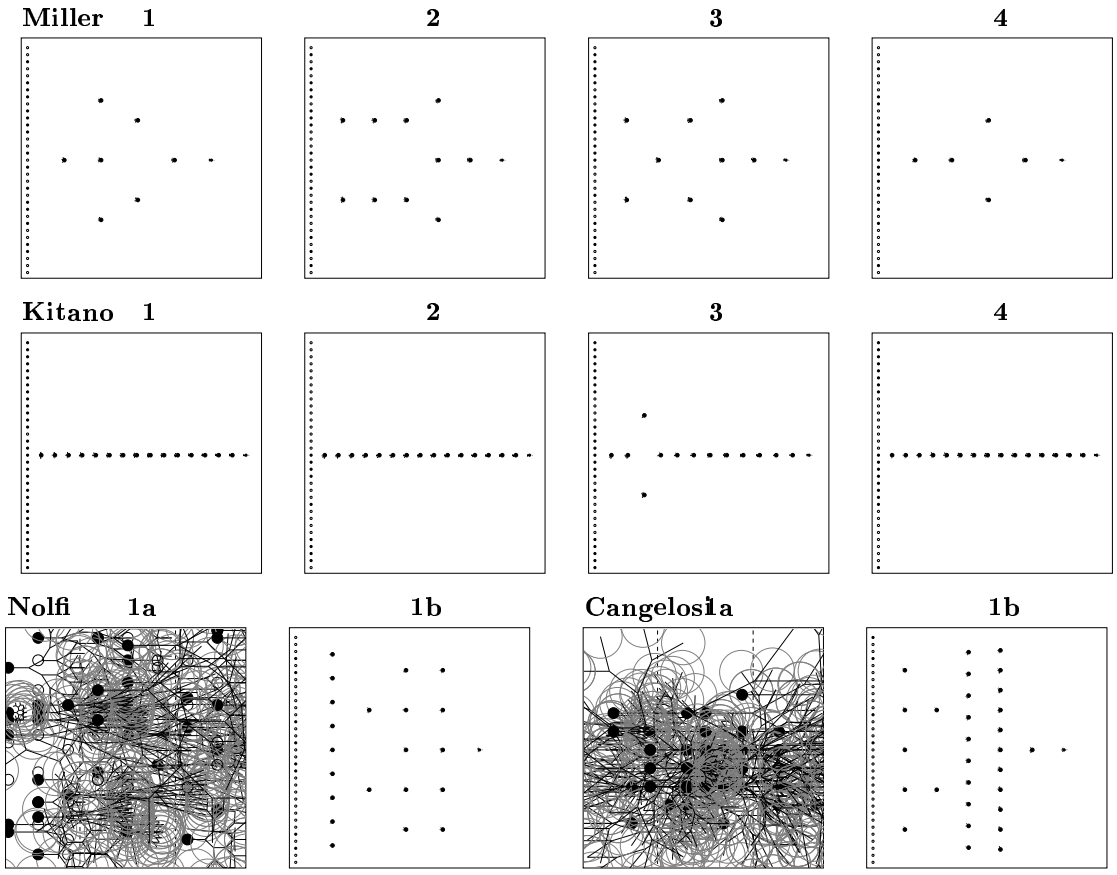

Figure 6.18: Networks for the `Bankrupt` problem.

6.3 Discussion

When we look at the results in Section 6.1.1, it seems that the learning gets more difficult as the network size approaches the minimal size. But, the best solutions found by the smaller networks are maybe even better than those found by the networks that perform better on average. We can also observe this phenomena in the results from the evolved networks. One explanation may be that a bigger network has more alternatives where to begin the search, so that, if it gets stuck in a poor local optima of one alternative it can still explore other alternatives. We might think that in a highly connected network, the number of similar alternatives would be exponential.

An alternative way of looking at this are maximally compressed computer programs (here implemented as neural networks). This is a view considered by Kauffman(1995), somewhat as follows. Think of a problem, and the shortest possible program that solves the problem. The shortest program can be defined by that it has no redundancy (repeating or unnecessary code) whatsoever, but any longer program does. For learning such programs it is important that the programs with redundant code are more robust; change them a little and a small difference in behaviour can be observed, which is the requirement for all learning and evolutionary adaptation. For the minimal program, any change results in total change in function, as the parts of the program are completely interdependent. Therefore, it is more difficult to find a correct program by adaptation as the required length becomes shorter and the topology of such an optimal network would perform less well on average after neural learning.

But what if we really want to find such smaller programs? One option might be to use regulation terms for the fitness functions, as suggested in Section 3.2. Another might be to use stronger selection and elitism, which might not care so much about the average performance of the individuals.

Chapter 7

Conclusions

We evolved artificial neural network topologies using the genetic encoding methods by Miller et al., Kitano, Nolfi and Parisi, and Cangelosi et al. The fitness of the network topologies was calculated using the training error with one of eight problems during the evolutionary search. The networks were trained using the RProp algorithm, which is a backpropagation variant. The problems included four artificial problems and four real-world classification problems. The best networks found by the evolutionary search runs were compared according to their classification accuracy with the real-world problems, and according to their ability to select the relevant inputs with the artificial problems.

The network topologies evolved with the encoding method by Kitano gave generally the best classification results. The method by Miller et al. was not far behind. The encoding method by Nolfi and Parisi performed worse than the two methods above, and the method by Cangelosi et al. was the weakest.

However, we observed that the best networks generated by such "worse" methods had sometimes much larger variance in their learning performance when trained several times with different random initializations of the network weights. We observed that sometimes as the network size becomes smaller, the average learning performance becomes worse, although the best networks trained with the smaller topologies can become even better than with the larger networks.

The classification results for the evolved network topologies were not significantly better, if at all better, than the results for the topologies found by Prechelt (1994) with a systematic search for the PROBEN1 datasets, three of which we used in this study.

The method by Miller et al. was perhaps the best for selecting the correct inputs, although the method by Nolfi and Parisi was not far behind. The latter method generally found solutions with much less inputs. The method by Kitano performed clearly worst in this task, because it had great difficulties in generating small details in the network description matrix it used.

We note that because of high requirements of computing resources required to perform the tests, we were able to evolve only one to four solutions for each problem-method pair. It is not possible to draw any statistically reliable conclusions from such a few samples.

In his introduction of the graph grammar encoding method (Kitano 1990a), Kitano

compared his method to the direct encoding method by Miller et al. with the 8-x-8 and 4-x-4 encoder problems. Although we confirm these results, we also see that this problem was inadequate for comparison, and perhaps somewhat biased towards the graph grammar encoding. We can not agree with the note by Siddiqi and Lucas (1998) that Kitano's original comparison was unfair because he used bad parameters for the direct encoding. Our analysis shows that the initial network connectivity was about the same for Kitano's method. However, it is not easy to implement such parameters for the graph generation grammar. Therefore, we can consider it good to have the possibility to use the parameters with the direct encoding.

The encoding methods by Nolfi and Parisi, and Cangelosi et al. are based on using a two-dimensional space where the neurons are located and where they grow "axon trees" to form connections. The methods did not perform very well in our tests. However, some of our results indicate that we did not pick our parameters very well, and although the methods might have performed better with some other parameters, the sensitivity of the parameters is itself a big problem. We see that the bad parameters caused the methods to produce very small networks that were difficult to evolve, because most mutations were most likely fatal. The *expression* gene used by the method is somewhat questionable, because it increases the already high epistasis and disabling unwanted neurons is easy enough with the other genes. We added a new parameter, the connection radius of the axon tips, and found this parameter useful in adjusting the connectivity of the networks.

The most obvious future task would be adding more different encoding methods to the library and evaluating them with the same benchmarks. The major difficulties in this study were finding the best EA strategy parameters and the parameters for the encoding methods. Autoadaptation of these parameters would make the evaluation task easier. Some encoding method parameters, such as the connection radius parameter for the methods by Nolfi et al. and Cangelosi et al., would be quite easy to autoadapt. Making the network size more scalable should be rather easy for all but the method by Miller et al.

One approach which we did not compare in this work is using genetic regulation networks (GRNs) to control the ontogenetic construction of neural networks, instead of using production grammars such as Kitano's. Some of such methods are mentioned in the introduction in Chapter 4. We see both of these as techniques for evolving computer programs; GRNs are computationally equivalent to recurrent neural networks. The programs generated by such a direct low-level encoding method are used to control the construction of higher-level programs - the final neural networks. Although many people have used such indirect two-level mechanism, the differences between the different low-level methods (grammars and GRNs) have not been compared, neither analytically or experimentally.

Benchmarking the ability of the methods to evolve suitable networks could be done differently from what we did here. For example, the fitness function could calculate the difference (however that is defined) between an evolving network and a target network topology. The benchmarking could also be done with an image compression task, if the encoding methods are highly fractal in their nature. In that case, we would have to remember that a certain neural network topology can be formed by an enormous number

of different connection matrices, while the evolutionary landscape of image bitmaps would not have the same "caleidoscopic" nature.

Appendix A

Neural network program library

We developed an artificial neural network (ANN) C++ program library to accomplish the experiments of this study. It is used by the evolutionary neural network library described in Appendix C. The library uses the well-known Stuttgart Neural Network Simulator (SNNS) to train the networks. This creates some limitations on the neural architecture and learning parameters:

- The training algorithm is global for the network, so different part of the network can not be trained with different algorithms, training parameters or radically different unit types
- Only one network object can exist and be run at a time
- Some training parameters can't be set (for example whether or not a bias value should be used, η^+ and η^- parameters for Rprop, etc.)

The neural library was validated by a comparison with Prechelt's (1994) results. The results were not exactly same as what would have been expected (this was probably due to different learning and early stopping parameters, and the above limitations of SNNS), but within acceptable limits.

A.1 Backpropagation

Below is a brief description of the backpropagation and RProp algorithms used in this study.

The basic idea of backpropagation (Rumelhart, Hinton, and Williams 1986; SNNS 1995) is to use the derivative of the transfer function to get an estimate about the direction to which the weights of the network should be changed. This partial derivative has the form $\frac{\partial E}{\partial w_{ij}}$, which in words means the slope of the error function in relation to any single weight.

First, a training pattern is fed to the inputs of the network and the signals are propagated to the output layer. This is the forward-propagation phase. After that, the

difference, or error, from the desired outputs o_j is calculated as MSE

$$E = \frac{\sum_k^{N_o}(y_j - o_j)^2}{N_o}. \tag{A.1}$$

This error is then back-propagated to the input layer and the weights are changed using the derivative of the threshold function

$$\Delta w_{ij} = -\eta y_i \delta_j, where \tag{A.2}$$

$$\delta_j = \begin{cases} \theta'(y_j)\sum_k \delta_k w_{kj} & \text{for hidden units} \\ y_j - o_j & \text{for output units,} \end{cases} \tag{A.3}$$

where δ_i is the error signal at a target unit and η is the *learning speed*. Derivative of the sigmoid function is

$$\theta'(o) = \theta(o)(1 - \theta(o)) = \frac{\partial E}{\partial w_{ij}}. \tag{A.4}$$

This is done for each pattern in the set, for a number of training cycles (typically some 100 to 10000 cycles are required for the standard backpropagation). In *batch learning*, which is a typical procedure, the weights are updated only after the weight-deltas have been calculated (and added together) for all the patterns in the training set (i.e., in the end of the training cycle).

Resilient backpropagation

Resilient backpropagation (Riedmiller 1993; SNNS 1995), or Rprop, was used in all experiments of this work. It is a modification of the original backprop algorithm. The basic idea is to eliminate the harmful influence of the size of the partial derivative $\frac{\partial E}{\partial w}$ on the weight step. In Rprop, only the *sign* of the derivative is used to indicate the *direction* of the weight change. The *size* of the weight change is determined with a local, weight-specific *update value* Δ_{ij}

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_{ij}^{(t)} & \text{, if } \frac{\delta E}{\delta w_{ij}}^{(t)} > 0 \\ +\Delta_{ij}^{(t)} & \text{, if } \frac{\delta E}{\delta w_{ij}}^{(t)} < 0 \\ 0 & \text{, otherwise,} \end{cases} \tag{A.5}$$

where $\frac{\delta E}{\delta w_{ij}}^{(t)}$ denotes the summed derivative over all patterns of the pattern set. Normally weight updates are done for each pattern separately. This detail of making the weight adjustments for all training patterns simultaneously is called *batch learning* and it is believed to help the learning.

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ \cdot \Delta_{ij}^{(t-1)} & \text{, if } \frac{\delta E}{\delta w_{ij}}^{(t-1)} \cdot \frac{\delta E}{\delta w_{ij}}^{(t)} > 0 \\ \eta^+ \cdot \Delta_{ij}^{(t-1)} & \text{, if } \frac{\delta E}{\delta w_{ij}}^{(t-1)} \cdot \frac{\delta E}{\delta w_{ij}}^{(t)} < 0 \\ 0 & \text{, otherwise,} \end{cases} \tag{A.6}$$

where $0 < \eta^- < 1 < \eta^+$.

At the beginning, all update values Δ_{ij} are set to some small value, say 0.1. The update values are constrained to constant range $[\Delta_{min}, \Delta_{max}]$. Riedmiller (1993) has suggested general values $\Delta_{max} = 50.0$ and $\Delta_{min} = 1e^{-6}$, but has also noted that setting Δ_{max} to some smaller value, like 1.0, would smoothen the convergence. Intuitively it would seem that Δ_{max} would affect the parameters of the early-stopping scheme described in page 8.

For the decrease factor η^- and increase factor η^+ Riedmiller has suggested general values of $\eta^- = 0.5$ and $\eta^+ = 1.2$. He also suggested that these parameters should be rather robust for obtaining optimal or nearly optimal convergence speed for most problems. This robustness of the parameters was the reason why this model was selected for training the evolving neural networks.

The RProp implementation in the SNNS uses also weight decay (see Section 2.3.1).

A.2 Equalization

The pattern set handling class of the library contains methods for equalization of the pattern sets. A pattern set can be equalized using the equalization parameters acquired from the pattern set itself, or using parameters from another set. Typically the validation and testing sets are equalized with the parameters acquired from the training sets.

Histogram equalization

A histogram is made for each component vector $\vec{s}$ of the training pattern set. A histogram $\vec{h}$ is formed by counting the number of the vector elements in $\vec{s}$ that fall in each slots with domain function $d(x, \vec{h}) = \left[\left[|\vec{h}| \cdot \frac{x - min(\vec{s})}{max(\vec{s}) - min(\vec{s})} \right]\right]$. Histogram resolution $|\vec{h}| = 100000$ was used in this study. The histogram is then transformed into a cumulative histogram with $c_i = \sum_{j=1}^{i} h_i$. The cumulative histogram can then be used to equalize values with $q(x, \vec{c}) = min(\vec{s}) + (max(\vec{s}) - min(\vec{s})) \cdot \frac{c_{d(x,\vec{c})}}{|\vec{s}|}$; the original component vector is equalized with $s'_i = q(s_i, \vec{c})$. This is done for all the pattern vector components. Other patterns can be equalized using the same equalization vectors.

We could imagine that the histogram equalization removes any clustering inside any one vector component, but also makes small variations within the clusters more easily detectable and possibly enchances clusterings within dependent component hypercubes.

Gaussian equalization

Mean $\bar{X}$ and standard deviation σ are calculated for each pattern set vector component set. Values are equalized $y_i = \frac{x_i - \bar{X}}{\sigma}$, so that the mean will be 0 and the standard deviation 1.0. This linear mapping does not lose any inherent clustering inside a single pattern component as the histogram equalization does.

A.3 Drawing networks

The Miller and Kitano encoding methods do not provide two-dimensional coordinates for the neurons, so a drawing algorithm must be used. We introduce below a simple algorithm suitable for drawing feedforward networks:

1. Split the hidden units into layers using a simple rule:

$$layer(i+1) = \begin{cases} layer(i)+1 & \Leftarrow connected(i, i+1) \\ layer(i) & \Leftarrow \neg connected(i, i+1) \end{cases} \tag{A.7}$$

2. Position input and output units according to their index
3. Iterate from the first to the last hidden layer:
   (a) For each unit in the layer, calculate the "optimal" position in the sense that the unit would be positioned at the average y-position of source unit of all connections
   (b) Order the units in the layer according to their "optimal" y-positions
4. Scale and center and all layers to fit the print area

We implemented this algorithm in the ANN library, and used it in producing the network figures for this work. A major problem with the algorithm can be observed when the network is, for example, connected so that each hidden neuron is connected to the neuron with the successive index value. The hidden neurons are then ordered one neuron per layer, at the center of the layer, and the connections thus overlap each other.

Appendix B

The evolutionary algorithm library

We developed a evolutionary algorithm library called *Cakra* to carry out the experiments of this study. We implemented it as a C++ library.

We designed the library using a heavily object-oriented approach, with good generality as the primary goal. The genetic code is typically just a bit string (or something equally uniform) in most genetic algorithms. The bit string is decoded by some procedure and then evaluated using a fitness function. We took an approach that the genetic code is a potentially very complex data structure. The elements of that structure must have an implementation of the basic genetic operators (recombination and mutation). When such operators are applied to the top-level container (typically a genome), they are applied recursively to all the lower-level genetic structures, and any parameters (such as mutation rates) can be modified by the specific implementations of the operators at each structural level. Another, a rather agent-oriented, feature is that although the genetic code can be read as usual by a "decoding function", the genetic structures (genes and genetic containers) can be sent a message that "activates" them and they build the phenotype of their host individual just by themselves, possibly by activating other genes. Many totally distinct phenotypic features can be designed in a modular manner, and then just "thrown in" in the genome, and the individual's ontogenetic procedure will take care of the growth process.

We took also a more "individual-centered" view of the selection process in the library. Typically the selection is handled from a "breeder's" point of view. In our approach the individuals assign a selection eagerness for each other individual, which are transformed into propabilities for selecting a particular pair. Without autoadaptation of the selection parameters (as such was not used in this work), this is equal to the normal "breeder's" view of selection.

The top-level class hierarchy of the library is shown in Figure B.1. The abstract genetic structure is impelented as the class GENSTRUCT, and the basic container as GENTAINER. The descendants of the GENE are intented to be "atomic genetic structures", and most user implementations are expected to inherit the GENTAINER. The two numeric gene types, ANYFLOATGENE for floating-point genes and ANYINTGENE for integer-valued genes, are divided in two forms. For example, the BITFLOATGENE is implemented using the tra-

ditional GA representation with binary genes, while the FLOATGENE uses floating-point representation directly, as in Evolution Strategies (see Chapter 3). The phenotype of the individuals can consist of any kind of objects, identified by a name. The GAENVIRONMENT is the environment where the individuals of the population evolve in. The details of the library will be explained further in a separate documentation.

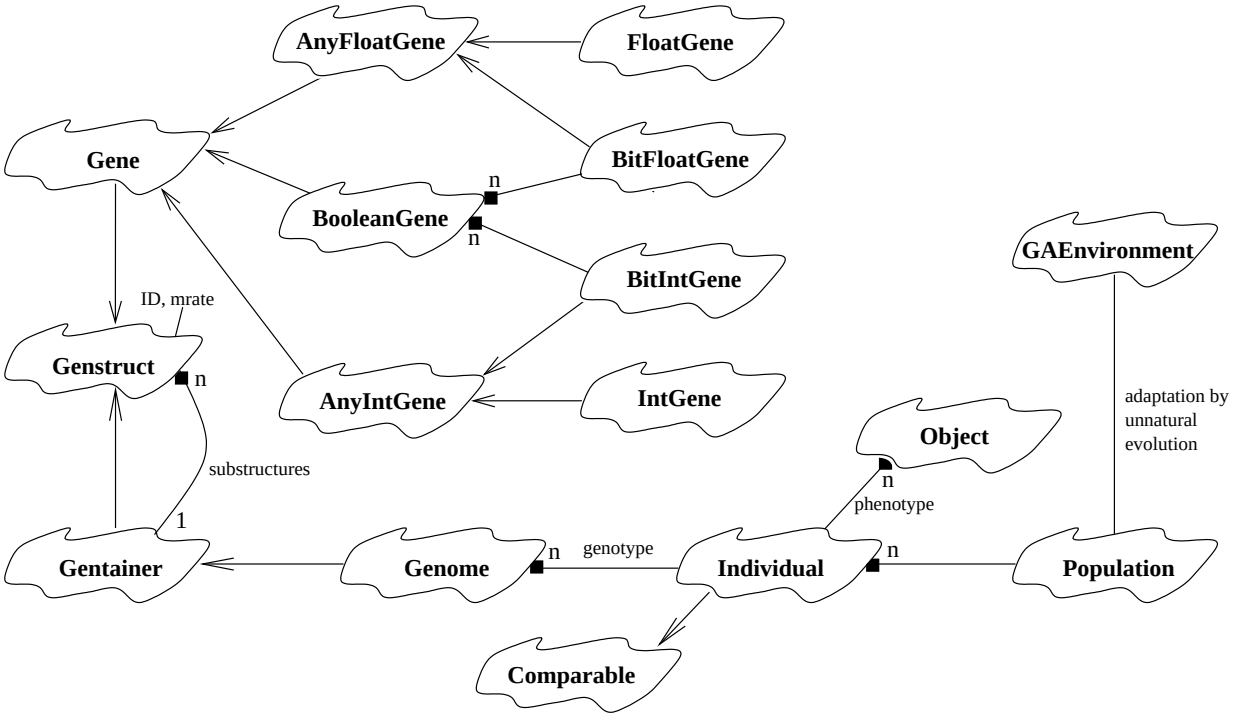

Figure B.1: Cakra's top-level class hierarchy in Booch-style relationship notation; the arrows represent inheritance (as "B inherits A") and lines with squares containing (as "B contains A"). A reference relationship is represented with a plain line.

Appendix C

Evolutionary neural network program library

We developed a C++ program library called *Annalee* for the evolutionary engineering of artificial neural network architectures. We used the library to perform the experiments of this study. It uses the agent-oriented genomic architecture of the Cakra library described in Appendix B. The neural network encoding methods are implemented as genetic structures. When activated during the ontogenesis of an individual, the structures construct the "brain" and the input/output structures of the individual. The basic class hierarchy of the library is illustrated in Figure C.1. Top-level baseclasses are omitted from the figure, as are the phenotypic object classes. Also, the encoding method objects dynamically contain numerous primitive genes that are not shown here. The ANNGene and LearningIOGene are interface classes that were required mostly because of limitations in SNNS (see Appendix A). A more detailed description will be given in a separate documentation.

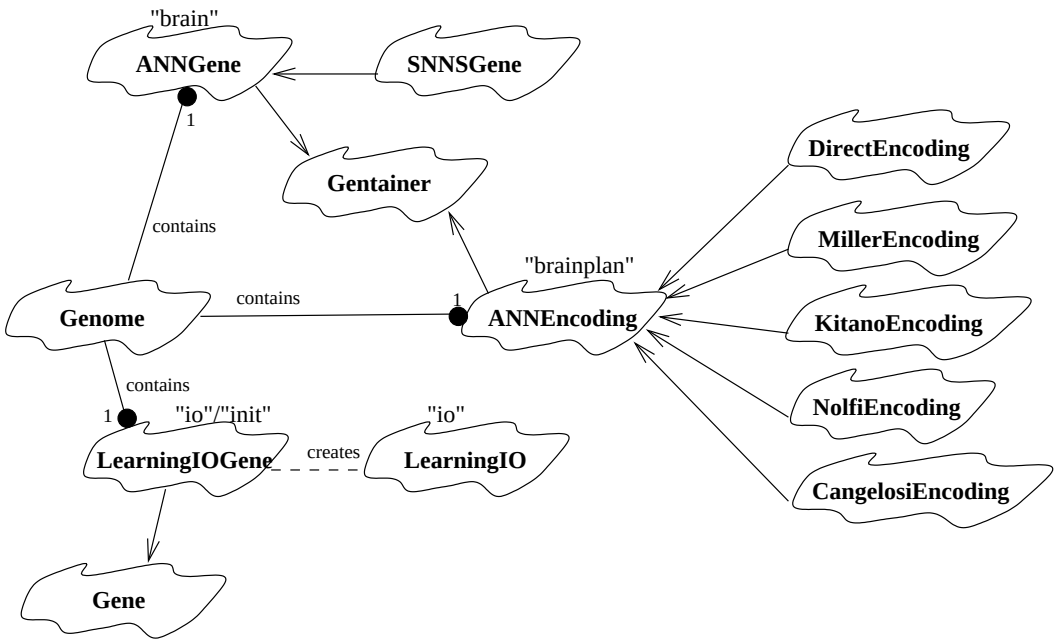

Figure C.1: Annalee's top-level class hierarchy. See Appendix B for description of the notation. The ball-ended lines denote dynamic run-time containing. The names within parentheses are gene labels.

Bibliography

Back, B., K. Sere, and T. Laitinen (1997). Robustness aspects in bankruptcy predictions. In *Proceedings of the The 20th Annual Congress of the European Accounting Association, Graz, Austria.*

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice.* Oxford University Press.

Balakrishnan, K. and V. Honavar (1995, January). Evolutionary design of neural architectures — a preliminary taxonomy and guide to literature. Technical Report CS TR 95–01, Department of Computer Science, Iowa State University, Ames, IA 50011.

Bishop, C. (1995a). *Neural Networks for Pattern Recognition.* Oxford University Press.

Bishop, C. (1995b). Regularization and complexity control in feed-forward networks. Technical Report NCRG/95/022, Neural Computing Research Group, Dept. of Computer Science and Applied Mathematics, Aston University, Birmingham, UK. (Also in: http://neural-server.aston.ac.uk/).

Boers, E. and H. Kuiper (1992). Biological metaphors and the design of modular artificial neural networks. (Also: http://www.wi.leidenuniv.nl/MScThesis/boers-kuiper.html).

Branke, J. (1995). Evolutionary algorithms for neural network design and training. In *1st Nordic Workshop on Genetic Algorithms and its Applications*, Vaasa, Finland, January 1995. (Also: ftp://ftp.aifb.uni-karlsruhe.de/pub/jbr/Vaasa.ps.gz).

Cangelosi, A. and J. Elman (1995). Gene regulation and biological development in neural networks: An exploratory model. Technical Report CRL-UCSD, University of California, San Diego. (Also: http://gracco.irmkant.rm.cnr.it/angelo/cang-pub.htm).

Cangelosi, A., D. Parisi, and S. Nolfi (1993). Cell division and migration in a 'genotype' for neural networks. *Network: Computation in Neural Systems 5*, 497–515.

Dawkins, R. (1986). *The Blind Watchmaker.* New York: Norton and Cooper.

Dellaert, F. (1995). Toward a biologically defensible model of development. Master's thesis, Case Western Reserve University, Dept. of Computer Engineering and Science. (Also in: http://www.cs.cmu.edu/afs/cs.cmu.edu/user/dellaert/web/publications.html).

Dellaert, F. and R. Beer (1994). *Toward an Evolvable Model of Development for Autonomous Agent Synthesis.* MIT Press Cambridge. (Also in: http://www.cs.cmu.edu/afs/cs.cmu.edu/user/dellaert/web/publications.html).

Duane, D. (1996). A relationship between amorphous neural network size and problem complexity. Master's thesis, George Mason University, Fairfax, Virginia.

Fahlman, S. E. and C. Lebiere (1991). The cascade-correlation learning architecture. Technical Report 189, School of Computer Science, Carnegie-Mellon University.

Fredriksson, K. (1997). Genetic algorithms and generative encoding of neural networks for some benchmark classification problems. In *Proceedings of the Third Nordic Workshop on Genetic Algorithms and their Applications (3NWGA)*, pp. 123–134. (Also: ftp://ftp.uwasa.fi/pub/cs/3NWGA/Fredriksson.ps.Z).

Friedman, J. (1991). Multivariate adaptive regression splines. *The Annals of Statistics 19*(1), 1–141.

Fullmer, B. and R. Miikkulainen (1991). Using marker-based genetic encoding of neural networks to evolve finite-state behaviour. In *Proceedings of the First European Conference on Artificial Life (ECAL-91)*. (Also: http://www.cs.utexas.edu/users/nn/pages/publications/neuro-evolution.html).

Gruau, F. (1994). *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm*. Ph. D. thesis, l'Ecole Normale Superieure de Lyon. Also: ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/PhD/PhD94-01-E.ps.Z.

Gruau, F. and D. Whitley (1993). Adding learning to the cellular developmental process: A comparative study. Technical report RR93-04, Laboratoire de l'Informatique du Parallilisme. Ecole Normale Superieure de Lyon. (Also in: http://www.cwi.nl/~gruau/gruau/node4.html).

Harp, S., T. Samad, and A. Guha (1989). Towards the genetic synthesis of neural networks. In J. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 360–369. Morgan Kaufmann.

Hebb, D. (1949). *The Organization of Behavior*. Wiley, New York.

Holland, J. (1975). *Adaptation in natural and artificial systems*. The University of Michigan Press, Ann Arbor, MI.

Jacobs, R. A., M. I. Jordan, and A. G. Barto (1990). Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks. Technical report, Department of Computer & Information Science, University of Massachusetts, Amherst, MA 01003.

Kauffman, S. (1993). *The Origins of Order: Self-Organization and Selection in Evolution*. New York: Oxford University Press.

Kauffman, S. (1995). *At Home in the Universe: The Search for Laws of Complexity*. New York: Oxford University Press.

Kitano, H. (1990a). Designing neural network using genetic algorithm with graph generation system. *Complex Systems 4*, 461–476.

Kitano, H. (1990b). Designing neural networks using genetic algrithms with graph generation system. Technical report, Center for Machine Translation , Carnegie Mellon University, Pittsburgh and NEC Corporation, Tokyo, Pittsburgh, PA.

Kodjabachian, J. and J. Meyer (1995). Evolution and development of control architectures in animats. *Robotics and Autonomous Systems 16:2-4*, 161–182.

Koza, J. (1990). Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Department of Computer Science, Stanford University, Stanford, CA. (Also: ftp://elib.stanford.edu/pub/reports/cs/tr/90/1314/CS-TR-90-1314.ps).

Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, MA, 1992.

Lapedes, A. and R. Farber (1988). How neural nets work. In D. Z. Anderson (Ed.), *Neural Information Processing Systems*, New York, NY. American Institute of Physics. Conf. held Nov. 1987 in Boulder, CO.

Lindenmayer, A. and G. Rozenberg (1976). *Automata, Languages, Development*. Amsterdam, North-Holland.

McCulloch, W. and W. Pitts (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics* (5), 115–133.

Miller, G., P. Todd, and S. Hegde (1989). Designing neural networks using genetic algorithms. In J. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 379–384. Morgan Kaufmann.

Minsky, M. and S. Papert (1969). *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, Cambridge, MA.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press.

Moody, J. (1994). Prediction risk and architecture selection for neural networks. In J. F. V. Cherkassky and H. Wechsler (Eds.), *From Statistics to Neural Networks: Theory and Pattern Recognition Applications*, NATO ASI Series F. Springer-Verlag.

Nolfi, S. and D. Parisi (1992). Growing neural networks. Technical report, Institute of Psychology, CNR, Rome.

Nolfi, S. and D. Parisi (1994). Genotypes for neural networks. In M. A. Arbib (Ed.), *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA: MIT Press, a Bradford book.

Prechelt, L. (1994). PROBEN1 - a set of neural network benchmark problems and benchmarking rules. Technical Report 21/94, Fakultät für Informatik, Universität Karlsruhe, 76128 Karlsruhe, Germany. (Also in: http://wwwipd.ira.uka.de/~prechelt/).

Prechelt, L. (1998). Automatic early stopping using cross validation. *Neural Networks 11*. (To appear) (Also in: http://wwwipd.ira.uka.de/~prechelt/biblio.html).

Riedmiller, M. (1993). A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Proceedings of the IEEE International Conference on Neural Networks*. (Also in: http://i11www.ira.uka.de/riedml/publications.html).

Rosenblatt, F. (1958). The perceptron: A propabilistic model for information storage and organization in the brain. *Psychological Review 65*, 386–408.

Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Washington DC: Spartan.

Rumelhart, D., G. Hinton, and R. Williams (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1: Foundations, Chapter

Learning internal representations by error propagation, pp. 318–362. The MIT Press, Cambridge, MA.

Sarle, W. (1995). Stopped training and other remedies for overfitting. In *Proceedings of the 27th Symposium on the Interface*. (Also in: ftp://ftp.sas.com/pub/neural/).

Schwefel, H.-P. (1977). Numerische optimierung von computer-modellen mittels der evolutionsstrategie. *Interdisciplinary Systems Research 11*.

Siddiqi, A. A. and S. M. Lucas (1998). A comparison of matrix rewriting versus direct encoding for evolving neural networks. In *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation (ICEC'98)*, pp. 392–397. IEEE.

SNNS (1995). SNNS, Stuttgart Neural Network Simulator, User Manual, Version 4.1. Technical Report 6/95, Institute for Parallel and Distributed High Performance Systems (IPVR), University of Stuttgart.

Vaario, J. (1993). *An Emergent Modeling Method for Artificial Neural Networks*. Ph. D. thesis, The University of Tokyo.